UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FELIPE ZORZO PEREIRA

Avaliação de concorrência e sincronização de diferentes linguagens de programação populares

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Claudio Fernando Resin Geyer

Porto Alegre
2021

AGRADECIMENTOS

Em primeiro lugar agradeço à minha vozinha Olga e ao meu vô Oracides, que se foram antes de me verem formado, pela linda passagem que tiveram na minha vida. Minha vozinha Olga basicamente me criou quando eu era criança; era muito bom e feliz sair da escola e ir direto à sua casa. Quanto ao meu vô, nunca vou esquecer de quando chegava em sua casa e ia correndo receber um abraço e ouvir "ôôô, meu neto querido". Não consigo definir se "saudade" é a dádiva ou a maldição da língua portuguesa, mas é o que sinto em relação a vocês. Te amo, vozinha; te amo, vozinho.

Agradeço à minha mãe Odete por tudo. Pelas horas que passou me ajudando a estudar para as provas da escola, por me ensinar e me fazer amar matemática, por sempre ter feito o máximo para me dar do bom e do melhor, por ter me dado todo o amor e carinho do mundo, por sempre ter me incentivado a seguir meus sonhos e por aguentar todas as minhas louquices. Te amo, mãe.

Agradeço ao meu pai Ricardo por todo o amor, por todas as brincadeiras, por todas as risadas, pelos churrascos aos fins de semana e por ter se desdobrado para continuarmos nos vendo mesmo após não morarmos mais juntos. Te amo, pai.

Agradeço à minha vó Therezinha por todos os ótimos momentos que passamos juntos, em especial na sua antiga casa, e por ter cuidado do Tutty por tantos anos. Como eu amava ir te visitar, jogar futebol no pátio da sua casa e brincar com o Tutty! Te amo, vozinha. Aproveito para também agradecer ao meu falecido cachorrinho Tutty pela companhia de anos e por todas as brincadeiras. Lembro que, quando descobri que meu vô Oracides havia falecido, o Tutty veio correndo para mim e sentou no meu colo; ficar contigo naquele momento foi muito reconfortante. Te amo, Tutty.

Agradeço à minha mana Amanda e ao meu mano Ricardinho por todos os momentos bons que já tivemos. Lembro que quando eu estava na escolinha, antes dos meus seis anos, as sextas-feiras eram meus dias favoritos da semana porque eu sabia que logo mais iríamos nos ver e passar o fim de semana juntos; eu fazia questão de contar isso às minhas professoras. Amo vocês.

Agradeço ao meu dindo Ivan por todos os momentos alegres que me proporcionou, assim como por sempre me incentivar a ler e estudar. Lembro que, quando criancinha, eu sempre chorava quando ele pegava um avião para sair de Porto Alegre. Da mesma forma,

agradeço à minha tia Sinara por ter feito parte da minha criação na casa da vó Olga. Amo vocês, dindo e tia.

Agradeço a todos os professores que encontrei durante o meu tempo de UFRGS pelos ensinamentos compartilhados. Em especial, agradeço aos professores Cláudio Geyer, Raul Weber e Renata Galante por serem pessoas exemplares que, além de ensinar, sempre fizeram questão de me ajudar. Deixo aqui minha homenagem póstuma ao professor Weber: lembro que minha primeira prova na UFRGS foi com ele e que ele sempre estava disponível para me ajudar, mesmo com conteúdos de cadeiras não palestradas por ele.

Também agradeço às professoras Vera Cáceres, Lauci Belle e Maria Lipinski, que fizeram parte da minha vida escolar e também sempre me ajudaram e me trataram com todo o carinho do mundo. Além de excelentes profissionais, são seres humanos incríveis.

Agradeço ao grupo de amigos que fiz durante o curso de Ciência da Computação: Bernardo, Rafael, João, William, Bruno e Leonardo. Bernardo, com seu jeito grosso, mas sempre rindo. Rafael, com seu jeito sério, mas sempre muito companheiro. William, sempre quietinho e sempre minha dupla nos trabalhos. João, sempre contando piadas e falando bobagens para nos fazer rir. Bruno, com seu jeito distante e misterioso que cativava e arrancava risadas de todos. Leonardo, com seu jeito bobo, mas completamente parceiro e sempre disposto a ouvir meus desabafos, em especial no período de pandemia. Foi muito bom ter a companhia de vocês durante as aulas, durante os almoços no RU e durante as caronas, sempre regadas a muita música, até o *campus*. As saídas para comer e ir ao cinema, então, foram inesquecíveis.

Agradeço aos meus diferentes grupos de amigos da época de escola pelos muitos anos de amizade e por terem feito parte do meu amadurecimento. Agradeço à Caroline e ao Alberto por serem tão sensíveis, ouvirem meus desabafos e compartilharem do meu amor por Harry Potter. Agradeço ao Vicente por me ditar o que os professores escreviam no quadro e por me passar as suas anotações antes de eu começar a usar óculos, além de ter sido um amigo incrível. Agradeço ao Henrique Vieira por reunir todos os nossos amigos em seus churrascos, assim como pelas manhãs que começávamos jogando bola antes das aulas. Foi muito bom te encontrar inesperadamente no T8 em meu primeiro dia de UFRGS; obrigado por ter me levado às minhas primeiras salas de aula. Agradeço ao Henrique Bilo por toda a companhia ao fim do Ensino Médio; foi um momento muito feliz quando descobri que eu voltaria a ter a tua companhia na Ciência da Computação da UFRGS! Agradeço aos basqueteiros Thiago, Roberto e Pedro pelos tantos anos de amizade, risadas e trabalhos em grupo; sempre foi bom saber que, independentemente do trabalho proposto, era só olhar para vocês que já tínhamos um grupo de amigos para trabalhar.

Também agradeço ao meu grupo de amigos do Counter-Strike: Dique, Finha, Laninha, Maskavo e Taly. Nunca imaginei que amizades que fiz na internet poderiam durar tantos anos! Obrigado por todas as risadas e por todos os momentos em que me ajudaram a descansar e relaxar depois de dias cansativos; jogar com vocês era uma das melhores partes dos meus dias. Amo relembrar que chegamos à final de um campeonato gigante, a Liga Amadora da Gamers Club, no mesmo dia em que comecei meus estudos na UFRGS.

Por fim, agradeço à minha grande amiga e ex-namorada Amanda, que conheci numa das pausas para descansar desse trabalho. Obrigado por ser tão frequentemente a primeira vez de algo na minha vida. Você me apresentou muitas coisas boas da vida e os momentos que vivemos juntos foram inesquecíveis. Obrigado por ter agitado minha vida e por ter sido a minha companhia diária por tanto tempo.

RESUMO

Concorrência é um conceito essencial para a escrita de programas reativos, rápidos e interativos. Atualmente, entretanto, existe uma grande quantidade de linguagens de programação que oferecem mecanismos de concorrência. A partir daí surge o objetivo desse trabalho: realizar uma avaliação sistemática de mecanismos de concorrência e sincronização de algumas das mais populares linguagens de programação de propósito geral, a fim de ajudar programadores a fazerem decisões informadas sobre quais linguagens melhor se adequam às suas necessidades de concorrência. As linguagens analisadas nesse trabalho são C++, Go, Java, Kotlin e Scala. A avaliação delas é feita com base em dados obtidos sobre as métricas de *overhead* da concorrência e da sincronização, *speedup* e justiça. Tais dados foram coletados por meio da execução de programas implementados usando as linguagens e os seus mecanismos. Para analisar os *overheads* simplesmente se comparou o desempenho de laços vazios e de laços que criam tarefas concorrentes ou entram e saem de seções críticas; para o *speedup* implementou-se a multiplicação de matrizes; por fim, para a justiça foi utilizado o problema dos filósofos. Os programas foram executados em duas máquinas com processadores e sistemas operacionais distintos, a fim de confirmar que as conclusões obtidas não são dependentes da plataforma utilizada. Os resultados demonstram que as linguagens com menos *overhead* de concorrência foram Go e Kotlin, com Scala ficando muito atrás das demais linguagens. Os *overheads* de sincronização de todas as linguagens são extremamente baixos, sendo aceitáveis para a grande maioria das aplicações; apesar disso, Scala ficou bem atrás das demais linguagens. Quanto ao *speedup*, considera-se que as *threads* de Java obtiveram os resultados mais constantes ao longo dos experimentos nas duas máquinas. Os experimentos para avaliar a justiça das linguagens obtiveram resultados bem esparsos, sendo difícil escolher uma linguagem como sendo a melhor. Entretanto, Java, assim como as linguagens baseadas em Java (Scala e Kotlin), se destacaram. Por fim, conclui-se que Java e Kotlin foram as linguagens com os melhores resultados gerais. Se o maior objetivo da concorrência é proporcionar *speedup*, recomenda-se Java; se, por outro lado, o programa exige a criação de dezenas ou centenas de milhares de tarefas concorrentes, recomenda-se Kotlin.

Palavras-chave: Concorrência. Sincronização. Paralelismo. C++. Go. Java. Kotlin. Scala. *Goroutines*. Coroutines. Threads. *Mutex*. Speedup.

Concurrency and Synchronization Evaluation of Different Programming Languages

ABSTRACT

Concurrency is an essential concept to the writing of reactive, fast and interactive programs. Nowadays, however, a great variety of programming languages offers concurrency mechanisms. From this emerges the objective of this work: to perform a systematic evaluation of concurrency and synchronization mechanisms of some of the most popular general-purpose programming languages, in order to help programmers make informed decisions on which languages best suit their needs for concurrency. The languages analyzed in this work are C++, Go, Java, Kotlin and Scala. Their evaluation is done based on data collected about their overhead, speedup and fairness. This data was collected through the execution of programs implemented using the languages and their mechanisms. To analyze the overhead of the languages, empty loops were compared to loops that create concurrent tasks or enter and leave critical sections; to analyze the speedup, matrix multiplication was implemented; lastly, the dining philosophers problem was used to analyze the fairness of the languages. The programs were executed in two different machines with different processors and operating systems, with the intention of confirming that the conclusions obtained are not dependent on the underlying platform. The results show that the languages with the least concurrency overhead are Go and Kotlin, with Scala's performance being way behind the other languages. The synchronization overhead of all the languages is extremely low, being acceptable to the majority of applications; Scala, however, stood way behind the other languages once again. Regarding the speedup it is considered that Java's threads had the most consistent results during the experiments on both machines. The experiments that evaluate the languages' fairness had very sparse results. Therefore, it is difficult to choose one language as the fairest. Despite of that, Java, as well as the Java-based languages (Scala and Kotlin), stood out. Finally, it is concluded that Java and Kotlin were the languages with the best overall results. If the main objective of concurrency is speedup, Java is recommended; if, on the other hand, the program requires the creation of dozens or hundreds of thousands of concurrency tasks, Kotlin is recommended.

Keywords: Concurrency. Synchronization. Parallelism. C++. Go. Java. Kotlin. Scala. *Goroutines*. Coroutines. Threads. *Mutex*. Speedup.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
I/O	Input/Output
CPU	Central Processing Unit
EPFL	École polytechnique fédérale de Lausanne
FIFO	First In, First Out
JVM	Java Virtual Machine
RAM	Random Access Memory

SUMÁRIO

1 INTRODUÇÃO

Concorrência, no contexto da computação, ocorre quando há sobreposição na execução de duas ou mais tarefas. Apesar dos problemas provenientes da programação concorrente, como a dificuldade de se controlar o fluxo do programa, suas vantagens são inegáveis. Escrever programas reativos que permitem interação enquanto tarefas rodam em *background*, impedir que tarefas longas atrasem a execução de tarefas curtas e criar tarefas que rodam em paralelo são alguns dos benefícios propiciados pela programação concorrente. Tais características se traduzem em programas mais fluidos, rápidos e interativos, beneficiando seus usuários finais.

Atualmente existe uma gama enorme de linguagens de programação que fornecem mecanismos de concorrência a seus utilizadores. Além de permitir a criação de tarefas concorrentes para que os programadores usufruam de seus benefícios, tais linguagens também oferecem mecanismos de sincronização para controlar o fluxo não-determinístico de programas concorrentes e paralelos. Entretanto, diferentes linguagens oferecem diferentes mecanismos para a criação de tarefas e para a sincronização das mesmas.

A grande diversidade de linguagens de programação que permitem concorrência é bem-vinda: proporciona flexibilidade aos programadores e torna mais acessível o acesso à programação concorrente. Entretanto, cada mecanismo e linguagem possui seus benefícios e desvantagens que devem ser levadas em consideração ao escolher quais linguagens e mecanismos utilizar para implementar algum algoritmo.

1.1 Objetivos

O objetivo desse trabalho é comparar e avaliar mecanismos de concorrência e sincronização de diferentes linguagens de propósito geral, a fim de auxiliar programadores a tomarem decisões informadas e adequadas antes de iniciarem a implementação de seus algoritmos concorrentes.

No contexto desse trabalho, foram escolhidas cinco populares linguagens de programação de propósito geral que fornecem concorrência e sincronização de forma simples. Avaliou-se, para cada linguagem, alguns de seus mecanismos de concorrência e sincronização. Tais linguagens e mecanismos serão detalhados adiante, assim como as métricas utilizadas para avaliá-los e compará-los.

1.2 Importância do trabalho atual

A importância desse trabalho se comparado aos trabalhos anteriores se deve ao fato de ele analisar uma variedade grande de linguagens de programação populares. Além disso, assim como o trabalho de Vargas (2019), esse trabalho se propõe a também analisar mecanismos de sincronização; tais mecanismos são essenciais para muitos algoritmos concorrentes.

Por fim, os demais trabalhos utilizam métricas como número de linhas do código-fonte e tempo de execução, que é relacionado ao desempenho sequencial da linguagem. O presente trabalho, por sua vez, utiliza apenas métricas estritamente relacionadas à programação concorrente, como será apresentado no capítulo 3. Com isso, esse trabalho se torna um guia geral para programadores, auxiliando-os a comparar diferentes linguagens e escolher aquela que melhor satisfaz suas necessidades de concorrência.

1.3 Escolha das linguagens e seus mecanismos

A escolha das linguagens avaliadas nesse trabalho se baseou nas linguagens escolhidas para serem avaliadas nos trabalhos relacionados. A finalidade disso foi obter linguagens já consolidadas para a programação concorrente. A lista de linguagens observadas ao longo dos trabalhos relacionadas foi: Chapel, Cilk, C++, Eiffel, Erlang, Go, Java e Kotlin.

O presente trabalho se propõe a avaliar linguagens de programação populares, a fim de auxiliar a maior quantidade possível de programadores. Portanto, das oito linguagens observadas nos trabalhos relacionados, apenas aquelas que estão dentre as 25 linguagens mais populares entre programadores (Anexo A), de acordo com a pesquisa realizada pelo site Stack Overflow, foram escolhidas para serem avaliadas. São elas: C++, Go, Java e Kotlin. Além dessas linguagens, optou-se por também avaliar Scala, já que ela é uma linguagem baseada em Java e, portanto, possui semelhanças com Java e Kotlin, que também é baseada em Java.

Por fim, escolheu-se os mecanismos de concorrência e sincronização a serem avaliados para cada uma das linguagens já selecionadas. Para a concorrência, com exceção de C++, optou-se pelo mecanismo de criação de tarefas concorrentes mais simples de cada linguagem. Para C++, além do mecanismo simples de criação de *threads*, também foi utilizada a API OpenMP, conforme sugerido por Silveira (2012).

Os mecanismos de sincronização, por sua vez, foram escolhidos com base nos programas utilizados para avaliar as métricas de interesse desse trabalho (apresentadas no

capítulo 3). Alguns dos mecanismos de sincronização disponibilizados pelas linguagens não costumam ser aplicados para solucionar os problemas utilizados para quantificar as métricas de interesse desse trabalho, conforme será exemplificado na subseção 3.2; portanto, tais mecanismos não foram analisados.

Com isso, a relação completa de linguagens e seus mecanismos de concorrência e sincronização avaliados está disposta na Tabela 1 abaixo. As versões das linguagens ou de seus compiladores também são apresentadas.

Tabela 1 — Mecanismos de concorrência e sincronização por linguagem

Linguagem (versão)	**Mecanismo de concorrência**	**Mecanismo de sincronização**
C++ (gcc 8.1.0)	*Threads* (OpenMP)	*Locks* (OpenMP)
	Threads (biblioteca padrão)	*Mutex*
Go (go1.13.5)	*Goroutines*	*Mutex*
Java (13.0.1)	*Threads*	Bloco *synchronized*
		Locks injustos
		Locks justos
		Semáforos injustos
		Semáforos justos
Kotlin (1.3.70)	*Coroutines*	Bloco *synchronized*
		Mutex
		Semáforos
Scala (2.13.1)	*Futures*	Bloco *synchronized*
		Locks injustos
		Locks justos
		Semáforos injustos
		Semáforos justos

1.4 Estrutura desse documento

O capítulo 2 desse trabalho discorre sobre alguns conceitos essenciais ao seu entendimento, assim como descreve as linguagens e mecanismos avaliados e, por fim, apresenta alguns trabalhos relacionados. O capítulo 3 discorre sobre a metodologia para a avaliação de desempenho das linguagens e seus mecanismos. O capítulo 4, por sua vez, expõe e analisa os resultados obtidos com os experimentos. Por fim, o capítulo 5 apresenta as conclusões do presente trabalho.

2 CONCEITOS E TRABALHOS RELACIONADOS

Esse capítulo apresenta conceitos relacionados à concorrência e necessários ao entendimento desse trabalho. Além disso, as linguagens avaliadas também serão brevemente apresentadas.

2.1 Concorrência e paralelismo

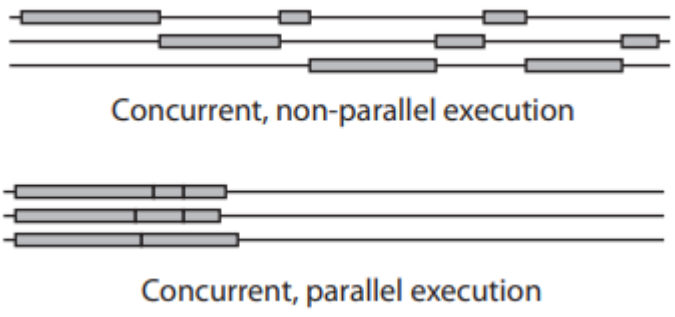
Concorrência, no contexto da computação, é uma condição que ocorre quando duas ou mais tarefas computacionais são executadas em períodos de tempo sobrepostos. Com isso, numa mesma unidade de tempo é possível que mais de uma tarefa esteja disponível para ser executada.

Paralelismo, por sua vez, é uma condição que ocorre quando duas ou mais tarefas computacionais são executadas simultaneamente. Para que isso seja possível, é necessário que o sistema possua múltiplas unidades de processamento; portanto, é necessário suporte de *hardware*.

Pelas definições, percebe-se que tarefas concorrentes podem ser paralelizadas caso o *hardware* permita. Além disso, caso o *hardware* não suporte paralelismo, por meio da divisão dos tempos de execução é possível simular paralelismo, de forma que o usuário final acredite que duas ou mais tarefas estão executando simultaneamente quando, na verdade, elas só estão executando de forma intercalada.

A Figura 1 ilustra as diferenças entre concorrência e paralelismo. Na parte superior da imagem pode-se ver três tarefas executando de forma concorrente, mas não paralela; a parte inferior, por sua vez, mostra três tarefas concorrentes executando de forma paralela.

Figura 2.1 — Concorrência e paralelismo



(SOTTILE et al., 2009)

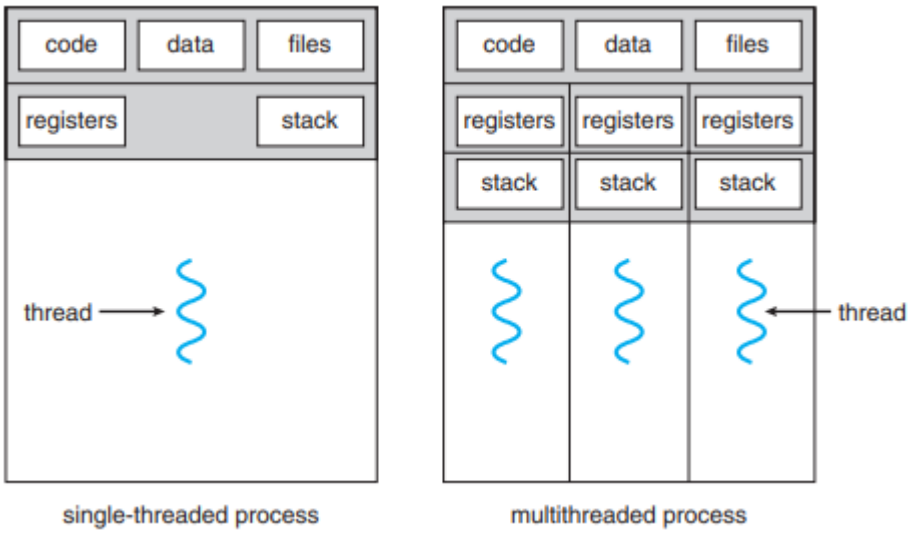
2.1.1 Processos e *Threads*

Processos podem ser entendidos como um programa em execução que necessita de determinados recursos, como tempo de CPU, memória, arquivos e acesso a dispositivos de I/O. Um processo também engloba o seu *program counter*, o conteúdo de seus registradores, uma pilha, seu segmento de código, seu segmento de dados e possivelmente um *heap* (SILBERSCHATZ et al., 2018).

Threads, por sua vez, são partes ou fluxos de programas que rodam concorrentemente e dividem o mesmo espaço de memória (LEE; SESHIA, 2017). *Threads* possuem um ID, um *program counter*, um conjunto de registradores e uma pilha; além disso, *threads* pertencentes a um mesmo processo compartilham o mesmo segmento de código, segmento de dados, *heap* e outros recursos do sistema operacional alocados ao processo (SILBERSCHATZ et al., 2018).

Tradicionalmente processos eram *single-threaded*, ou seja, possuíam apenas uma *thread*. Portanto, para uma aplicação ser concorrente ou paralela era necessária a criação e o gerenciamento de vários processos. Todavia, a maioria das aplicações que rodam em computadores modernos são compostas por um processo *multithreaded*, ou seja, que possui várias *threads*. A Figura 2 demonstra a diferença de um processo single-*threaded* para um *multithreaded*.

Figura 2.2 — Processo *single-threaded* e processo *multithreaded*



(SILBERSCHATZ et al., 2018)

Essa mudança de *single-threaded* para *multithreaded* se deve aos benefícios de um processo possuir várias *threads*. Um dos benefícios é o fato de que *threads* dividem o espaço de memória e os recursos de seu processo. Portanto, enquanto processos precisam comunicar

entre si para compartilhar recursos, o que é custoso, *threads* compartilham os recursos do processo ao qual pertencem sem custo adicional.

Threads são consideradas mais leves do que processos, já que compartilham recursos como segmento de dados, segmento de código e *heap*. Com isso surge outro benefício proporcionado por um processo *multithreaded*: a diminuição do custo de gerenciamento se comparado a vários processos. Enquanto o gerenciamento de vários processos (criação, destruição, mudança de contexto) é muito custoso, o gerenciamento de várias *threads* é mais econômico (SILBERSCHATZ et al., 2018).

2.1.2 *Thread Pool*

Apesar de serem mais leves que processos, a criação e a destruição de *threads* não possuem custo negligenciável; logo, criar e destruir algumas centenas de *threads* pode não ser eficiente. Uma forma de lidar com isso é utilizar uma *thread pool*. Uma *thread pool* consiste de um número fixo, normalmente igual à quantidade de núcleos do processador, de *threads* às quais são delegadas tarefas de forma fácil e eficiente (KRIEMANN, 2004). As *threads* numa *thread pool* são reusadas, de forma que o *overhead* para a criação e destruição ocorre somente uma vez por *thread* na *pool* (LING, 2000).

2.2 Sincronização

Apesar dos benefícios trazidos pela concorrência, com ela também surgem novos problemas. Um desses problemas é chamado de condição de corrida. Uma condição de corrida ocorre quando duas ou mais tarefas acessam e manipulam um mesmo dado de forma concorrente e o resultado da execução depende da ordem em que o acesso ao dado ocorre (SILBERSCHATZ et al., 2018).

Portanto, a fim de se evitar condições de corrida e garantir a corretude de um programa, faz-se necessária a sincronização do acesso aos recursos compartilhados, de forma que apenas uma tarefa manipule determinado recurso em um dado momento.

Para atingir esse objetivo são utilizados mecanismos de sincronização que definem o início e o fim de seções críticas. Uma seção crítica é um segmento de código em que uma tarefa concorrente pode estar manipulando recursos compartilhados. Os mecanismos de sincronização devem, então, garantir a propriedade de exclusão mútua da seção crítica; tal propriedade afirma

que quando uma tarefa está em sua seção crítica, nenhuma outra tarefa pode estar em sua seção crítica.

Apesar da necessidade de sincronização para a corretude de programas concorrentes e paralelos, ela traz consigo novas dificuldades. Um dos problemas que pode surgir devido à sincronização é o *deadlock*. *Deadlock* é uma situação que surge quando recursos foram alocados a várias tarefas de forma que nenhuma delas consegue continuar sua execução (HABERMANN, 1969). Por exemplo, *deadlock* ocorre quando tarefas concorrentes possuem alguns dos recursos necessários para começar a sua execução, mas nenhuma consegue começar porque os recursos faltantes já estão na posse de alguma das outras tarefas que também estão aguardando por recursos que nunca receberão.

Outro problema que pode surgir com a sincronização é denominado *starvation*. *Starvation* ocorre quando uma tarefa aguarda por um recurso, mas nunca o recebe, uma vez que sempre que o recurso se torna disponível existe outra tarefa que é mais favorecida pela política de escalonamento e recebe o recurso (HSIEH; UNGER, 1988).

2.2.1 *Mutex* ou *locks*

Mutex, chamado por algumas linguagens de programação de *lock*, é um mecanismo de sincronização simples. Um *mutex* é uma variável compartilhada que pode estar em um de dois estados: trancada ou destrancada (TANENBAUM; BOS, 2015).

Quando uma tarefa concorrente (seja um processo, uma *thread* ou outra construção) necessita acessar uma seção crítica, ela tenta obter a tranca do *mutex*. Se o *mutex* está destrancado, o pedido é atendido e a tarefa pode entrar em sua seção crítica. Quando uma tarefa obtém a tranca do *mutex*, ele passa para seu estado de trancado.

Por outro lado, caso o *mutex* esteja trancado, a tarefa que requisitou a sua tranca é bloqueada até que a tarefa que já está em sua seção crítica libere a tranca do *mutex*, destrancando-o. Quando uma tarefa libera a tranca do *mutex* e existem diversas tarefas esperado para entrar na seção crítica, apenas uma delas é escolhida para obter a tranca. Tal escolha depende da implementação do mecanismo.

2.2.2 Semáforos

Um semáforo é uma variável composta por um número inteiro que funciona como um mecanismo de sincronização. As únicas operações que podem ser realizadas em semáforos são

chamadas P e V (KOSARAJU, 1973). A operação P em um semáforo checa se o seu valor é maior que zero. Se sim, o valor do semáforo é decrementado em uma unidade e a tarefa que executou P continua a sua execução. Em contrapartida, se o valor do semáforo for menor ou igual a zero, a tarefa que tentou executar P é posta para dormir sem decrementar o valor do semáforo.

Quando uma tarefa executa a operação V, por sua vez, o valor do semáforo é incrementado. Quando o valor de um semáforo é incrementado e uma ou mais tarefas estavam aguardando nesse semáforo após tentar utilizar P, uma delas é escolhida e pode completar a execução de P no semáforo.

Essas operações sobre semáforos devem ser atômicas. Com isso quer-se dizer que as operações P e V devem ocorrer sem interrupção: quando uma tarefa começou a execução de uma operação sobre um semáforo, nenhuma outra tarefa pode acessar o semáforo até que a operação já iniciada seja finalizada.

Existem dois tipos principais de semáforo. O que será utilizado e analisado ao longo desse trabalho se chama semáforo binário. O valor de um semáforo binário possui a restrição de só poder variar entre um e zero. Consequentemente, um semáforo binário funciona como um *mutex* e pode ser utilizado para garantir a exclusão mútua de uma seção crítica.

O outro tipo de semáforo é denominado semáforo contador. Semáforos contadores, diferentemente dos semáforos binários, não possuem a restrição de poder variar apenas entre um e zero. Com isso, semáforos contadores podem ser utilizados para controlar o acesso a determinado recurso com um número finito de instâncias. Quando uma tarefa deseja uma instância de determinado recurso, executa P sobre o semáforo que guarda tal recurso. Caso exista alguma instância disponível, o valor do semáforo é decrementado. Caso o valor do semáforo já esteja em zero, não há nenhuma instância do recurso disponível, então a tarefa bloqueia e espera. Por fim, quando uma tarefa possui uma instância do recurso, mas não necessita mais dela, utiliza V sobre o semáforo, incrementando-o.

2.2.3 Barreiras

Uma barreira é um mecanismo para a sincronização da atividade de um número de tarefas concorrentes. Quando uma tarefa atinge a barreira, ela não pode prosseguir até que todas as demais tarefas também tenham atingido a barreira (SILBERSCHATZ et al., 2018). Isso permite que um grupo de tarefas concorrentes sincronize sua execução.

2.2.4 Variáveis de condição e sinalização

Enquanto *mutex* permite que tarefas concorrentes sincronizem ao controlar o seu acesso a um dado, uma variável de condição permite que tarefas concorrentes sincronizem de acordo com o valor do dado (NICHOLS et al., 1996). Com isso é possível fazer com que as tarefas sincronizem de acordo com condições arbitrárias definidas pelo programador.

Sem variáveis de condição, tarefas teriam que constantemente testar o valor de uma condição, possivelmente dentro de sua seção crítica, para descobrir se a condição foi atingida; isso é custoso. Devido à forma como as variáveis de condição funcionam, entretanto, isso é evitado: tarefas que não podem continuar sua execução devido a alguma condição insatisfeita são postas para dormir, evitando o gasto de recursos do sistema, e são acordadas por meio da sinalização feita por outras tarefas. Esse processo se dá por meio das operações *wait* e *notify*.

Uma variável de condição sempre possui um *mutex* associado. Uma tarefa obtém o *mutex*, entrando em sua seção crítica, e testa a condição da variável de condição. Se a condição for verdadeira, a tarefa pode completar a sua execução, liberando o *mutex* quando apropriado. Se for falsa, entretanto, a tarefa executa a operação *wait* na variável de condição; essa operação faz com que a tarefa libere o *mutex* e seja posta para dormir. Quando alguma outra tarefa muda algum aspecto da condição, ela executa a operação *notify* na variável de condição. Com isso, alguma das tarefas que estava dormindo na variável de condição é acordada e recebe o *mutex* associado a ela; após isso, tal tarefa reavalia a condição (LEWIS; BERG, 1996). Se a condição foi atendida, a tarefa pode finalmente continuar a sua execução; se a condição ainda não foi atendida, a tarefa volta a dormir na variável de condição.

2.3 Linguagens

Essa subseção fará uma breve apresentação das linguagens avaliadas nesse trabalho. Além das linguagens em si, seus mecanismos também serão descritos.

2.3.1 C++

C++ é uma linguagem de programação de propósito geral desenvolvida inicialmente por Bjarne Stroustrup. Stroustrup começou o seu trabalho na linguagem em 1979 e, em 1985, a primeira versão comercial de C++ foi lançada, popularizando a linguagem. De acordo com seu criador, a linguagem foi projetada para juntar as facilidades de Simula, a primeira linguagem

de programação com objetos e classes como conceitos centrais (CAPRETZ, 2003), à eficiência e flexibilidade de C.

Concorrência era um conceito importante desde o início do projeto da linguagem. Além de objetos e classes, outra facilidade de Simula que Stroustrup pretendia trazer à sua linguagem era o suporte à concorrência. Com isso em mente, os primeiros códigos escritos no que viria a se tornar C++ eram uma biblioteca que fornecia concorrência no estilo de Simula à nova linguagem. Os primeiros programas da linguagem eram simulações utilizando tal biblioteca: simulações de tráfego de rede e de layout de placas de circuito impresso.

Apesar disso, mecanismos de concorrência e sincronização só foram adicionados à biblioteca padrão da linguagem a partir da versão C++11. Os mecanismos da biblioteca padrão testados nesse trabalho foram *threads* para concorrência e *mutex* para sincronização.

Além disso, desde 1998 a API OpenMP está disponível para C++. OpenMP é uma API para expressar concorrência; foi desenvolvida em 1997 para Fortran, sendo portada para C e C++ no ano seguinte. Um dos grandes benefícios de OpenMP é a facilidade com que permite implementar concorrência: adicionando algumas diretivas a blocos existentes de códigos sequenciais, é possível transformá-los em programas concorrentes.

Por exemplo, suponha a existência do seguinte bloco de código que imprime “Hello World” na tela:

```
std::cout << "Hello World\n";
```

Caso deseje-se executar tal trecho de código quatro vezes de forma concorrente, basta adicionar a diretiva “#pragma omp parallel num_threads(4)” ao bloco:

```
#pragma omp parallel num_threads(4)
{
    std::cout << "Hello World\n";
}
```

Esse exemplo demonstra a facilidade de se programar de forma concorrente com OpenMP: com apenas uma diretiva é possível criar o número desejado de *threads*, executá-las de forma concorrente e aguardar o término das *threads* criadas para prosseguir com a execução do programa.

Devido a essa facilidade, optou-se por também avaliar o desempenho de OpenMP para C++: esse é o único mecanismo avaliado nesse trabalho que não faz parte da sintaxe da linguagem a que se refere nem pertence a alguma de suas bibliotecas padrão. Além das *threads* para concorrência, foram avaliados os *locks* que OpenMP oferece para sincronização.

Ressalta-se que, quando as *threads* de OpenMP forem utilizadas, não será utilizado o *mutex* da biblioteca padrão de C++; da mesma forma, os *locks* de OpenMP não serão utilizados

com as *threads* da biblioteca padrão de C++. Essa escolha se deve a preocupações com compatibilidade que fogem do escopo desse trabalho. Além disso, o único programa de teste desse trabalho que utiliza ao mesmo tempo mecanismos de concorrência e mecanismos de sincronização busca avaliar de fato apenas os mecanismos de sincronização. Portanto, testar diferentes mecanismos de concorrência com o mesmo mecanismo de sincronização não resultaria em dados de interesse.

2.3.2 Go

Go é uma linguagem de programação introduzida pelo Google em 2009. Ela tem o objetivo de ser uma linguagem simples, facilitando a escrita de programas, de compilação rápida e cujo código compilado é eficiente.

Ademais, Go também objetiva fornecer uma forma simples, eficiente e segura de escrever programas concorrentes (TU et al., 2019), sendo esse um de seus grandes pontos positivos como linguagem de programação. Primitivas para a execução de código concorrente fazem parte da definição de Go. O próprio nome da linguagem é uma palavra reservada que executa uma função de forma concorrente: ao preceder uma chamada de função com a primitiva *go*, inicia-se a execução da função em uma *Goroutine*.

Threads são gerenciadas pelo sistema operacional e possuem pilha de tamanho fixo. *Goroutines*, por outro lado, não possuem tamanho fixo para a sua pilha e são gerenciadas pelo *runtime* de Go; elas funcionam como *threads* eficientes e leves (FREITAS; MATOS; NOGUEIRA, 2017) que são escalonadas pelo *runtime* de Go para *threads* do sistema operacional. Mais de uma *Goroutine* pode ser escalonada para a mesma *thread* do sistema operacional (PRABHAKAR; KUMAR, 2011).

Como elas não possuem tamanho fixo de pilha, *Goroutines* são iniciadas com pilhas pequenas, tipicamente 2KB (DONOVAN; KERNIGHAN, 2015). Isso as faz muito mais leves que *threads* do sistema operacional, tornando sua criação muito mais rápida e menos custosa. Devido a isso, decidiu-se avaliar o desempenho de *Goroutines* para concorrência, assim como a estrutura *mutex* que a biblioteca padrão do Go oferece para sincronização.

2.3.3 Java

Java é uma das mais difundidas linguagens de programação da atualidade, sendo utilizada em uma variedade imensa de sistemas: desde aplicações para *desktops* e celulares até sistemas embarcados como reprodutores de Blu-ray.

Sua história teve início em um projeto da Sun Microsystems para desenvolver um dispositivo que controlasse todos os aparelhos eletrônicos de uma casa. Como o projeto falhou, a Sun passou a promover a linguagem que havia desenvolvido para o projeto, lançando sua primeira versão pública em 1995.

Java é uma linguagem orientada a objetos que tem como simplicidade uma de suas principais características, em especial ao se tratar do gerenciamento de memória. Outra característica importante de Java é a sua portabilidade: o mesmo código Java pode ser compilado apenas uma vez e, após isso, executado em qualquer *hardware* com suporte à JVM. Isso ocorre pois códigos Java são compilados para *bytecode* que são, posteriormente, interpretados pela JVM para rodar no sistema em questão.

Outra característica de Java é a sua atenção à concorrência: alguns mecanismos para concorrência e sincronização fazem parte da própria sintaxe da linguagem, enquanto diversos outros estão presentes em sua biblioteca padrão. Para o mecanismo de concorrência serão avaliadas as *threads* de Java, enquanto os mecanismos de sincronização são:

- Bloco *synchronized*: usando a palavra reservada *synchronized* é possível criar um bloco de código em exclusão mútua. Em Java todo objeto possui um *lock* associado a si (NIEMEYER; LEUCK, 2013); *synchronized* recebe um objeto e usa o *lock* associado a ele como *lock* do bloco de código. Portanto, em Java qualquer objeto pode funcionar, por meio do uso da palavra reservada *synchronized*, como um *mutex*. Nesse trabalho objetos da classe Object foram passados ao bloco *synchronized*;
- *Locks* injustos: não há garantia sobre a ordem em que as *threads* adquirem o *lock*;
- *Locks* justos: favorecem a *thread* que está esperando para adquirir o *lock* há mais tempo;
- Semáforos injustos: não há garantia sobre a ordem em que as *threads* adquirem um recurso;

- Semáforos justos: as *threads* adquirem recursos de acordo com a ordem com que os requisitaram (FIFO).

Ressalta-se que os *locks* avaliados são da classe ReentrantLock, uma vez que Java possui outras classes menos comuns que também implementam a interface *Lock*. O nome ReentrantLock tem origem no fato de que as *threads* podem acessar o *lock* mais de uma vez, como todos os demais *locks* e mecanismos de sincronização avaliados nesse trabalho.

2.3.4 Kotlin

Kotlin é uma linguagem de programação de propósito geral desenvolvida pela empresa tcheca JetBrains. Seu projeto começou em 2010, mas sua primeira versão oficial foi liberada apenas em 2016. Kotlin combina programação orientada a objetos e programação funcional e foca em ser uma linguagem segura, concisa e que fornece interoperabilidade com Java (BRESLAV, 2016).

Por ser interoperável com Java, qualquer biblioteca e código Java pode ser utilizado em Kotlin; da mesma forma, códigos em Kotlin também podem ser utilizados em programas Java. Além disso, Kotlin também compila para a JVM, podendo ser utilizada em qualquer sistema com suporte a Java.

Apesar da interoperabilidade com Java, Kotlin possui uma extensa biblioteca padrão própria, que inclui mecanismos de concorrência e sincronização. Para a concorrência, optou-se por avaliar as C*oroutines* fornecidas pela linguagem. Elas, assim como as *Goroutines*, podem ser vistas como *threads* leves (KOTLIN FOUNDATION, 2021): milhares de *Coroutines* podem ser criadas, executando sobre uma *pool* compartilhada de *threads*, com pouco impacto na performance do programa. Para que todas as *Coroutines* possam executar concorrentemente, é necessário que o tamanho da *pool* de *threads* seja no mínimo igual ao número de *Coroutines* que se deseja executar.

A *thread pool* padrão de Kotlin, entretanto, possui o mesmo número de *threads* que o número de núcleos da CPU. Portanto, como alguns dos testes a serem apresentados na subseção 3.3 utilizam mais tarefas concorrentes do que o número de núcleos da CPU, necessitou-se adicionar mais *threads* à *pool* para que todas as *Coroutines* executassem de forma concorrente

Para sincronização foram escolhidos *mutex*, semáforos e bloco *synchronized*. Os semáforos de Kotlin são justos (KOTLIN FOUNDATION, 2021), mantendo uma fila de requisições de recursos. Blocos *synchronized*, por sua vez, funcionam da mesma forma que os

blocos *synchronized* de Java; nas implementações desse trabalho, entretanto, utilizou-se objetos da classe Int como *lock*, já que Kotlin não possui a classe Object.

2.3.5 Scala

Scala é uma linguagem de programação que unifica programação orientada a objetos e programação funcional. Ela começou a ser desenvolvida em 2001 pela EPFL e em 2003 a sua primeira versão foi lançada. Programas em Scala se assemelham, em diversas formas, com programas em Java (ODERSKY et al., 2015). Além disso, Scala foi desenvolvida para ser interoperável com Java, assim como Kotlin.

O mecanismo de concorrência de Scala que será avaliado nesse trabalho é denominado *Future*. Um *Future* é um marcador de posição para um valor que ainda não existe, mas que será disponibilizado de forma concorrente e poderá ser utilizado no futuro.

Futures definem tarefas que serão executadas numa *thread pool*. Da mesma forma que Kotlin, para que todos os *Futures* executem concorrentemente é necessário utilizar uma *thread pool* com no mínimo o mesmo número de *Futures* a serem executados. A *pool* padrão de Scala, entretanto, fornece o mesmo número de *threads* que o número de núcleos da CPU. Logo, necessitou-se criar uma *thread pool* maior que a padrão.

Para sincronização, optou-se por avaliar o único mecanismo fornecido diretamente por Scala, os blocos *synchronized*. Além disso, como Scala possui interoperabilidade com Java, decidiu-se também avaliar como os mecanismos de sincronização de Java performam em Scala. Essa escolha se deve ao fato de que possuir mais de um mecanismo de sincronização traz mais flexibilidade à programação e, consequentemente, beneficia os seus usuários. Kotlin, apesar de também ser interoperável com Java, possui implementação própria dos mesmos mecanismos de sincronização de Java avaliados nesse trabalho: semáforos, blocos *synchronized* e *locks*. Portanto, para Kotlin optou-se por avaliar apenas os mecanismos fornecidos diretamente pela linguagem.

2.4 Trabalhos Relacionados

Essa subseção apresenta alguns trabalhos cujo tema é relacionado ao trabalho atual; além dos trabalhos, é apresentada uma pesquisa anual que coleta dados sobre programadores. Tais trabalhos contribuíram na escolha das linguagens de programação a serem analisadas e

comparadas. Por fim, argumenta-se sobre a importância do trabalho atual em comparação aos trabalhos relacionados.

2.4.1 Developer Survey

Stack Overflow é um famoso *site* de perguntas e respostas para programadores. Anualmente o site realiza uma pesquisa com seus usuários a fim de obter informações sobre os seus usuários programadores. A pesquisa faz perguntas sobre a geografia e demografia de seus respondentes, sua experiência com programação, nível de educação e etc. A pergunta de maior importância para o presente trabalho é a que questiona seus respondentes sobre quais linguagens de programação eles utilizam. Devido a essa pergunta, a pesquisa do *site* se relaciona ao trabalho atual ao listar as linguagens de programação mais populares dentre os usuários do *site*.

2.4.2 *A Comparative Study of Programming Models for Concurrency*

Silveira (2012) realiza, nesse trabalho, a avaliação e comparação de diferentes linguagens e modelos para programação concorrente, com ênfase em modelos de programação diferentes do tradicional modelo *multithreaded*. As linguagens e modelos analisados foram Chapel, Cilk, Erlang, Go, SCOOP (modelo para a linguagem Eiffel) e TBB (biblioteca para a linguagem C++); as métricas utilizadas foram tempo para programar, número de linhas do código-fonte, tempo de execução e *speedup* (quão mais rápido um algoritmo paralelo é em comparação ao seu semelhante sequencial). Por fim, Silveira recomenda a avaliação de OpenMP (C++) em trabalhos futuros.

2.4.3 *Concurrency in Go and Java: Performance Analysis*

Nesse trabalho Togashi e Klyuev (2014) realizam a comparação de Go e Java quanto à concorrência. As métricas utilizadas foram número de linhas do código-fonte, tamanho em bytes do código-fonte, tempo de compilação e, por fim, tempo de execução.

2.4.4 Avaliação de concorrência e de sincronização no Android

Vargas (2019) realiza, nesse trabalho, a avaliação de diferentes mecanismos para a programação concorrente de aplicações Android. Ademais, além de avaliar os mecanismos para

criação de tarefas concorrentes, tal trabalho também avaliou mecanismos de sincronização. Alguns dos mecanismos avaliados, como o framework HaMeR, o IntentService e o AsyncTask, são específicos para Android; outros, como as *Coroutines* de Kotlin e as *threads* de Java, são de uso geral. As métricas utilizadas foram tempo de execução, justiça, *throughput* (quanto o sistema produziu) e *speedup*.

2.5 Considerações finais

Esse capítulo apresentou conceitos importantes para o entendimento dos capítulos seguintes. Inicialmente foram apresentados conceitos básicos da concorrência: processos e *threads*. Tais conceitos são essenciais, visto que todos os mecanismos de concorrência avaliados nesse trabalho são ou *threads* propriamente ditas, ou dependem de *threads* e *thread pools*.

Além disso, foram apresentados problemas típicos da programação concorrente que são solucionados por meio de mecanismos de sincronização. Diversos mecanismos de sincronização também foram apresentados; entendê-los, assim como entender quando utilizar cada um deles, é essencial para compreender o motivo de alguns serem deixados de lado da avaliação de desempenho desse trabalho, como será exemplificado na subseção 3.2.

As linguagens estudadas e os seus mecanismos de interesse foram, em seguida, apresentados, de forma a facilitar a compreensão dos experimentos realizados e analisados por esse trabalho. Por fim, os trabalhos relacionados a esse trabalho e que influenciaram na escolha das linguagens estudadas foram brevemente apresentados.

3 METODOLOGIA

O presente capítulo inicia apresentando a metodologia utilizada para realizar esse trabalho e avaliar a performance das linguagens e de seus mecanismos. As suas subseções, por sua vez, minuciam alguns passos da metodologia, como as métricas e os programas-teste utilizados.

3.1 Metodologia para avaliação sistemática de performance

A metodologia de avaliação empregada nesse trabalho é a metodologia sistemática proposta por Jain (1991). Tal metodologia segue dez passos essenciais. Inicialmente apresentar-se-á um mapa geral dos dez passos. Ao longo desse capítulo os oito primeiros passos serão explicados detalhadamente, enquanto os dois últimos, referentes aos resultados coletados, serão apresentados no capítulo 4.

Os dez passos são:

1. **Definir o sistema e formular o objetivo do trabalho**: O objetivo desse trabalho é avaliar e comparar diferentes linguagens de programação de propósito geral em relação a seus mecanismos para concorrência e sincronização. O sistema desse trabalho é, portanto, o conjunto de linguagens estudadas: C++, Go, Java, Scala e Kotlin.
2. **Listar serviços**: Os serviços oferecidos pelo sistema são os mecanismos de concorrência e sincronização fornecidos pelas linguagens. Os mecanismos avaliados nesse trabalho estão listados na Tabela 1.
3. **Selecionar métricas**: Nesse passo selecionam-se os critérios para avaliação e comparação de desempenho; esses critérios são denominados métricas. As métricas escolhidas para esse trabalho são apresentadas na subseção 3.2.
4. **Listar parâmetros que podem afetar a performance**:
    a. CPU;
    b. Parâmetros específicos de cada programa-teste;
    c. Sistema operacional;
    d. Versão das linguagens de programação.
5. **Listar parâmetros a serem variados**: Nesse passo listam-se os parâmetros variados para garantir que as conclusões obtidas pela avaliação sejam genéricas. Dos parâmetros citados no passo anterior, considera-se que a versão das linguagens não possui significância grande e, portanto, utilizou-se apenas uma versão recente de cada

linguagem. Todos os demais parâmetros foram variados e serão melhor detalhados na subseção 3.3.

6. **Definir técnica de avaliação**: A técnica utilizada para a avaliação é a medição real da execução de programas que utilizam os mecanismos de concorrência e sincronização de interesse. Tais medições, então, são utilizadas para quantificar as métricas do passo 3, de forma que os diferentes mecanismos e linguagens possam ser avaliados e comparados entre si.
7. **Programas de teste**: Eles são essenciais para avaliar as linguagens, conforme explicado no passo anterior. Os programas implementados para esse trabalho estão descritos na subseção 3.3.
8. ***Design* de experimentos**: O *design* dos experimentos é descrito na subseção 3.4.
9. **Analisar os dados**: A análise é apresentada no capítulo 4.
10. **Apresentar os dados**: Os dados são apresentados no capítulo 4.

3.2 Métricas

Essa subseção busca descrever as quatro métricas de interesse para esse trabalho. Tais métricas são:

- *Speedup*: o *speedup* é uma medida do ganho de desempenho proporcionado pela paralelização de um algoritmo anteriormente sequencial (CARULLO, 2020). Segundo Sottile, Mattson e Rasmussen (2009), o *speedup* de um programa paralelo é dado pela equação abaixo.

$$S(n) = \frac{T_s}{T_n}$$, em que:

$S(n)$: é o *speedup* proporcionado pelo algoritmo paralelo ao utilizar n tarefas concorrentes;

T_s: é o tempo de execução do algoritmo sequencial;

T_p: é o tempo de execução do algoritmo paralelo ao utilizar n tarefas concorrentes.

- *Overhead*: no contexto empresarial, *overhead* é também conhecido como custo indireto (SNYDER; DAVENPORT, 1997). No contexto desse trabalho, *overhead* é o custo indireto causado pela utilização dos mecanismos de

concorrência e sincronização; logo, é dado pelo tempo extra que um algoritmo leva para executar devido à utilização de tais mecanismos.

- Justiça: essa métrica busca avaliar se todas as tarefas concorrentes recebem acesso exclusivo a recursos do sistema por uma porcentagem razoável de tempo (SOTTILE et al., 2009). Essa métrica é importante pois é indesejável que alguma tarefa nunca ou muito raramente receba recursos do sistema necessários para executar a sua função, enquanto outras tarefas recebem os recursos do sistema sempre ou muito frequentemente.

Ressalta-se que, devido às métricas escolhidas para avaliar as linguagens e seus mecanismos, determinados mecanismos de sincronização oferecidos pelas linguagens foram deixados de lado. Por exemplo, Java oferece barreiras como um mecanismo de sincronização. Entretanto, não faz sentido avaliar a justiça desse mecanismo, uma vez que o objetivo dele não é garantir a exclusividade de acesso de uma tarefa concorrente a um recurso; o seu objetivo é simplesmente fazer com que todas as tarefas parem em um determinado ponto e, quando todas chegarem nesse ponto, voltem a executar.

Outro exemplo são as variáveis de condição oferecidas por Go, Java e C++. Como os programas implementados nesse trabalho não necessitaram de sinalização para serem solucionados, optou-se por não as analisar, dando preferência aos mecanismos que esse trabalho considera mais importantes. Apesar disso, enfatiza-se que variáveis de condição são importantes para a solução de diversos problemas, como o problema do produtor-consumidor.

3.3 Parâmetros e programas de teste

A seguir apresentar-se-á os programas-teste utilizados nesse trabalho. A apresentação consiste em descrever os problemas, listar os seus parâmetros variados e explicitar quais métricas eles visam avaliar nos mecanismos de concorrência e sincronização. Além dos parâmetros das aplicações apresentados nas próximas subseções, também se variou o sistema operacional e o *hardware* durante os experimentos.

Os sistemas operacionais utilizados foram Windows 10 e Ubuntu 18.04 LTS; essa variação visa confirmar que as conclusões obtidas com esse trabalho são independentes de plataforma. Devido ao mesmo motivo, decidiu-se realizar os testes com diferentes CPUs:

- A máquina com Windows 10 possuía o processador Intel® Core™ i7-7700K de 4.20Ghz, com 4 núcleos físicos e 8 núcleos lógicos. Ela também será chamada de M1 a partir daqui;
- A máquina com Ubuntu 18.04 LTS possuía o processador Intel® Xeon® Silver 4116 de 2.10GHz, com 12 núcleos físicos e 24 núcleos lógicos. Ela também será chamada de M2 a partir daqui.

3.3.1 Multiplicação de matrizes

A multiplicação de matrizes é uma operação necessária a diversas áreas do conhecimento; com isso, torna-se uma operação indispensável para uma quantidade imensa de aplicações. O produto de uma matriz A_{MxP} por uma matriz B_{PxN} resulta numa matriz C_{MxN}; cada elemento c_{ij} de C é dado pela soma dos resultados parciais da multiplicação dos elementos da iésima linha de A pelos respectivos elementos da jésima coluna de B. A definição matemática dessa operação é:

$$c_{ij} = \sum_{k=1}^{P} a_{ik} b_{kj} \quad para\ i = 1, \dots, M\ e\ j = 1, \dots, N$$

Com base nessa equação, um simples algoritmo sequencial para a multiplicação de matrizes pode ser obtido com três laços aninhados. O primeiro laço percorre as linhas da matriz A, o segundo laço percorre as colunas da matriz B e o terceiro laço percorre os elementos da linha atual e da coluna atual, multiplicando-os e somando-os.

Esses três laços, entretanto, tornam o algoritmo muito custoso conforme o número de elementos da matriz cresce. Uma possível solução a esse custo alto é paralelizar o algoritmo, uma vez que cada elemento c_{ij} independentemente dos demais elementos c_{ij}. Portanto, por meio da comparação do desempenho da versão sequencial do algoritmo com o desempenho da versão concorrente, a única métrica que se busca avaliar com a multiplicação de matrizes é o *speedup* proporcionado pela concorrência.

Para tanto, a implementação da multiplicação de matrizes utilizada nesse trabalho assume que o número de tarefas concorrentes será uma potência de dois; também se assume que as matrizes terão dimensão NxN, com N sendo uma potência de dois divisível pelo número de tarefas. Assumir essas características permite implementar a multiplicação de matrizes de forma simples e direta; isso, por sua vez, permite dar ênfase à diferença de desempenho proporcionada pela concorrência, eliminando detalhes de implementação.

Além de dificultar a implementação, caso tais características não fossem assumidas as tarefas teriam cargas de trabalho diferentes. Com isso, o tempo total de execução da multiplicação de matrizes concorrente seria definido pela tarefa de maior carga, penalizando o desempenho da concorrência e afetando negativamente a métrica que se deseja avaliar com esse teste.

Com base nisso, o algoritmo concorrente utilizado nesse trabalho divide as linhas da matriz A igualmente entre as tarefas concorrentes. Cada tarefa, então, ainda utiliza três laços aninhados: o primeiro laço percorre as linhas da matriz A atribuídas à tarefa atual, o segundo laço percorre as colunas da matriz B e o terceiro laço percorre os elementos da linha atual e da coluna atual, multiplicando-os e somando-os. Dessa forma, o custo computacional da multiplicação das matrizes é dividido igualmente entre todas as tarefas concorrentes.

Os parâmetros variados para esse programa-teste são:

- Tamanho das matrizes quadradas: 256, 1024 e 2048;
- Número de tarefas concorrentes: 1 (algoritmo sequencial), 1 (algoritmo concorrente), 4 e 8.

Além disso, o algoritmo concorrente apresentado não necessita de sincronização para o seu funcionamento. Com isso, os mecanismos avaliados com esse programa são:

- C++: *Threads* de OpenMP;
- C++: *Threads* da biblioteca padrão;
- Go: *Goroutines*;
- Java: *Threads*;
- Kotlin: *Coroutines*;
- Scala: *Futures*.

3.3.2 Criação de tarefas concorrentes

Esse programa consiste em simplesmente executar um laço que cria N tarefas concorrentes que executam uma função vazia. Além disso, criou-se outro programa que executa um laço que chama uma função vazia N vezes de forma sequencial. Com base na comparação entre os laços dos dois programas, esse teste objetiva avaliar o *overhead* causado pela criação de tarefas concorrentes.

O único parâmetro da aplicação variado foi o número de tarefas criadas. Os valores utilizados foram 1.000, 10.000 e 100.000. Como esse programa não necessita de sincronização,

os mecanismos avaliados são apenas os de concorrência, assim como na multiplicação de matrizes:

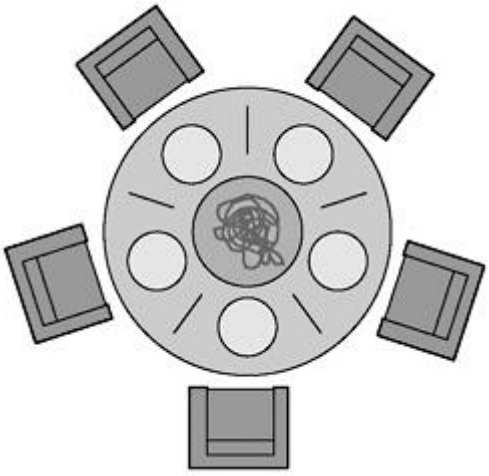
- C++: *Threads* de OpenMP;
- C++: *Threads* da biblioteca padrão;
- Go: *Goroutines*;
- Java: *Threads*;
- Kotlin: *Coroutines*;
- Scala: *Futures*.

3.3.3 Problema dos filósofos

Esse é um problema apresentado e resolvido por Dijkstra em 1965. No problema existem cinco filósofos sentados numa mesa redonda com um prato de macarrão na frente de cada um. Entre cada par adjacente de filósofos existe um garfo. Os filósofos passam o tempo todo num ciclo alternado de pensar e comer. Para comer, entretanto, um filósofo necessita pegar os dois garfos que estão diretamente ao seu lado.

Logo, quando algum dos filósofos fica com fome ele precisa aguardar até que ambos os seus garfos estejam disponíveis (ou seja, nenhum outro filósofo pode estar utilizando algum dos seus dois garfos). Na posse dos dois garfos aos seu lado, o filósofo pode finalmente se alimentar do macarrão; ao matar sua fome, ele libera os garfos para que os demais filósofos possam utilizá-los. O problema é exemplificado na Figura 3.

Figura 3.1 — Mesa do problema dos filósofos



(MAGEE; KRAMER, 2004)

Uma boa solução para o problema dos filósofos, além de seguir as regras postas na definição do problema, deve ser à prova de *deadlocks*. Para evitar *deadlocks*, a solução para o problema implementada nesse trabalho faz com que um filósofo comece sempre tentando pegar o garfo à sua esquerda; apenas caso consiga pegar o garfo à sua esquerda o filósofo passa a tentar obter o garfo à sua direita. Todos os demais filósofos, entretanto, fazem o contrário: começam sempre tentando pegar o garfo à sua direita quando desejam comer e, caso consigam pegar o garfo à sua direita, passam a tentar pegar o garfo à sua esquerda.

Com isso evita-se que todos os filósofos comecem pegando o seu respectivo garfo à direita ou à esquerda. Caso isso ocorresse, todos os filósofos teriam posse de um garfo e manteriam essa posse enquanto aguardam o seu outro garfo ser liberado. Isso, entretanto, nunca ocorreria, já que todos os filósofos estão aguardando a liberação do seu segundo garfo.

A implementação desse programa-teste, diferentemente da definição original do problema, permite a existência de um número N arbitrário de filósofos, que são representados por tarefas concorrentes. Além disso, os N garfos disputados pelos filósofos são representados pelos mecanismos de sincronização avaliados. A ação de comer, por sua vez, é representada pela multiplicação sequencial de duas matrizes de tamanho 64x64. Por fim, cada execução do programa dura cinco minutos.

Algo desejável para qualquer solução para o problema dos filósofos é que não ocorra *starvation*, ou seja, que nenhum filósofo passe fome. Portanto, esse problema é ideal para avaliar a justiça dos mecanismos de sincronização. Logo, apesar de esse ser o único programa que utiliza tanto os mecanismos de concorrência quanto os mecanismos de sincronização das linguagens, o que de fato deseja-se avaliar com o problema dos filósofos são os mecanismos de sincronização.

O único parâmetro do programa que foi variado foi o número de filósofos, ou seja, o número de tarefas concorrentes. Os valores utilizados foram 4, 8 e 50.

A única métrica que se deseja avaliar com esse programa de teste é a justiça dos mecanismos de avaliação. Com isso, os mecanismos avaliados com base no problema dos filósofos são:

- C++: *locks* (OpenMP);
- C++: *mutex* (biblioteca padrão);
- Go: *mutex*;
- Java: bloco *synchronized*;
- Java: *locks* injustos;

- Java: *locks* justos;
- Java: semáforos injustos;
- Java: semáforos justos;
- Kotlin: bloco *synchronized*;
- Kotlin: *mutex*;
- Kotlin: semáforos;
- Scala: bloco *synchronized*;
- Scala: *locks* injustos;
- Scala: *locks* justos;
- Scala: semáforos injustos;
- Scala: semáforos justos.

3.3.4 Uso de seções críticas

Esse programa passa N vezes por um laço que chama uma função que simplesmente entra numa seção crítica e a abandona logo em seguida. Além disso, criou-se outro programa que executa um laço que chama uma função vazia N vezes de forma sequencial. Com base na comparação entre os laços dos dois programas, esse teste objetiva avaliar o *overhead* causado por se entrar e sair de uma seção crítica.

O único parâmetro da aplicação variado foi o número de seções críticas das quais se entra e sai. Os valores utilizados foram 1.000, 10.000 e 100.000. Como esse programa não cria tarefas concorrentes, os mecanismos avaliados são apenas os de sincronização:

- C++: *locks* (OpenMP);
- C++: *mutex* (biblioteca padrão);
- Go: *mutex*;
- Java: bloco *synchronized*;
- Java: *locks* injustos;
- Java: *locks* justos;
- Java: semáforos injustos;
- Java: semáforos justos;
- Kotlin: bloco *synchronized*;
- Kotlin: *mutex*;
- Kotlin: semáforos;

- Scala: bloco *synchronized*;
- Scala: *locks* injustos;
- Scala: *locks* justos;
- Scala: semáforos injustos;
- Scala: semáforos justos.

3.4 Design de experimentos

Para a multiplicação de matrizes, um total de 72 experimentos foram planejados: há 3 tamanhos diferentes de matrizes e 3 números de tarefas concorrentes diferentes. Além disso, há a versão sequencial do algoritmo, que também é executada com os 3 tamanhos diferentes de matrizes. Esses 12 experimentos, por sua vez, foram executados com 6 mecanismos de concorrência diferentes. Com isso, chega-se nas 72 execuções.

Tanto o programa de criação de tarefas concorrentes quanto o de uso de seções críticas são comparados com um laço que chama uma função vazia N vezes, com N variando entre 1.000, 10.000 e 100.000. Como os mesmos valores de N são utilizados em ambos os casos, apenas 3 experimentos com o laço que chama uma função vazia são necessários por linguagem para realizar a comparação com os programas-teste. Isso resulta, então, em 15 experimentos.

Para o programa de criação de tarefas concorrentes, por sua vez, foram planejados 18 experimentos: há 3 valores para o número de tarefas criadas que são executados com os 6 diferentes mecanismos de concorrência.

O programa de uso de seções críticas possui 3 diferentes valores de seções críticas utilizadas. Isso, multiplicado pelos 16 mecanismos de sincronização utilizados, resulta em 48 experimentos.

Por fim, o problema dos filósofos possui 3 valores diferentes para filósofos que também são testados com todos os 16 mecanismos de sincronização de interesse para esse trabalho. Com isso, obtém-se mais 48 experimentos.

Somando esses valores se chega no número final de 201 experimentos. Para se obter resultados estatisticamente confiáveis, cada experimento foi executado 30 vezes. Os 201 experimentos executados 30 vezes resultam em 6.030 execuções por máquina; como foram utilizadas duas máquinas, o número total de execuções dos experimentos foi 12.060.

4 EXPERIMENTOS E ANÁLISE DE DADOS

Esse capítulo apresentará e analisará os resultados obtidos a partir dos experimentos descritos no capítulo 3. Ressalta-se que cada experimento foi executado trinta vezes, a fim de se obter resultados estatisticamente confiáveis. Para cada experimento, utilizou-se a mediana das trinta execuções para realizar as análises desse capítulo. Optou-se por utilizar a mediana pois ela é mais resistente a *outliers* se comparada à média (JAIN, 1991).

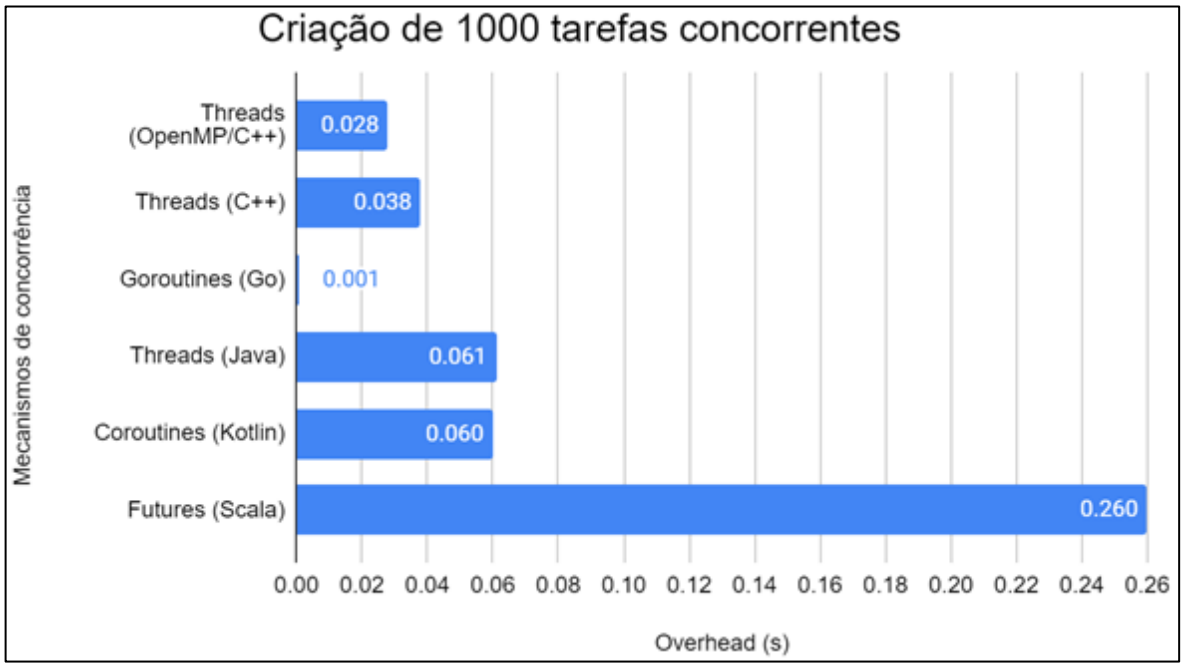
4.1 Criação de tarefas concorrentes

Para a análise desse programa se subtraiu o tempo de execução do laço que chama uma função vazia do tempo de execução do laço que cria tarefas concorrentes que chamam uma função vazia. Com isso obteve-se o tempo adicional que o programa levou para ser executado devido à criação das tarefas concorrentes. Idealmente o valor obtido seria zero, indicando que não há *overhead* devido à criação das tarefas concorrentes.

4.1.1 Criação de tarefas concorrentes em M1

O primeiro experimento realizado utiliza um laço de 1.000 iterações. Os resultados obtidos são apresentados na Figura 4. Nela é possível notar que *Goroutines* foi disparado o mecanismo de concorrência com menos *overhead*. Tal mecanismo teve apenas um milissegundo de *overhead*, um valor basicamente negligenciável. Também se percebe que os *Futures* de Scala foram o mecanismo com pior *overhead*. Apesar disso, em termos absolutos nenhum mecanismo sofreu muito com *overhead*, uma vez que todos tiveram *overhead* abaixo de um segundo.

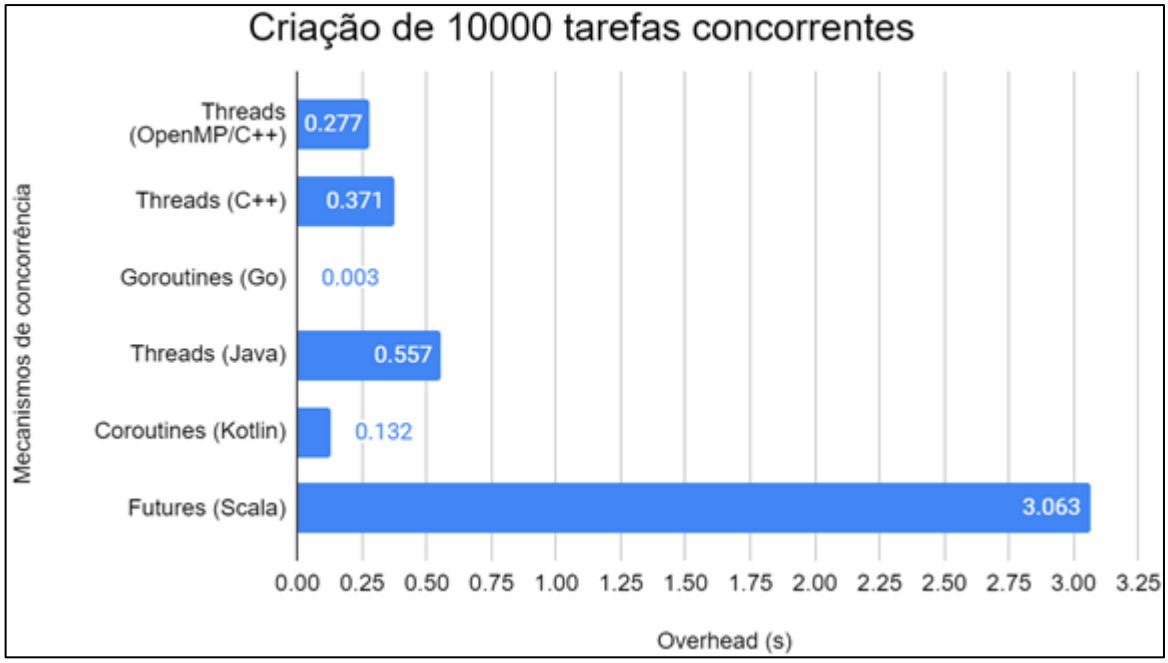
Figura 4.1 — Criação de 1.000 tarefas concorrentes na M1



Para o segundo experimento foi utilizado um laço de 10.000 iterações; os dados obtidos para ele estão na Figura 5. Mais uma vez as *Goroutines* foram o mecanismo com menor *overhead*, com um tempo de execução bem menor que os demais. Apesar disso, nota-se que as *Coroutines* de Kotlin escalaram muito bem com o aumento do número de tarefas concorrentes, superando ambos os mecanismos de C++ e ficando com o segundo melhor *overhead*.

Com um *overhead* de mais de três segundos, os *Futures* de Scala se mantiveram como o mecanismo com o maior *overhead* por uma grande margem.

Figura 4.2 — Criação de 10.000 tarefas concorrentes na M1

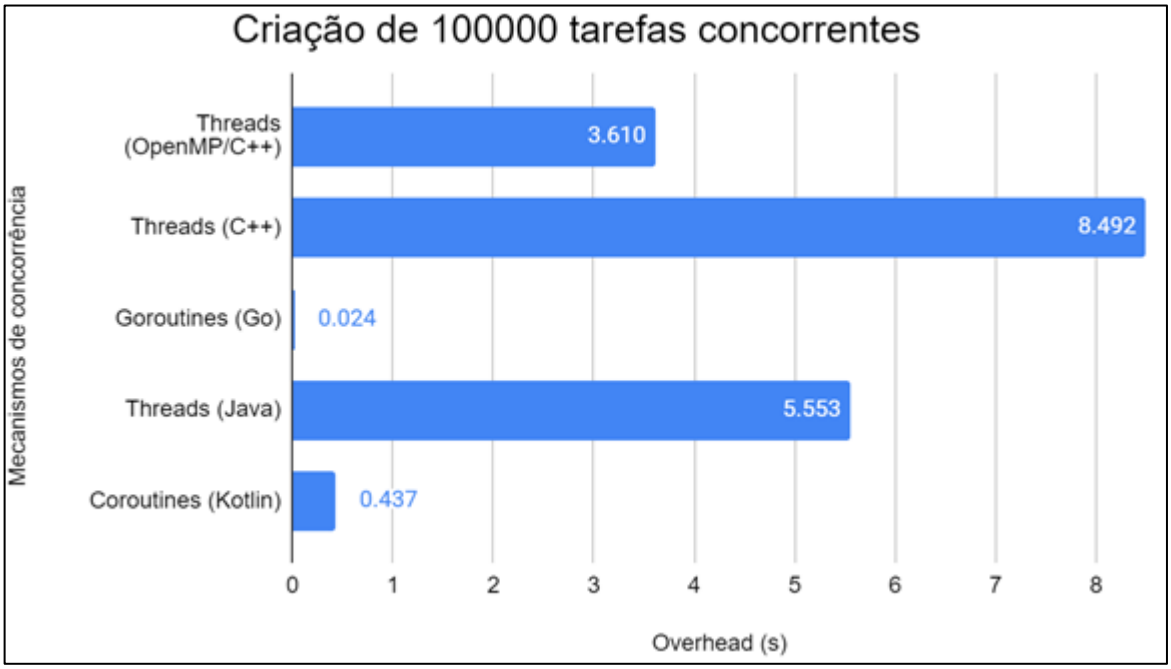


O último experimento utilizou um laço de 100.000 iterações cujos resultados estão representados na Figura 6. Mais uma vez as *Goroutines* foram melhores que os demais mecanismos de concorrência por uma grande margem, demonstrando ser o mecanismo mais escalável. Apesar disso, as *Coroutines* de Kotlin também sofreram pouco prejuízo no tempo de execução, mantendo com facilidade o seu segundo lugar como mecanismo com menos *overhead*.

Nesse experimento as *threads* da biblioteca padrão de C++ tiveram uma grande penalidade temporal e o seu desempenho em relação às *threads* de OpenMP piorou bastante: enquanto nos dois experimentos anteriores o seu *overhead* era apenas cerca de 33% maior, nesse experimento passou a ter cerca de 135% a mais de *overhead*. Além disso, pela primeira vez o seu desempenho foi pior que as *threads* de Java.

Os *Futures* de Scala foram, mais uma vez, o mecanismo de concorrência com maior *overhead*: 277 segundos. Dessa vez, entretanto, o *overhead* foi tão maior que os demais que se optou por não o apresentar no gráfico.

Figura 4.3 — Criação de 100.000 tarefas concorrentes na M1



Com isso, para a máquina com Intel® Core™ i7-7700K e Windows 10 o mecanismo com menor *overhead* foram as *Goroutines* de Go em todos os experimentos, enquanto os *Futures* de Scala tiveram o pior desempenho em todos os experimentos.

As *threads* de OpenMP e da biblioteca padrão de C++ tiveram tempos de execução alguns milissegundos abaixo das *Coroutines* de Kotlin no teste com mil tarefas concorrentes. As *threads* de Java, por sua vez, tiveram basicamente o mesmo desempenho que as *Coroutines*

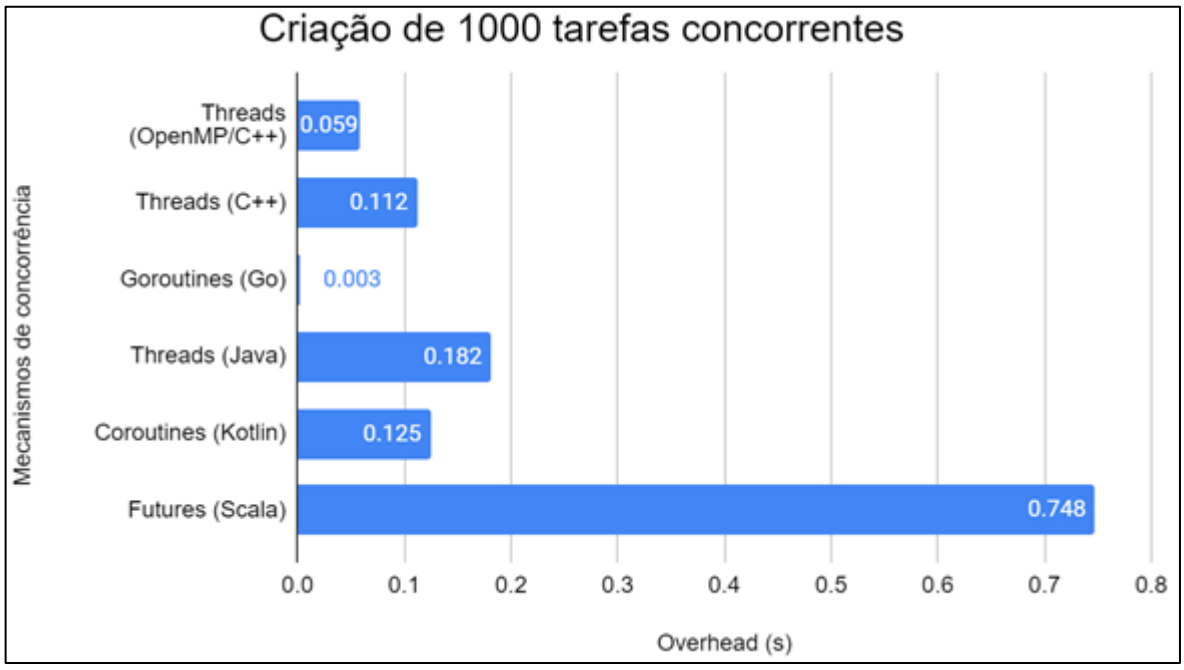
de Kotlin no teste com mil tarefas. Entretanto, nos testes seguintes as *Coroutines* não sofreram penalidades grandes em seu *overhead*, enquanto os três mecanismos com *threads* sim. Dessa forma, as *Coroutines* demonstraram ser muito mais escaláveis que esses três mecanismos e, portanto, considera-se que elas são o segundo melhor mecanismo de concorrência em relação ao *overhead*.

Numa comparação direta dos dois mecanismos de C++, as *threads* de OpenMP se mostraram preferíveis às *threads* da biblioteca padrão. Elas, além de terem um *overhead* menor em todos os experimentos, escalaram melhor.

4.1.2 Criação de tarefas concorrentes em M2

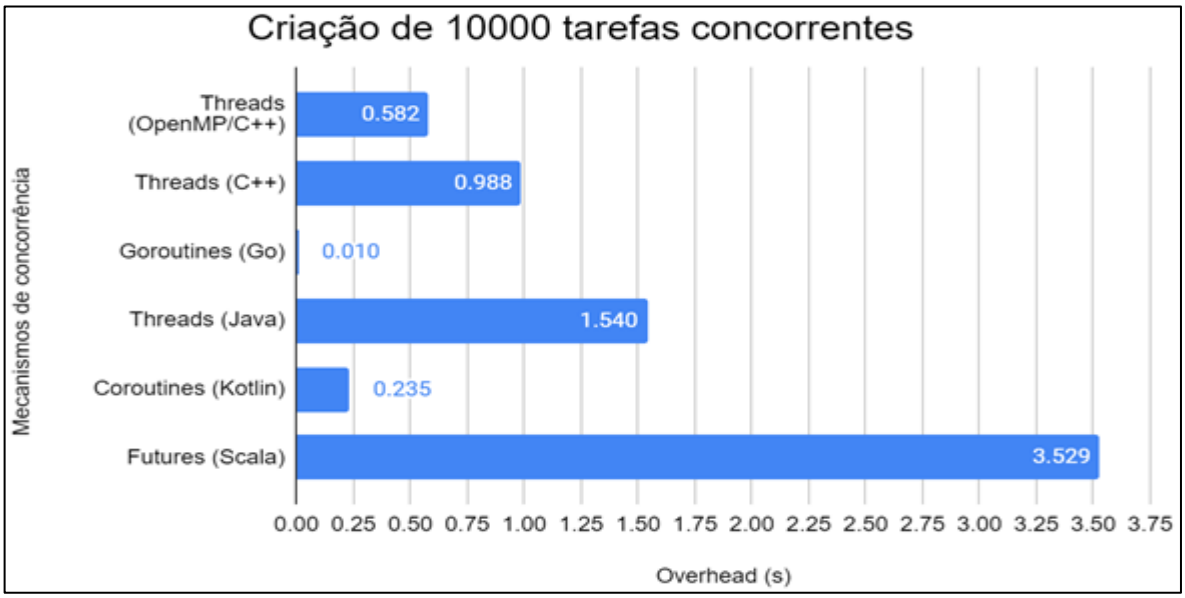
A Figura 7 apresenta os resultados da criação de 1.000 tarefas concorrentes na máquina com Ubuntu 18.04 LTS. Os resultados foram semelhantes aos da máquina com Windows 10: *Goroutines* foram o melhor mecanismo e *Futures* foram o pior. Além disso, novamente todos os mecanismos tiveram *overhead* de menos de um segundo.

Figura 4.4 — Criação de 1.000 tarefas concorrentes na M2



No experimento com 10.000 tarefas concorrentes, cujos resultados estão na Figura 8, as *Goroutines* se mantiveram como o mecanismo com menor *overhead*. Assim como aconteceu na máquina com Windows 10, nesse experimento as *Coroutines* superaram ambos os mecanismos de C++, mostrando serem mais escaláveis. *Futures*, por sua vez, continuaram sendo o mecanismo de concorrência com maior *overhead*.

Figura 4.5 — Criação de 10.000 tarefas concorrentes na M2

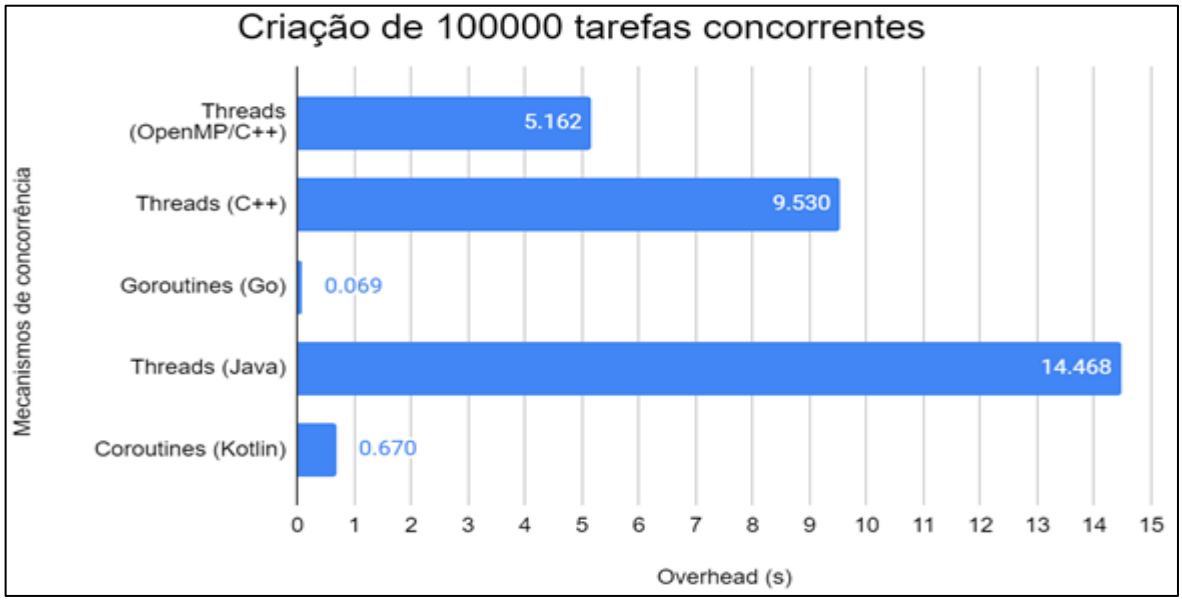


Na Figura 9 estão os resultados do experimento com 100.000 tarefas concorrentes. Novamente as *Goroutines* continuaram com o melhor *overhead*, enquanto as *Coroutines* continuaram com o segundo melhor.

Os *Futures* de Scala, por sua vez, foram mais uma vez o mecanismo de pior desempenho por uma grande margem. A mediana de seu *overhead* foi de cerca de 494 segundos. Devido a isso optou-se por não a apresentar no gráfico, já que é um valor muito maior que os demais.

Por fim, diferentemente do que aconteceu em M1, as *threads* da biblioteca padrão de C++ não tiveram desempenho muito pior se comparado aos testes com 1.000 e 10.000 tarefas. Nessa máquina elas se mantiveram com *overhead* inferior ao de Java e o percentual de diferença às *threads* de OpenMP não sofreu grandes variações.

Figura 4.6 — Criação de 100.000 tarefas concorrentes na M2



Com resultados muito similares aos da máquina com Windows 10, o mesmo vale para essa máquina: *Goroutines* foram o melhor mecanismo e, apesar de terem apenas o quarto melhor *overhead* no experimento com 1.000 tarefas, ao longo dos demais experimentos as *Coroutines* demonstraram ser o segundo melhor mecanismo em termos de *overhead*.

Por fim, se comparados apenas os mecanismos de C++, mais uma vez as *threads* de OpenMP apresentaram *overhead* menor que as *threads* da biblioteca padrão.

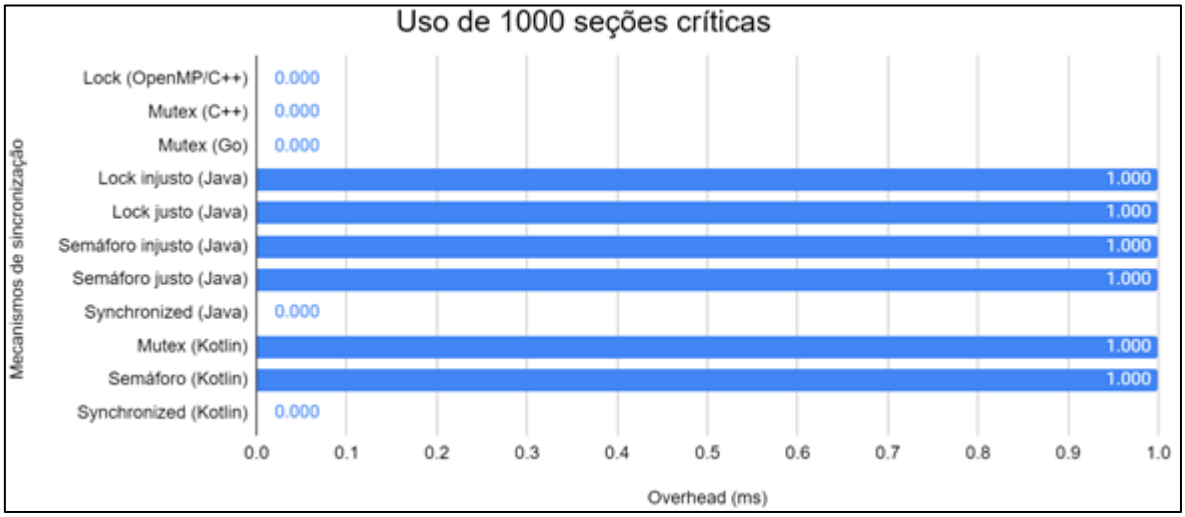
4.2 Uso de seções críticas

A fim de se analisar os dados obtidos com esse programa, o tempo de execução do laço que chama uma função vazia é subtraído do tempo de execução do laço que chama uma função que entra e sai de seções críticas definidas pelos mecanismos de sincronização; com isso o *overhead* devido à utilização de seções críticas é obtido. Assim como para o teste de criação de tarefas concorrentes, idealmente esse tempo é zero.

4.2.1 Uso de seções críticas em M1

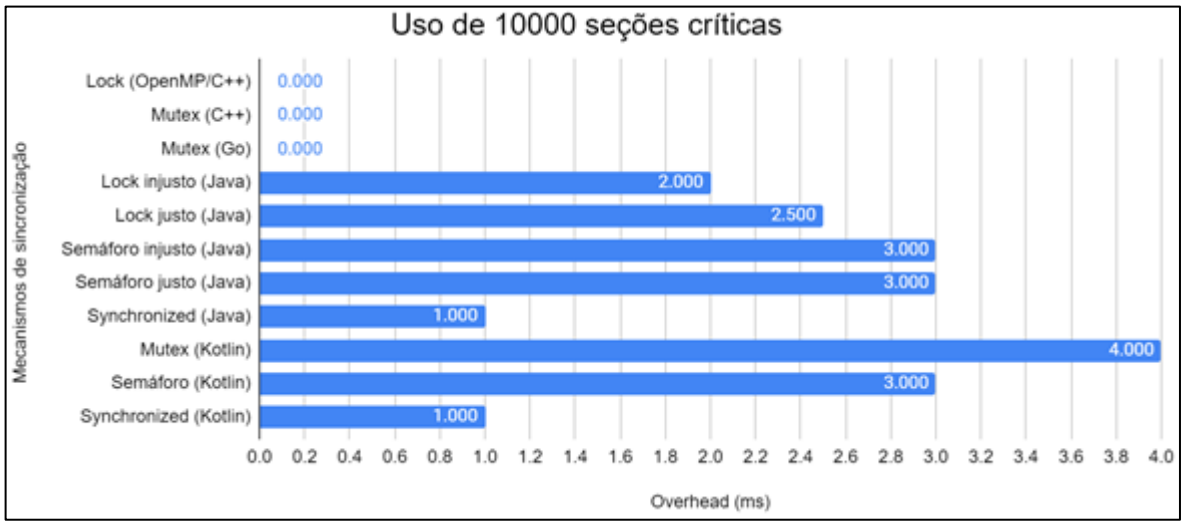
O primeiro experimento consistiu em usar 1.000 seções críticas. Tal valor se mostrou extremamente pequeno, uma vez que a mediana de todos os mecanismos, exceto os de Scala, não ultrapassou um milissegundo. Isso é demonstrado na Figura 10. A mediana do *overhead* de todos os cinco mecanismos de sincronização de Scala (*lock* inujsto, *lock* justo, semáforo injusto, semáforo justo e bloco *synchronized*) foi a mesma: 87 milissegundos. Como esse valor foi muito acima dos demais, optou-se por não representar os mecanismos de Scala na Figura 10.

Figura 4.7 — Uso de 1.000 seções críticas na M1



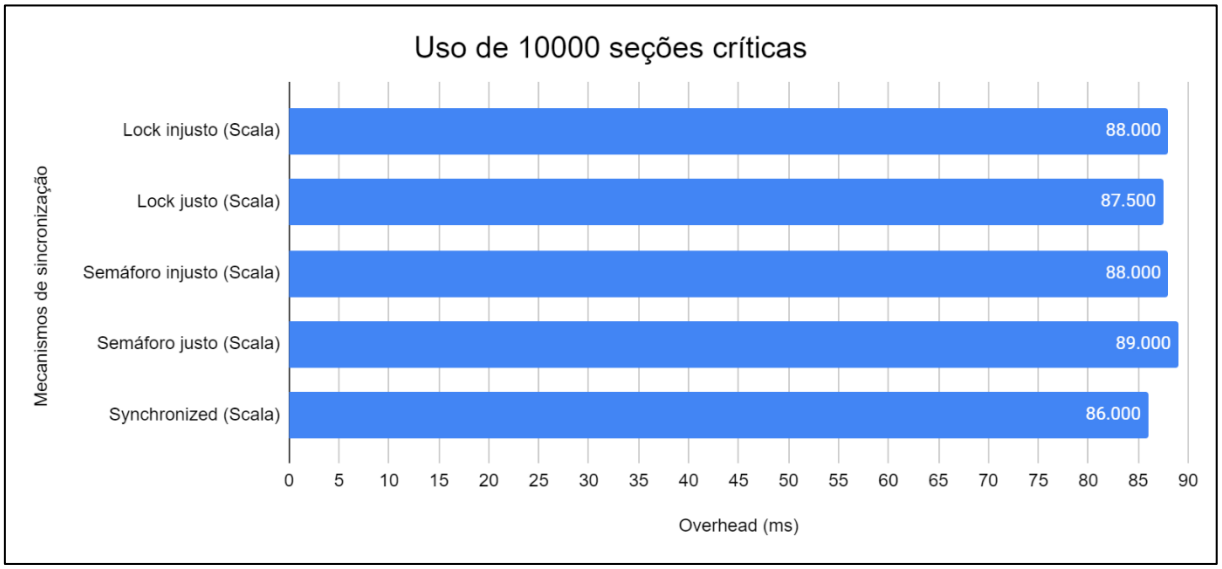
Para o segundo experimento, 10.000 seções críticas foram utilizadas; os resultados obtidos estão na Figura 11. Ambos os mecanismos de C++ e o *mutex* de Go permaneceram com *overhead* mediano de zero milissegundos, obtendo os menores *overhead*s. Apesar disso, observa-se que mais uma vez todos os mecanismos, exceto os de Scala, tiveram *overhead* ínfimo, não ultrapassando os quatro milissegundos.

Figura 4.8 — Uso de 10.000 seções críticas na M1



A Figura 12 apresenta os valores de *overhead* obtidos para os mecanismos de sincronização de Scala. Para esse experimento os mecanismos não obtiveram os mesmos resultados, mas eles se mantêm com desempenho extremamente próximo ao experimento com apenas mil seções críticas, em torno dos 87 milissegundos.

Figura 4.9 — Uso de 10.000 seções críticas na M1 (Scala)

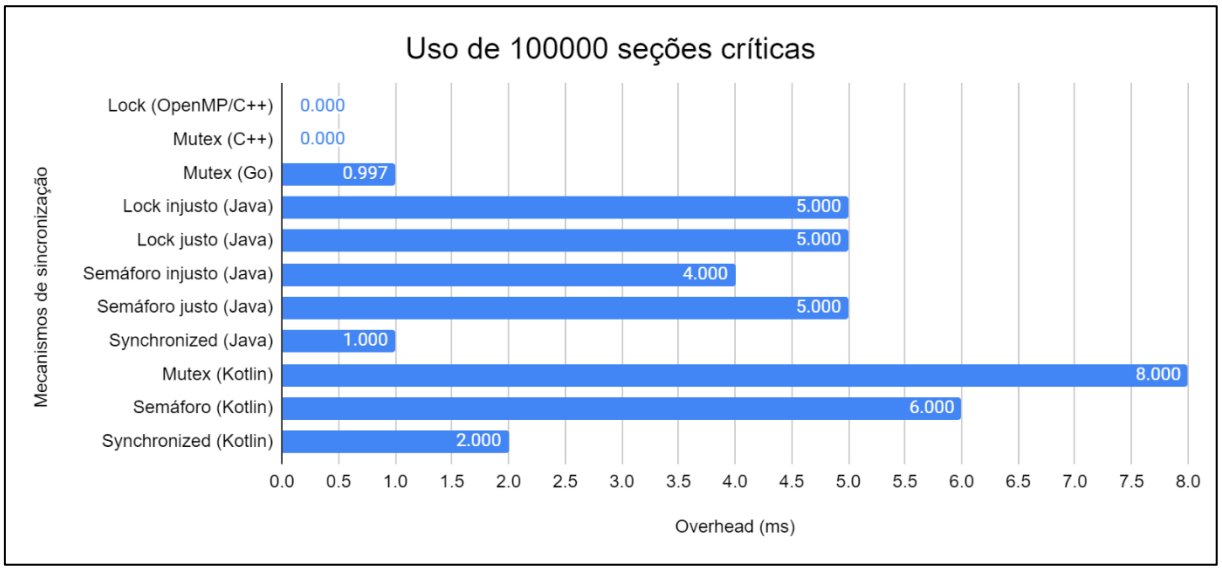


O último experimento consistiu em entrar e sair de 100.000 seções críticas; os resultados obtidos estão representados na Figura 13. Mais uma vez a mediana do *overhead* dos

mecanismos de C++ foi zero milissegundos; esses foram, portanto, os melhores mecanismos de sincronização na máquina com Windows 10. Apesar disso, ressalta-se que o *mutex* de Go ficou a menos de um milissegundo de também ter mediana zero, enquanto os demais mecanismos, excetuando-se os de Scala, tiveram *overhead* de no máximo oito milissegundos.

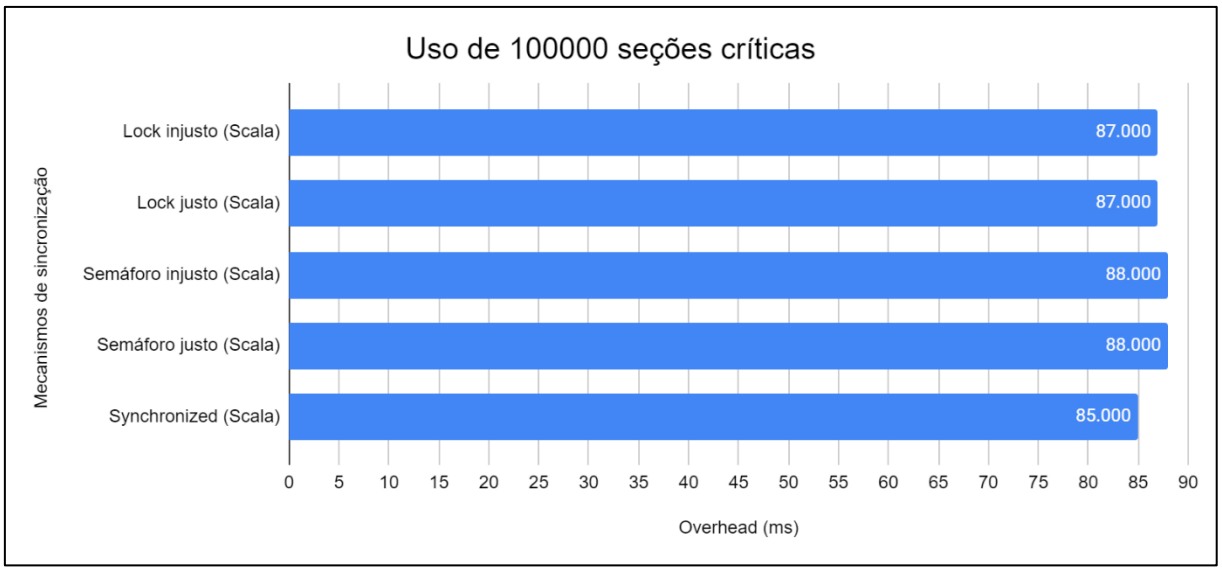
Com isso, conclui-se que, apesar de C++ ser a linguagem com menos *overhead*, nenhum mecanismo das demais linguagens teve *overhead* muito significativo. Também é perceptível que, apesar de as diferenças entre os *overhead*s serem ínfimas, o bloco *synchronized* foi o mecanismo de melhor desempenho em todas as linguagens que o possuem.

Figura 4.10 — Uso de 100.000 seções críticas na M1



Na Figura 14 estão os dados obtidos com os mecanismos de Scala. Em todos os experimentos os valores foram muito parecidos, cerca de 87 milissegundos, sendo disparado o mecanismo com pior desempenho.

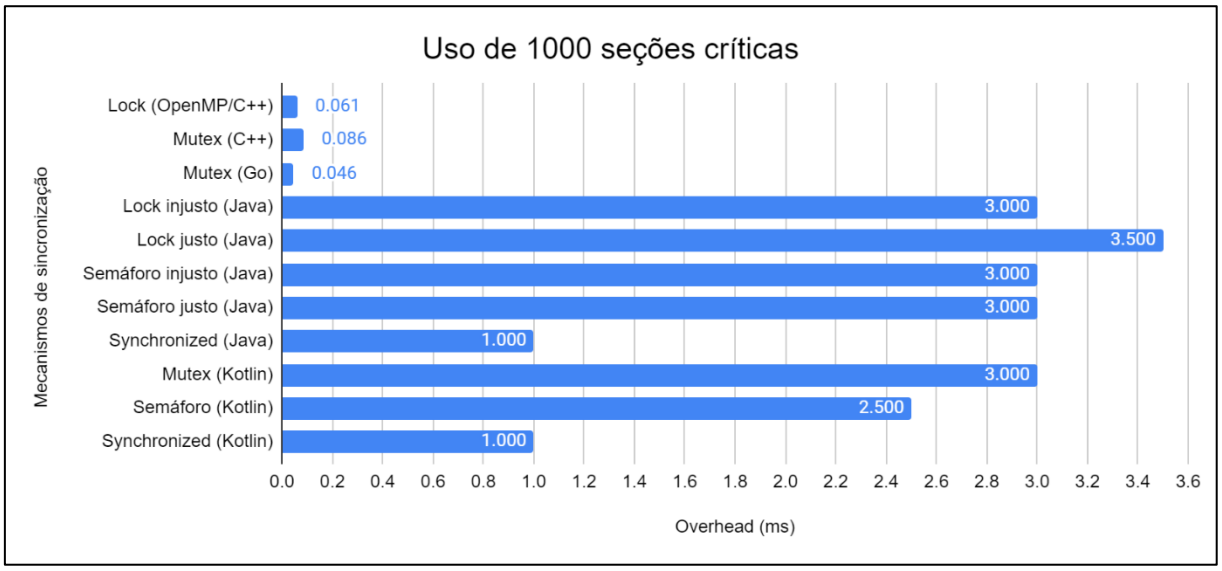
Figura 4.11 — Uso de 100.000 seções críticas na M1 (Scala)



4.2.2 Uso de seções críticas em M2

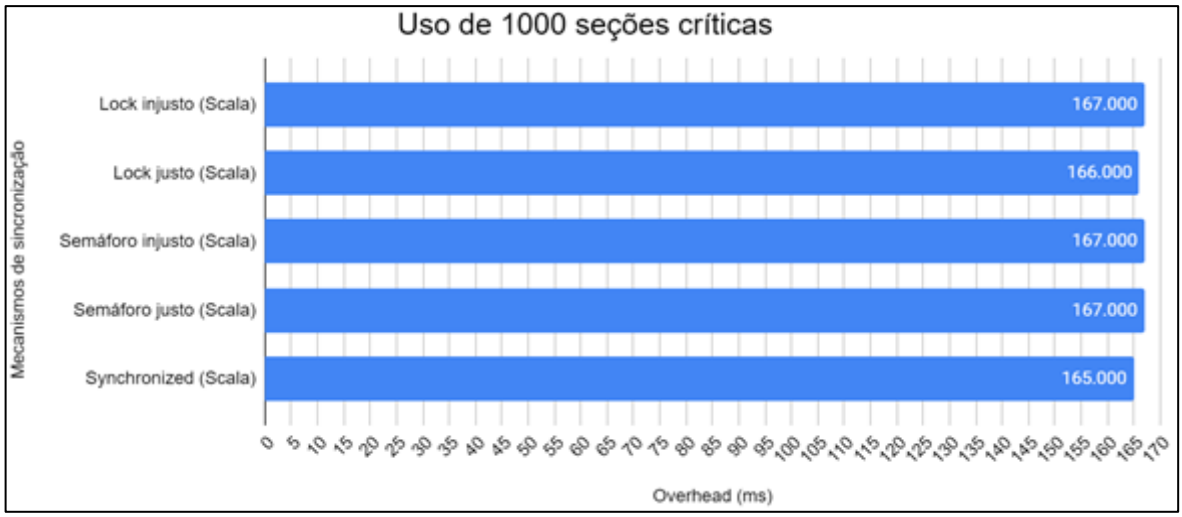
No experimento com 1.000 seções críticas na máquina com Ubuntu 18.04 LTS obteve-se valores baixos de *overhead* para todos os mecanismos, exceto os de Scala. Apesar de nessa máquina nenhum mecanismo ter tido *overhead* mediano de zero milissegundos, o *lock* e *mutex* de C++ tiveram *overhead* bem abaixo de um milissegundo, assim como o *mutex* de Go, que teve o melhor desempenho. Os dados de todos os mecanismos, menos os de Scala, estão representados na Figura 15.

Figura 4.12 — Uso de 1.000 seções críticas na M2



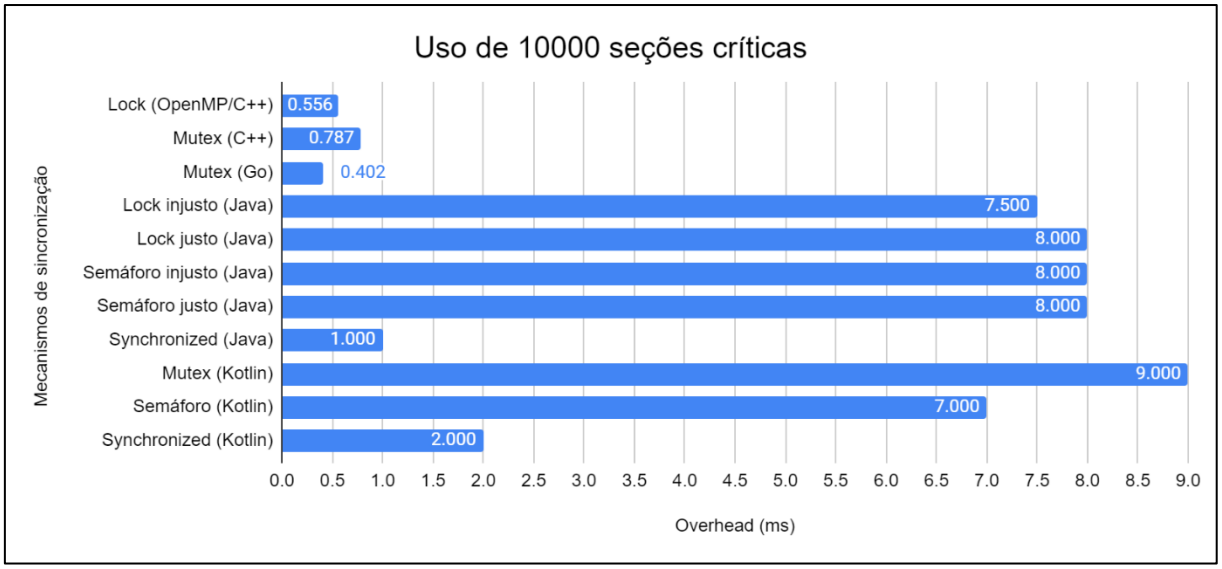
A Figura 16 apresenta os resultados obtidos pelos mecanismos de sincronização de Scala no experimento com mil seções críticas. Todos os mecanismos tiveram *overhead* extremamente próximos entre si, mas muito maior que o *overhead* dos mecanismos de sincronização das demais linguagens.

Figura 4.13 — Uso de 1.000 seções críticas na M2 (Scala)



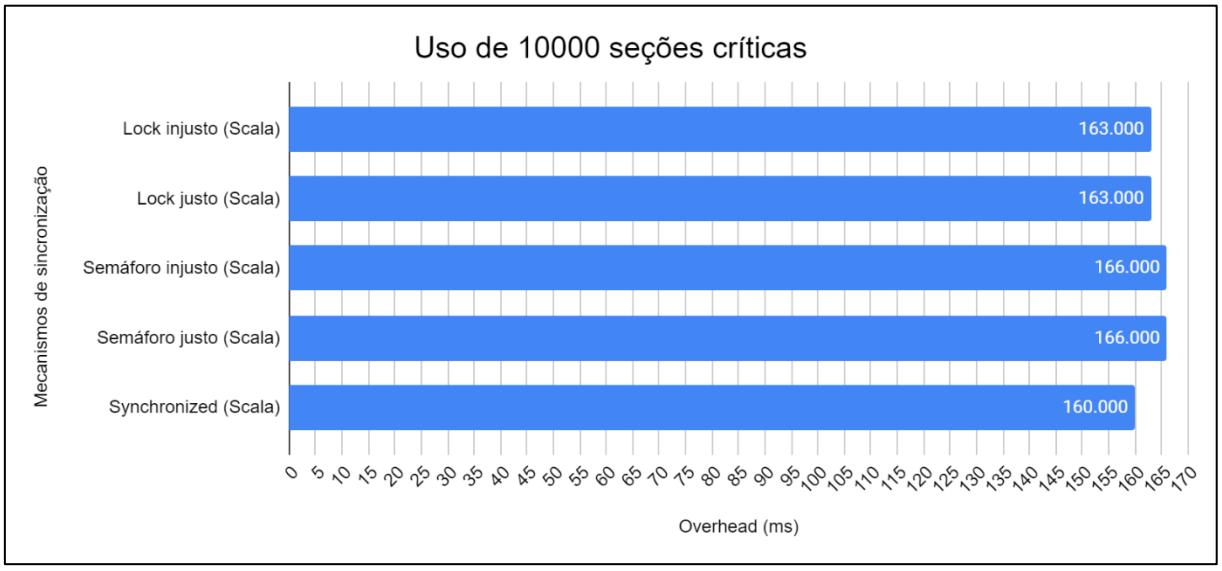
Na Figura 17 estão os resultados obtidos para o experimento com 10.000 seções críticas, exceto os de Scala. Com ele percebe-se que o *mutex* de Go segue tendo o melhor desempenho, seguido pelos dois mecanismos de C++. Mais uma vez esses três mecanismos tiveram *overhead* abaixo de um milissegundo. Nesse experimento a diferença entre o bloco *synchronized* das linguagens para os seus demais mecanismos aumentou em relação ao experimento com mil seções críticas, mostrando que os blocos *synchronized* possuem o menor *overhead* nas linguagens que possuem tal construção.

Figura 4.14 — Uso de 10.000 seções críticas na M2



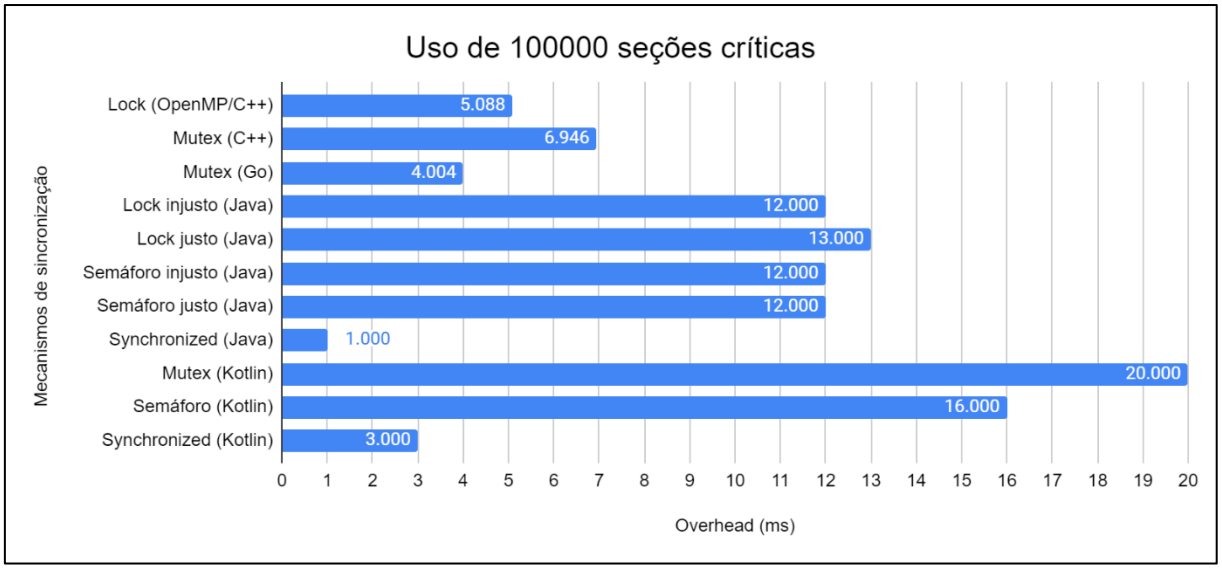
A Figura 18 mostra o *overhead* dos mecanismos de Scala para o experimento com 10.000 seções críticas. Os resultados foram extremamente próximos aos do experimento com 1.000 seções críticas, mas um pouco mais baixos.

Figura 4.15 — Uso de 10.000 seções críticas na M2 (Scala)



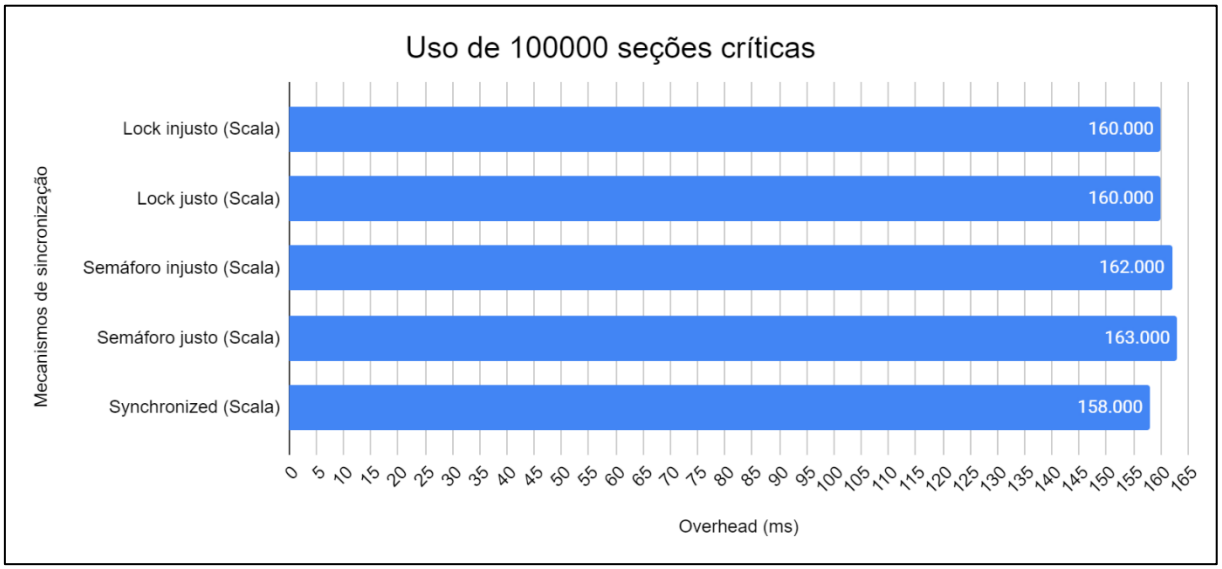
No último experimento foram utilizadas 100.000 seções críticas. Diferentemente do que ocorreu na máquina com Windows 10, nesse experimento o bloco *synchronized* de Java e Kotlin, respectivamente, tiveram o melhor desempenho dentre todos os mecanismos de sincronização. Com isso, conclui-se que, apesar de o *mutex* de Go ser o mecanismo de sincronização com menor *overhead* para o uso de algumas dezenas de milhares de seções críticas, o bloco *synchronized* de Java é o mecanismo de sincronização com *overhead* mais escalável, seguido do bloco *synchronized* de Kotlin. Isso é demonstrado na Figura 19 abaixo.

Figura 4.16 — Uso de 100.000 seções críticas na M2



A Figura 20, por sua vez, apresenta os *overhead*s dos mecanismos de Scala para o experimento em questão. Mais uma vez foram os mecanismos com os piores resultados, mas novamente os *overhead*s sofreram uma pequena queda em relação ao experimento anterior.

Figura 4.17 — Uso de 100.000 seções críticas na M2 (Scala)



4.3 Multiplicação de matrizes

Como discutido no capítulo 3, a métrica de interesse para esse programa é o *speedup*, que é calculado por meio da razão entre o tempo de execução do algoritmo sequencial e o tempo de execução do algoritmo paralelo utilizando N tarefas concorrentes. O valor ideal de *speedup* proporcionado pelo paralelismo é N.

Considera-se que, conforme maior o tamanho das matrizes, mais importante é o experimento para a análise. Isso se deve ao fato de que o principal objetivo de paralelizar um algoritmo é diminuir o seu tempo de execução. Logo, quanto maior o tempo de execução do algoritmo sequencial, mais importante se torna ter *speedup*s altos na versão paralela do algoritmo, a fim de diminuir o seu tempo de execução.

Reitera-se que, diferentemente dos trabalhos relacionados, o presente trabalho não se propõe a analisar os tempos de execução utilizados para calcular o *speedup*. Isso é explicado pelo fato de o foco desse trabalho ser analisar métricas relativas à concorrência; como o tempo de execução é dependente do desempenho sequencial das linguagens, optou-se por não o analisar. Apesar disso, as medianas dos tempos de execução da multiplicação de matrizes são apresentadas no Anexo B.

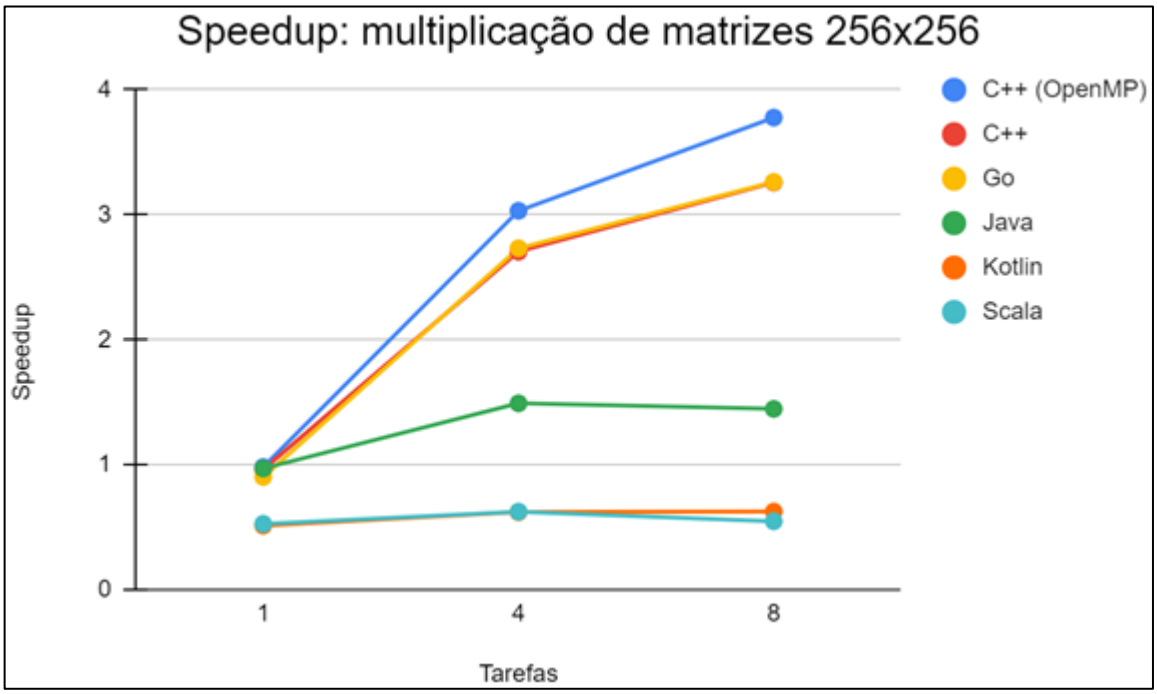
4.3.1 Multiplicação de matrizes em M1

O primeiro experimento consistiu em multiplicar duas matrizes quadradas de tamanho 256. Como o tempo de execução do algoritmo sequencial é extremamente baixo para matrizes sequenciais desse tamanho, nenhum mecanismo teve *speedup* surpreendente.

As *threads* de OpenMP tiveram o melhor desempenho, seguidas das *threads* da biblioteca padrão de C++ e as *Goroutines* de Go, que tiveram praticamente o mesmo *speedup* com os três números de tarefas concorrentes. Em quarto lugar aparecem as *threads* de Java, com *speedup* próximo a 1,5 ao utilizar quatro e oito tarefas concorrentes.

Por fim, tanto as *Coroutines* de Kotlin quanto os *Futures* de Scala tiveram *speedup* menor que um; com isso, elas foram as únicas linguagens cujo desempenho do algoritmo sequencial foi melhor do que o algoritmo paralelo para matrizes desse tamanho. Essas informações podem ser visualizadas na Figura 21 da página seguinte.

Figura 4.18 — Multiplicação de matrizes 256x256 na M1

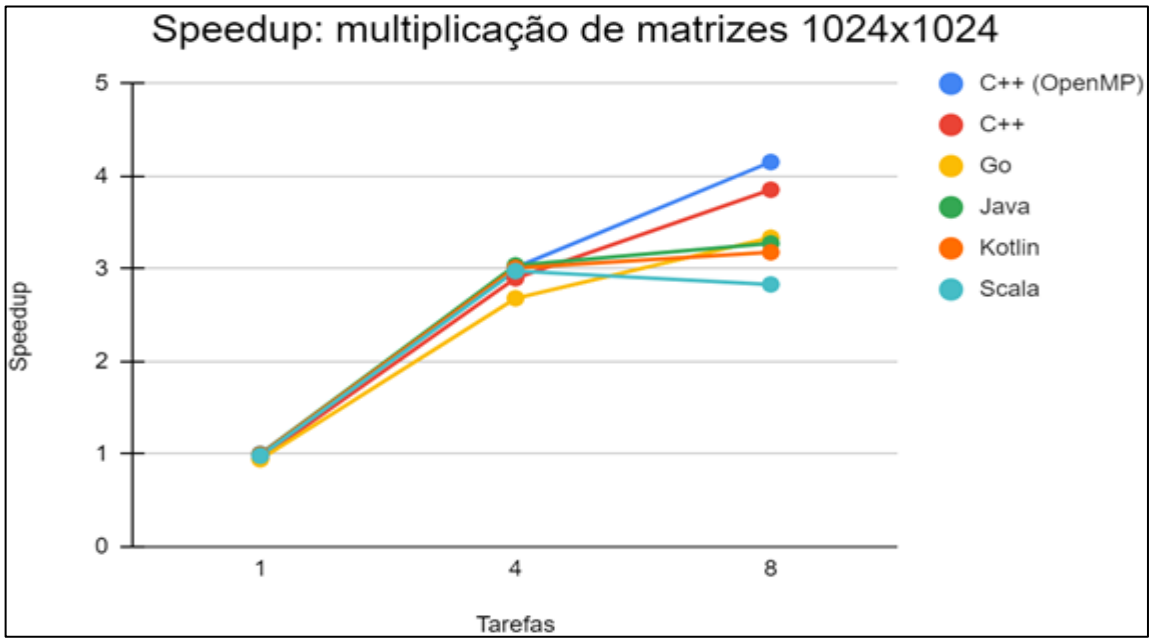


No segundo teste utilizou-se matrizes de tamanho 1024; os *speedup*s obtidos estão representados na Figura 22. O tempo de execução do algoritmo sequencial passa a ser bem maior com matrizes desse tamanho; com isso, nesse experimento nenhum mecanismo teve *speedup* menor que um. As *threads* de OpenMP seguiram com o melhor *speedup* quando oito tarefas concorrentes foram utilizadas, seguidas das *threads* da bliblioteca padrão de C++.

Ademais, notou-se que quando o experimento passou de quatro para oito tarefas, os mecanismos de concorrência de Java e Kotlin não apresentaram um aumento significativo do *speedup*; o *speedup* de Scala, por sua vez, diminuiu cerca de 5%.

As *Goroutines* de Go, por sua vez, tiveram o pior *speedup* no caso com quatro tarefas concorrentes. Entretanto, no caso com oito tarefas houve melhora, de forma que seu *speedup* ficou extremamente próximo ao de Java e melhor que o de Kotlin e Scala.

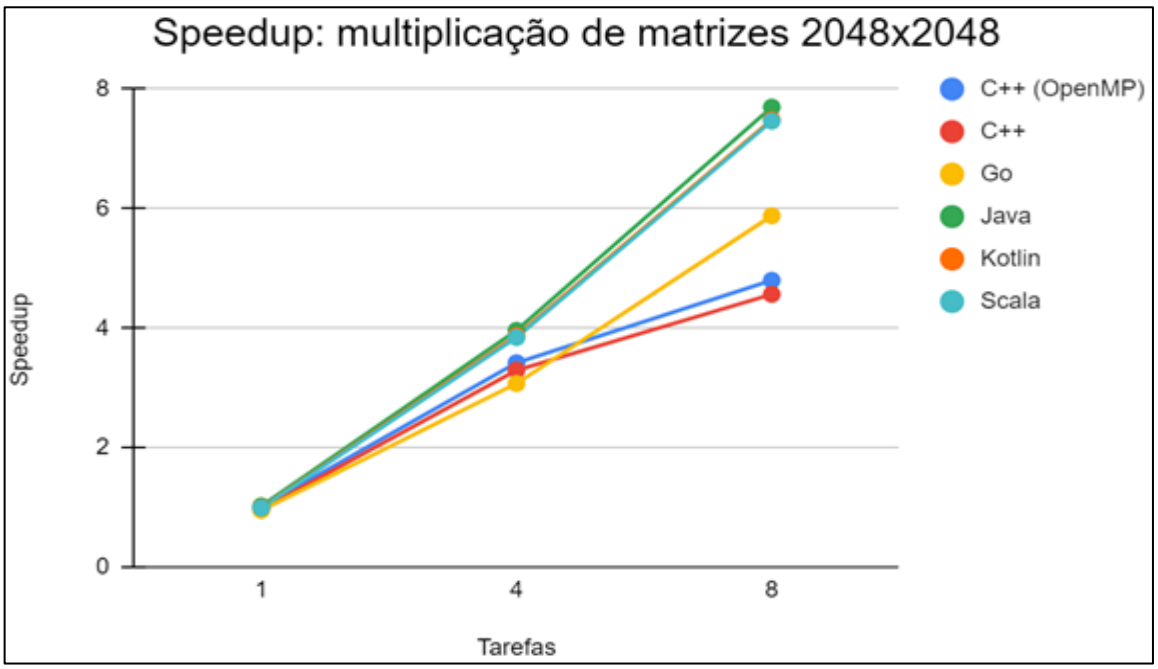
Figura 4.19 — Multiplicação de matrizes 1024x1024 na M1



Para o último experimento foram utilizadas matrizes de tamanho 2048 e os resultados obtidos estão apresentados na Figura 23. As *threads* de OpenMP e as *threads* da biblioteca padrão de C++, que nos experimentos anteriores tiveram os melhores resultados gerais, passaram a ter *speedup* bem menor que os mecanismos de concorrência das demais linguagens no caso com oito tarefas. As *Goroutines* de Go mais uma vez tiveram o pior desempenho no caso com quatro tarefas; entretanto, seu *speedup* praticamente dobrou no caso com oito tarefas.

Java teve o melhor *speedup* para esse experimento. Apesar disso, Scala e Kotlin tiveram *speedup* próximo ao de Java, com as três linguagens tendo desempenho próximo ao ideal.

Figura 4.20 — Multiplicação de matrizes 2048x2048 na M1



Conclui-se que, na média, Java foi o mecanismo que apresentou o melhor *speedup* proporcionado pela concorrência na máquina com Windows 10. Em nenhum dos experimentos Java teve *speedup* menor que um e no experimento de maior importância Java foi a linguagem que obteve os melhores *speedup*s, chegando próximo dos valores ideais.

Scala e Kotlin, apesar de apresentarem *speedup* abaixo de um no primeiro experimento, apresentaram melhoras nos experimentos seguintes. Em especial no último experimento, em que tiveram *speedup*s próximos aos de Java e, por consequência, próximos aos *speedup*s ideais.

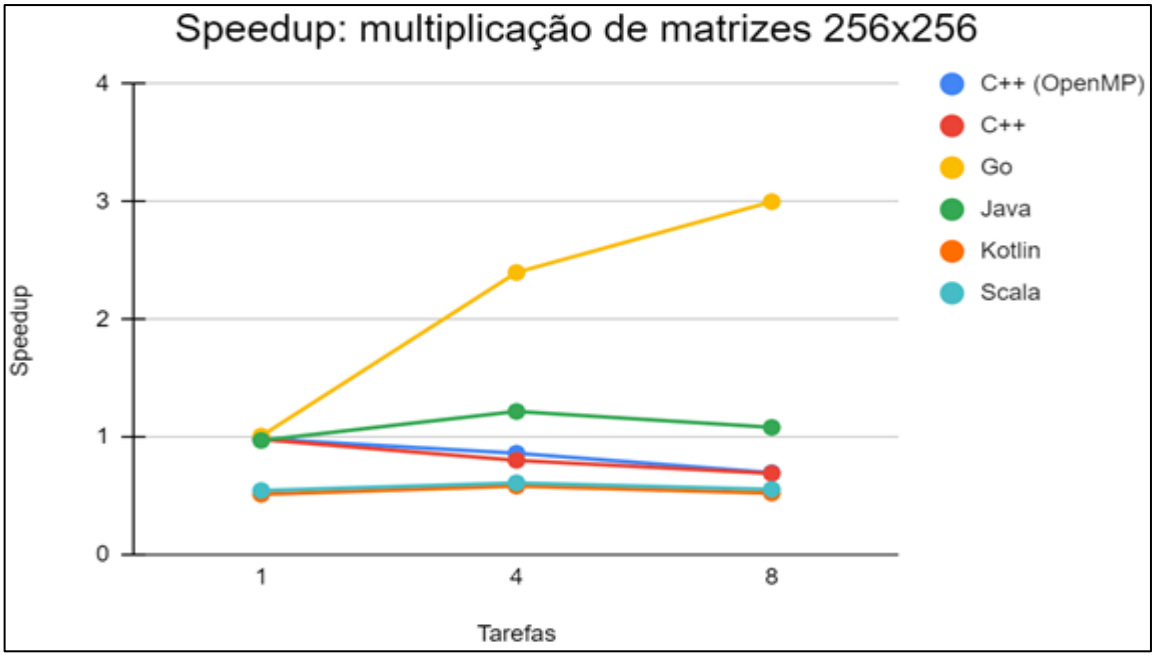
Entre os mecanismos de C++, observou-se que as *threads* de OpenMP tiveram desempenho superior às *threads* da biblioteca padrão em todos os experimentos. Além disso, ambos os mecanismos tiveram ótimos *speedup*s nos dois primeiros experimentos, mas ficaram bem atrás das demais linguagens no experimento de maior importância.

Go, por sua vez, teve ótimo desempenho no primeiro experimento, com *speedup*s praticamente idênticos aos das *threads* da biblioteca padrão de C++. Nos experimentos seguintes, contudo, foi intermediário: não se destacou nem positivamente, nem negativamente.

4.3.2 Multiplicação de matrizes em M2

O primeiro experimento para essa máquina utiliza matrizes de tamanho 256 novamente. Conforme mostrado na Figura 24, apenas as *Goroutines* tiveram *speedup* considerável. Java, por sua vez, teve um pequeno *speedup*. Todas as demais linguagens tiveram *speedup* abaixo de um; em outras palavras, tiveram desempenho paralelo pior que o desempenho sequencial.

Figura 4.21 — Multiplicação de matrizes 256x256 na M2



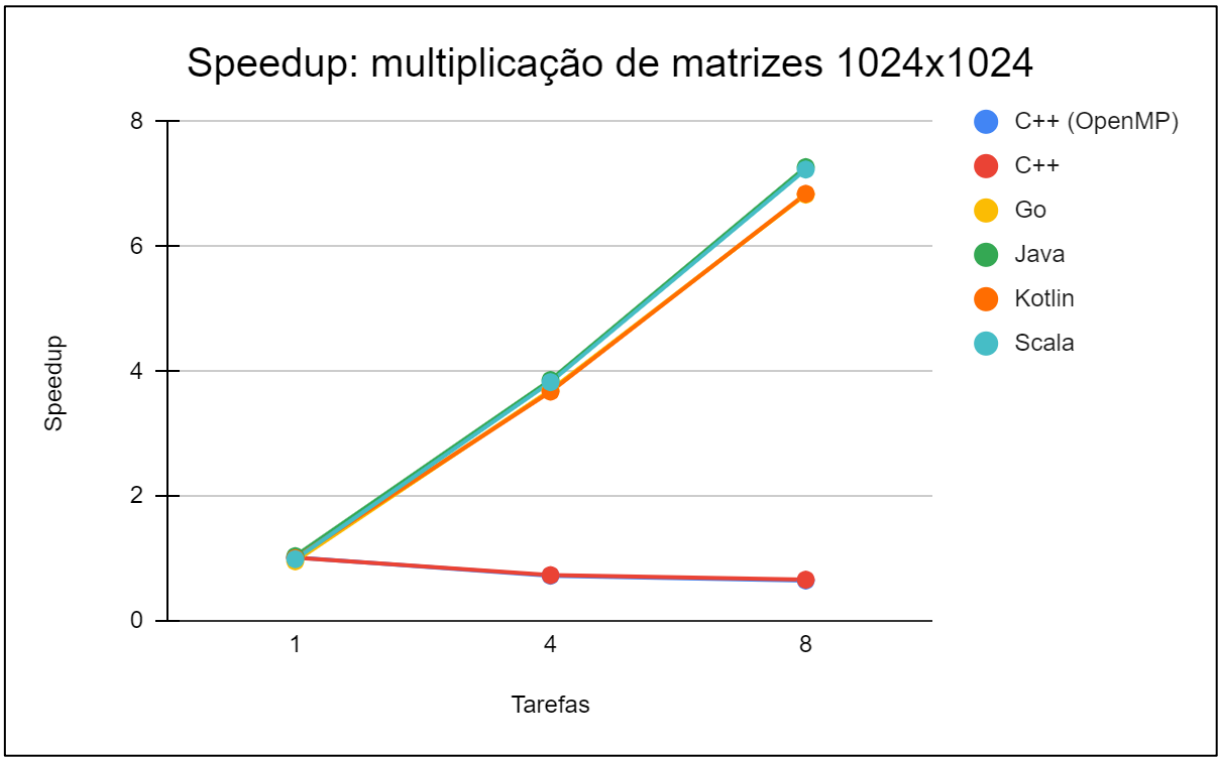
Matrizes de tamanho 1024 foram utilizadas no segundo experimento. Como pode ser visto na Figura 25, os mecanismos podem ser divididos em três grupos com duas linguagens cada; o desempenho dos mecanismos de cada grupo foi extremamente semelhante.

Um dos grupos é composto pelos dois mecanismos de C++: as *threads* da biblioteca padrão e as *threads* de OpenMP. Ambas tiveram *speedup*s menores que um. Além disso, seus desempenhos foram ainda piores que no experimento anterior tanto para o caso com quatro tarefas quanto para o caso com oito tarefas.

Outro grupo é composto pelas *Goroutines* de Go e pelas *Coroutines* de Kotlin. Ambas as linguagens tiveram ótimos *speedup*s: independentemente do número de tarefas o *speedup* delas foi próximo ao ideal.

Por fim, o último grupo é composto pelas *threads* de Java e pelos *Futures* de Scala. O seu desempenho foi extremamente próximo ao grupo composto pelas *Goroutines* e *Coroutines*, mas um pouco melhor e ainda mais próximo dos *speedup*s ideais.

Figura 4.22 — Multiplicação de matrizes 1024x1024 na M2

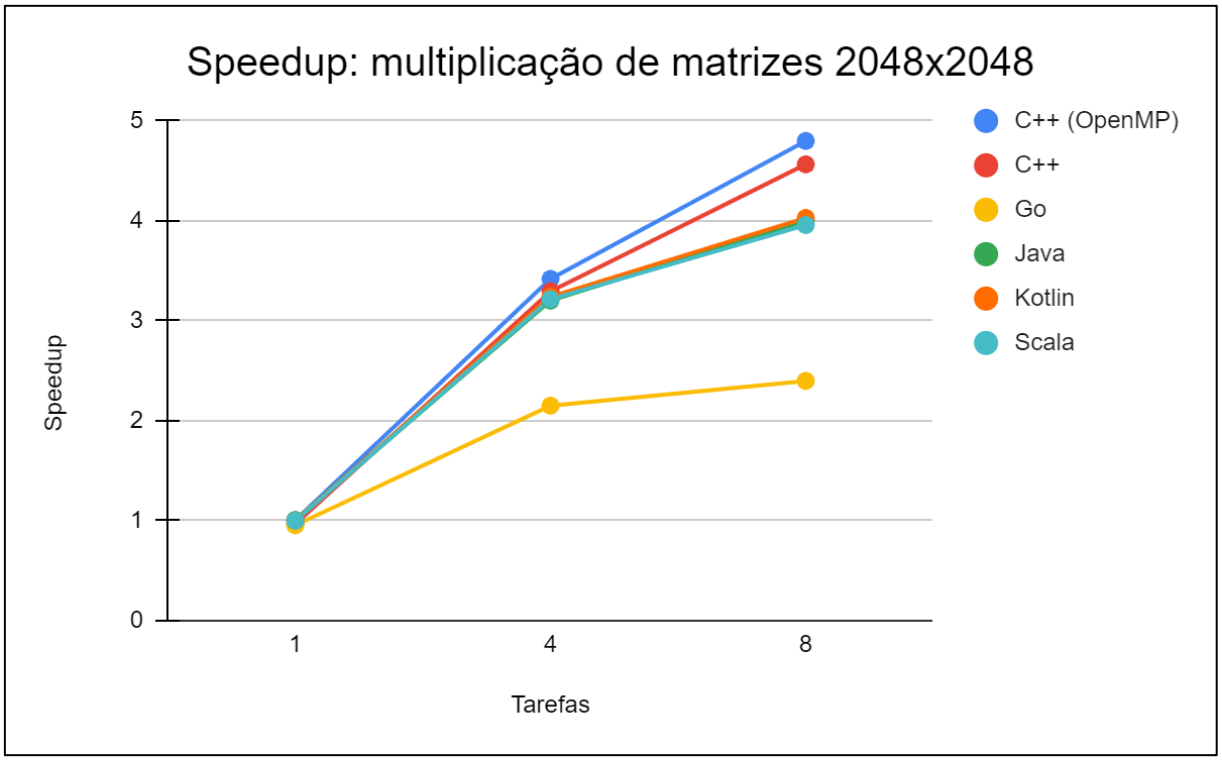


O último experimento consistiu em multiplicar matrizes de tamanho 2048. Os mecanismos de C++, que tiveram *speedup* menor que um nos dois experimentos anteriores, dessa vez tiveram os melhores desempenhos, com as *threads* de OpenMP um pouco à frente das *threads* da biblioteca padrão.

Kotlin, Scala e Java tiveram *speedup*s bastante similares. Entretanto, seus desempenhos pioraram em relação ao experimento anterior, no qual tinham obtido *speedup*s próximos aos ideais.

Go, que também havia apresentado *speedup*s próximos aos ideais no experimento com matrizes de tamanho 1024, teve *speedup*s bem menores nesse experimento, ficando em torno de 2. Tais informações podem ser vistas na Figura 26 abaixo.

Figura 4.23 — Multiplicação de matrizes 2048x2048 na M2



Dados esses resultados, considera-se que a linguagem que apresentou melhor *speedup* na máquina com Ubuntu 18.04 LTS foi Java. Apesar de ela não ter sido a melhor linguagem no teste de maior importância, os seus *speedup*s não foram muito menores que os dos mecanismos de C++ em tal teste. No segundo experimento mais importante, por sua vez, Java teve, junto a Scala, o melhor desempenho. Além disso, Java foi a única linguagem, além de Go, a não apresentar *speedup* abaixo de um em qualquer dos experimentos.

Scala e Kotlin exibiram resultados bastante semelhantes aos de Java. A principal diferença de Java para as duas linguagens é que Java não teve *speedup*s abaixo de um no menor experimento, enquanto as outras duas linguagens tiveram.

Go, por sua vez, apresentou ótimos resultados nos dois primeiros experimentos. Entretanto, as *Goroutines* tiveram o pior resultado no maior e mais importante experimento. De forma oposta à Go, os mecanismos de C++ apresentaram os melhores resultados no experimento com matrizes de tamanho 2048, mas seus *speedup*s foram abaixo de um nos dois

primeiros experimentos. Com isso, nenhuma das duas linguagens apresentou resultados muito satisfatórios.

4.4 Problema dos filósofos

Esse programa busca avaliar a justiça dos mecanismos de sincronização, conforme discutido no capítulo 3. Para dar um valor numérico à justiça, cada execução do programa calcula o número médio de vezes que os filósofos comeram, assim como o desvio padrão. Com isso, ao fim das trinta execuções de cada experimento, tem-se trinta valores da média de vezes que os filósofos comeram e trinta valores do desvio padrão. A partir desses valores é retirada a mediana da média de vezes que os filósofos comeram, assim como a mediana do desvio padrão. A partir daí calcula-se o coeficiente de variação, que é dado pelo desvio padrão dividido pela média.

O coeficiente de variação foi escolhido para representar a justiça dos mecanismos de sincronização por ser uma medida da variabilidade de um conjunto de dados que facilita a comparação de conjuntos de dados muito diferentes. Isso é útil ao problema dos filósofos pois alguns dos mecanismos de sincronização possuem desvio padrão muito maior que outros, mas isso não necessariamente quer dizer que eles são mais injustos: desvio padrão maior pode simplesmente ser um sinal de que os filósofos comeram mais vezes ao utilizar o mecanismo em questão.

O coeficiente de variação ideal para os mecanismos de sincronização é zero, o que indicaria que todos os filósofos comeram o mesmo número de vezes. Além disso, quanto maior o coeficiente de variação, maior o grau de dispersão em torno da média, o que indica mecanismos de sincronização menos justos.

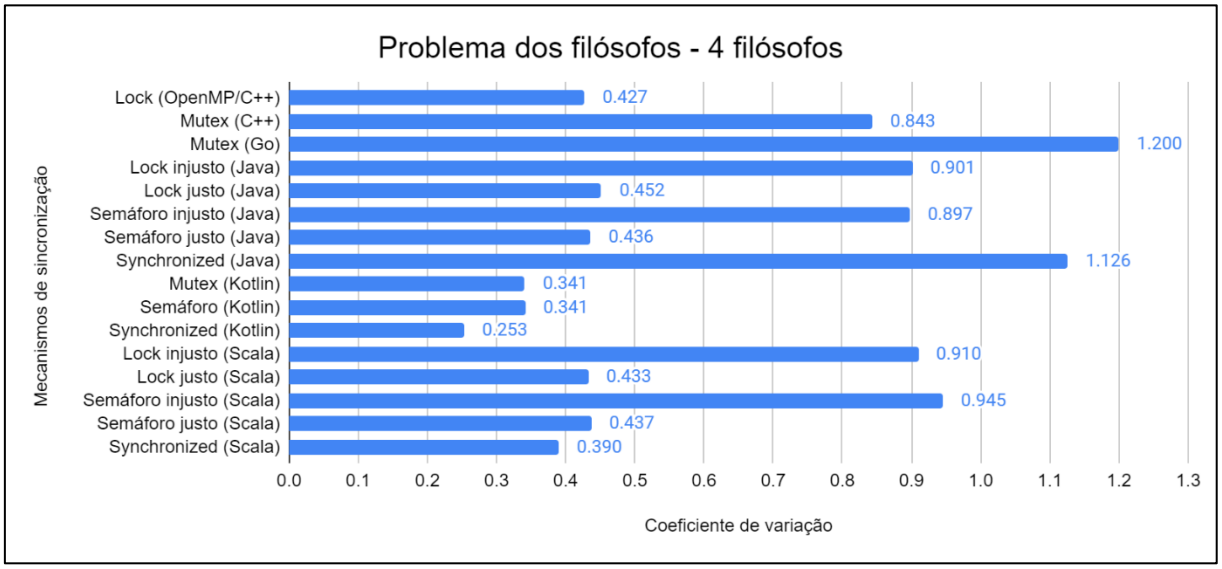
4.4.1 Problema dos filósofos em M1

No primeiro experimento da máquina com Windows 10 foram utilizados quatro filósofos; os coeficientes de variação estão representados na Figura 27. Nela pode-se notar que os três mecanismos de sincronização mais justos são os três mecanismos de Kotlin, com o bloco *synchronized* à frente do *mutex* e dos semáforos.

Além disso, como esperado, todos os mecanismos cuja *flag* de justiça foi ligada tiveram resultados bem melhores que os resultados obtidos por seus respectivos mecanismos sem a *flag* ligada.

Por fim, destaca-se que, numa comparação direta entre os mecanismos de C++, o *lock* de OpenMP teve coeficiente de variação quase duas vezes menor que o *mutex* da biblioteca padrão. O *mutex* de Go, por sua vez, teve o pior desempenho dentre todos os mecanismos de sincronização, seguido do bloco *synchronized* de Java.

Figura 4.24 — Problema dos filósofos (4 filósofos) na M1



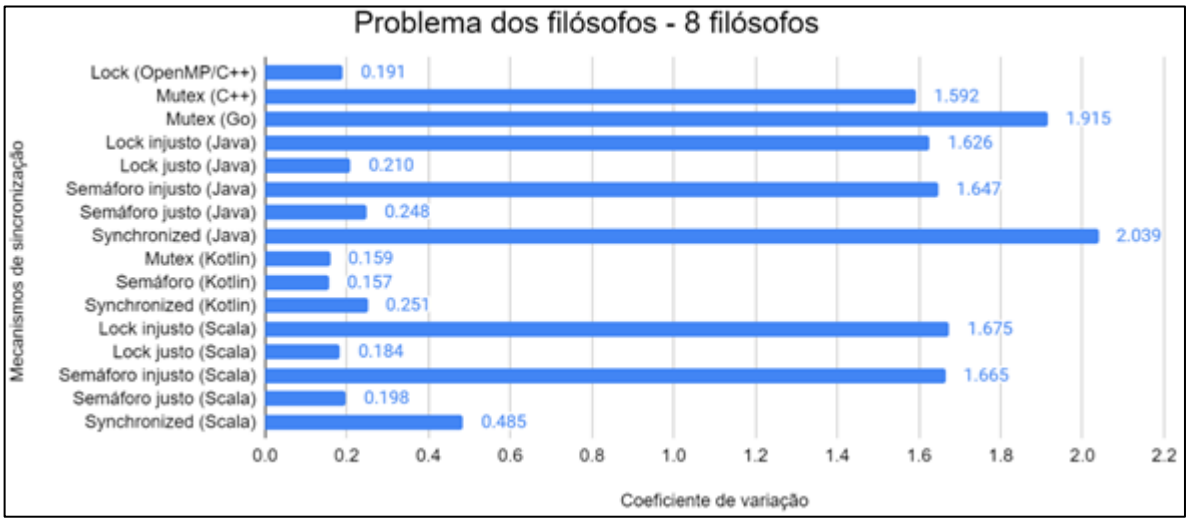
Na Figura 28 estão os coeficientes de variação obtidos para o segundo experimento, que utilizou oito filósofos. Nesse experimento os dois melhores coeficientes de variação foram obtidos, novamente, por mecanismos de Kotlin: semáforo e *mutex*, respectivamente. Entretanto, o bloco *synchronized* de Kotlin, que havia sido o mecanismo mais justo no experimento anterior, teve apenas o sétimo melhor coeficiente de variação nesse experimento.

A disparidade dos mecanismos de sincronização de C++ aumentou ainda mais: enquanto o coeficiente de variação do *lock* de OpenMP diminuiu, o coeficiente de variação do *mutex* da biblioteca padrão aumentou. Com isso, o coeficiente de variação do *mutex* da biblioteca padrão passou a ser mais de oito vezes pior que o *lock* de OpenMP.

Comportamento semelhante aos mecanismos de C++ pôde ser notado com todos os mecanismos que possuem uma versão justa e outra injusta: enquanto os mecanismos injustos ficaram ainda mais injustos, os mecanismos justos ficaram ainda mais justos.

Os dois mecanismos mais injustos continuaram sendo o bloco *synchronized* de Java e o *mutex* de Go. Entretanto, nesse experimento o bloco *synchronized* passou a ser o mecanismo mais injusto.

Figura 4.25 — Problema dos filósofos (8 filósofos) na M1

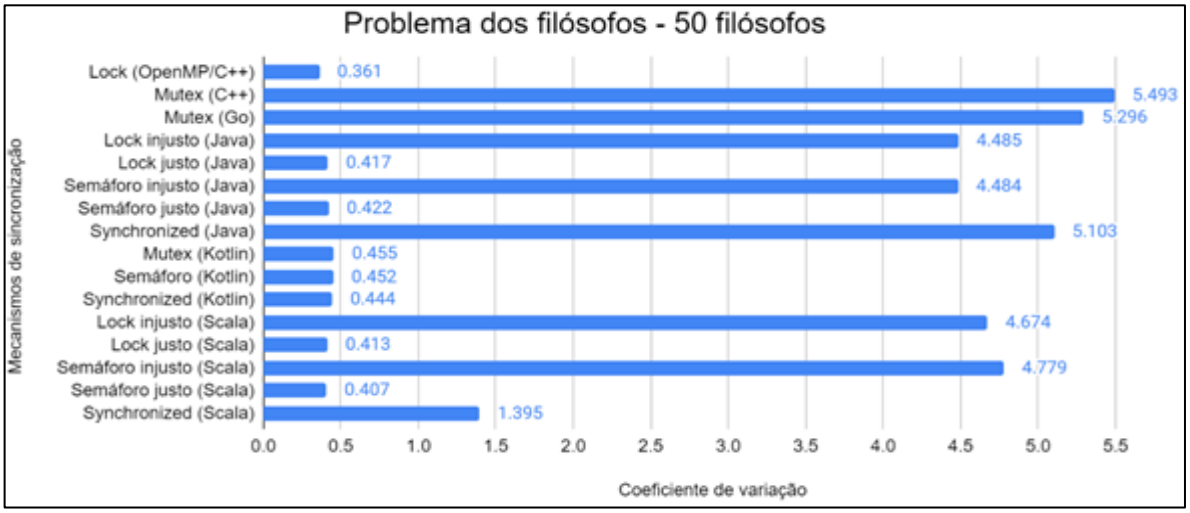


O último experimento utiliza cinquenta filósofos; os resultados coletados estão na Figura 29. Nela é possível notar que o coeficiente de variação de todos os mecanismos de sincronização aumentou em relação ao experimento anterior. Ademais, nesse experimento tanto o mecanismo mais justo quanto o mais injusto pertencem a C++: o menor coeficiente de variação foi obtido pelo *lock* de OpenMP, enquanto o maior coeficiente foi obtido pelo *mutex* da biblioteca padrão.

O *mutex* de Go voltou a ter coeficiente de variação maior que o do bloco *synchronized* de Java, com ambos ficando logo atrás do *mutex* de C++ como mecanismos mais injustos nesse experimento.

Por fim, os três mecanismos de sincronização de Kotlin, assim como os mecanismos de Java e Scala com a *flag* de justiça ligada, tiveram ótimos valores para o coeficiente de variação. Enquanto isso, os mecanismos de Java e Scala sem a *flag* de justiça ligada continuaram a apresentar péssimos resultados.

Figura 4.26 — Problema dos filósofos (50 filósofos) na M1



Nenhum mecanismo se destacou o suficiente para ser considerado mais justo que todos os demais: cada experimento teve um mecanismo diferente como o mecanismo de menor coeficiente de variação. Apesar disso, considera-se que Kotlin foi a linguagem mais justa, uma vez que os seus três mecanismos de sincronização tiveram ótimos coeficientes de variação em todos os experimentos. Além disso, em dois dos três experimentos o melhor mecanismo foi um dos mecanismos de Kotlin.

Quanto aos mecanismos mais injustos, por sua vez, há destaques: o *mutex* de Go e o bloco *synchronized* de Java estiveram entre os piores mecanismos nos três testes. Devido a isso, considera-se que ambos foram os mecanismos mais injustos.

Dentre os mecanismos de C++, o *lock* de OpenMP foi mais justo que o *mutex* da biblioteca padrão em todos os experimentos. O mesmo pode ser dito para os mecanismos de Scala e Java que possuem versão justa e injusta: enquanto a versão justa apresentava ótimos coeficientes de variação, a versão injusta apresentava péssimos resultados.

Algo interessante a ser reparado ao longo dos testes é a diferença na justiça dos blocos *synchronized* de Kotlin, Scala e Java. Enquanto o bloco *synchronized* de Kotlin apresentou ótimos resultados nos três experimentos, o bloco *synchronized* de Java apresentou péssimos resultados em todos. O bloco *synchronized* de Scala, por sua vez, teve resultados intermediários nos dois últimos experimentos.

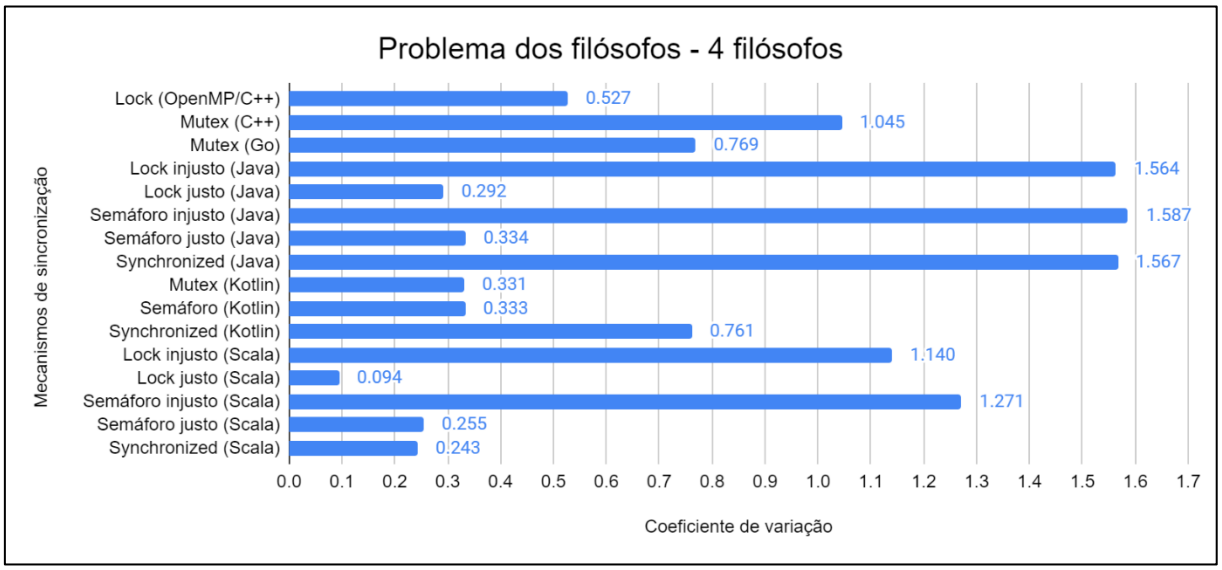
4.4.2 Problema dos filósofos em M2

O primeiro experimento nessa máquina utiliza quatro filósofos; os resultados estão na Figura 30. Nela pode-se ver que o mecanismo mais justo foi o *lock* justo de Scala, com uma grande vantagem em relação aos demais mecanismos. Além disso, da mesma forma que no experimento da máquina com Windows 10, todos os mecanismos justos tiveram desempenho bem superior ao de seus semelhantes injustos.

Os três piores mecanismos, por sua vez, foram de Java: semáforo injusto, bloco *synchronized* e *lock* injusto, respectivamente. Numa comparação dos mecanismos de C++, mais uma vez o *lock* de OpenMP foi mais justo que o *mutex* da biblioteca padrão.

O bloco *synchronized* de Kotlin e o *mutex* de Go, que haviam sido, respectivamente, o mecanismo mais justo e mais injusto no experimento da máquina com Windows 10, passaram a ter desempenho intermediário no experimento com Ubuntu 18.04 LTS.

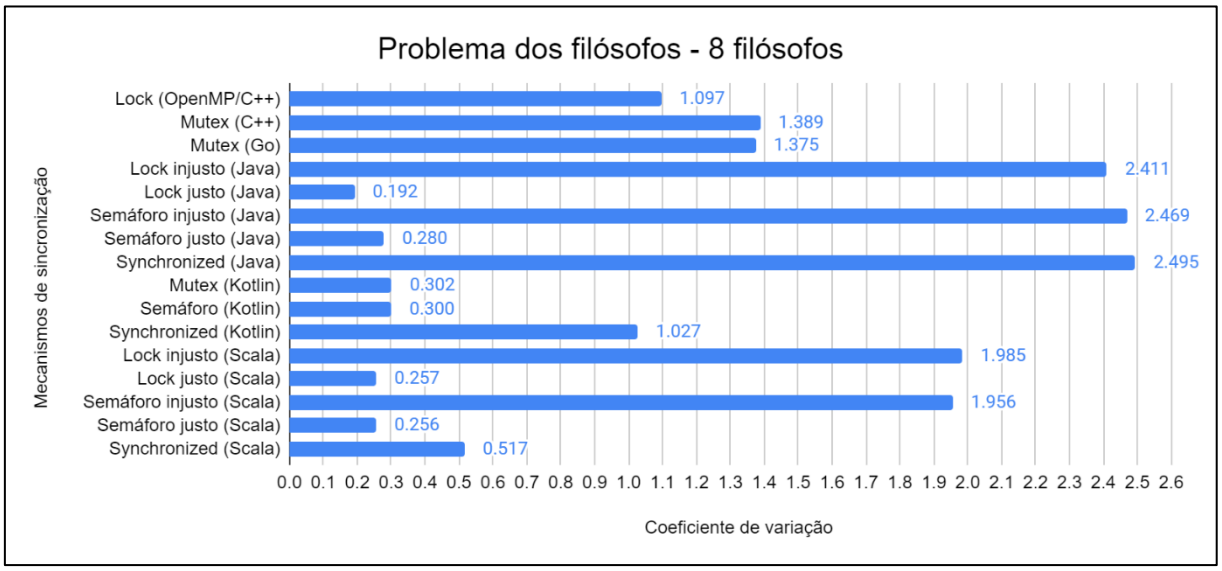
Figura 4.27 — Problema dos filósofos (4 filósofos) na M2



No segundo experimento, que utiliza oito filósofos e cujos resultados estão na Figura 31, o mecanismo mais justo foi a versão justa do *lock* de Java. Os três mecanismos mais injustos foram novamente os mesmos mecanismos do experimento anterior, apesar de a ordem entre eles ter mudado: o bloco *synchronized* de Java foi o mais injusto, seguido do semáforo injusto e do *lock* injusto de Java, respectivamente.

A relação entre os mecanismos justos e injustos se manteve, com os justos possuindo desempenho muito superior. A diferença percentual entre o *lock* de OpenMP e o *mutex* da biblioteca padrão de C++, entretanto, diminuiu bastante, apesar de o *lock* ainda ser mais justo.

Figura 4.28 — Problema dos filósofos (8 filósofos) na M2

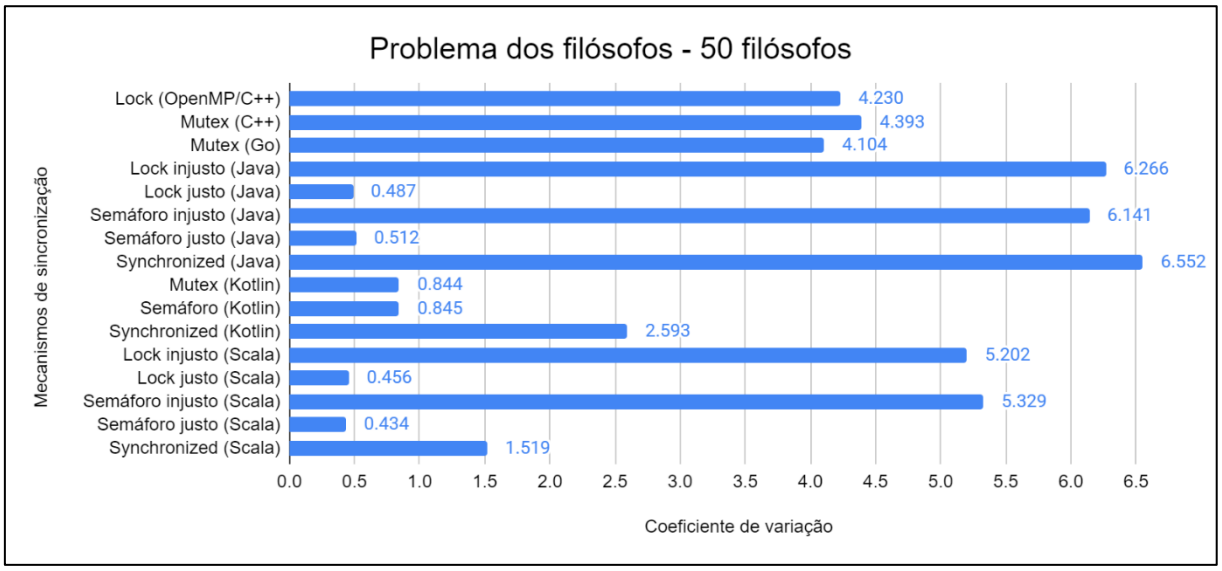


Por fim, o ultimo experimento foi feito com cinquenta filósofos. Como apresentado na Figura 32, mais uma vez os três mecanismos mais injustos foram de Java, com o bloco

synchronized mantendo seu posto como o mais injusto. O mecanismo mais justo passou a ser o semáforo justo de Scala, seguido de forma muito próxima do *lock* justo de Scala.

Os mecanismos que possuem versão justa e injusta mantiveram o padrão dos experimentos anteriores. Além disso, mais uma vez a diferença percentual entre o *lock* de OpenMP e o *mutex* da biblioteca padrão de C++ diminuiu, apesar de o *lock* continuar com coeficiente de variação um pouco melhor. Ressalta-se que, pela primeira vez em todos os experimentos, independentemente da máquina utilizada, o *mutex* de Go teve coeficiente de variação menor que o *lock* de OpenMP.

Figura 4.29 — Problema dos filósofos (50 filósofos) na M2



Assim como nos experimentos com a máquina com Windows 10, considera-se que nenhum mecanismo se destacou o suficiente para ser considerado mais justo que os demais: cada experimento teve um mecanismo diferente como o mais justo. Scala, entretanto, foi a linguagem de destaque, possuindo o mecanismo mais justo no experimento com cinquenta filósofos e no experimento com quatro filósofos.

Java, apesar de possuir os três piores mecanismos nos três experimentos realizados nessa máquina, apresentou ótimos resultados com seu *lock* justo e com seu semáforo justo. Kotlin, apesar de ter sido considerada a linguagem mais justa em M1, teve desempenho inferior a Java e Scala nessa máquina.

Tanto o *mutex* da biblioteca padrão de C++ quanto o *mutex* de Go apresentaram coeficientes de variação altos ao longo dos três testes nessa máquina. O *lock* de OpenMP, apesar de ter sido bem mais justo que ambos no primeiro teste, teve desempenho extremamente semelhante nos experimentos seguintes.

Por fim, apesar de nenhum dos três blocos *synchronized* avaliados ter tido desempenho extremamente positivo em algum dos testes, percebe-se novamente que a relação entre os três foi constante ao longo dos experimentos nessa máquina: o bloco *synchronized* de Scala teve os menores coeficientes de variação, seguido do bloco *synchronized* de Kotlin e, em último lugar, o bloco *synchronized* de Java.

5 CONCLUSÃO

Programação concorrente, nos dias atuais, é essencial para a escrita de programas fluidos, rápidos e interativos. Existem diversas linguagens de programação que oferecem mecanismos de concorrência e de sincronização, a fim de proporcionar aos seus usuários os benefícios da programação concorrente.

Entretanto, com essa diversidade surge um problema: qual linguagem escolher para a criação de programas concorrentes? Com base nessa questão surgiu esse trabalho, que busca auxiliar programadores a tomarem uma decisão. Para tanto, cinco populares linguagens de programação foram avaliadas por meio de experimentos com seus mecanismos de concorrência e sincronização.

Com os experimentos concluiu-se que os mecanismos de concorrência com menos *overhead* foram as *Goroutines* de Go e as *Coroutines* de Kotlin. Os *Futures* de Scala, por sua vez, tiveram *overhead* muito maior que os demais mecanismos, de forma que não são recomendáveis para aplicações que utilizam centenas de milhares de tarefas concorrentes. Tais conclusões são endossadas pelo fato de que os dois mecanismos foram os com menor *overhead* tanto na máquina com Windows 10 quanto na máquina com Ubuntu 18.04 LTS.

Os experimentos para uso de seções críticas demonstram que o *lock* de OpenMP, o *mutex* da biblioteca padrão de C++, o *mutex* de Go e o bloco *synchronized* de Java foram os mecanismos de sincronização com menor *overhead*. Em especial, Java foi a que manteve melhores resultados ao se considerar as duas máquinas.

Entretanto, considera-se que, excluindo-se os mecanismos de sincronização de Scala, o *overhead* de nenhum mecanismo é muito significativo: o *overhead* máximo obtido por algum deles foi 20 milissegundos. Os *overhead*s obtidos pelos mecanismos de Scala, apesar de muito maiores que os obtidos pelos demais mecanismos, também são provavelmente aceitáveis para boa parte das aplicações: o valor máximo observado foi 167 milissegundos. Com isso, conclui-se que o *overhead* dos mecanismos de sincronização pode ser relevado ao se escolher uma linguagem de programação.

Quanto ao *speedup*, considera-se que as *threads* de Java obtiveram os resultados mais constantes ao longo dos experimentos nas duas máquinas. Java foi, por exemplo, uma das duas únicas linguagens a não ter *speedup* abaixo de um em qualquer experimento, junto de Go. Entretanto, os *speedup*s de Java foram melhores que os de Go na maioria dos experimentos, em especial nos de maior importância. Com isso, conclui-se que, na média, Java é a melhor

linguagem quando se deseja maximizar o *speedup*. Também se destaca que nenhuma linguagem obteve *speedup*s ruins a ponto de não poder ser recomendada.

Os experimentos com o problema dos filósofos apresentaram os resultados mais dispersos. Ao todo foram realizados três experimentos em cada máquina; cada um dos seis experimentos totais teve um mecanismo de sincronização diferente como o mais justo. Além disso, cada máquina teve uma linguagem de destaque: na máquina com Windows 10 Kotlin foi considerada a linguagem mais justa, enquanto na máquina com Ubuntu 18.04 LTS Scala foi considerada a mais justa.

Apesar dessa dispersão, nota-se que Kotlin, Scala e Java tiveram mecanismos com bons resultados em todos os experimentos. Go, por sua vez, teve desempenho de ruim a intermediário, dependendo da máquina, sendo contraindicado quando se busca justiça. Já C++ teve desempenho ótimo na máquina com Windows 10 ao utilizar os *locks* de OpenMP, mas, independentemente do mecanismo utilizado, também variou de intermediário a ruim na máquina com Ubuntu 18.04 LTS.

Portanto, recomenda-se utilizar Java ou alguma das duas linguagens baseadas em Java quando se deseja justiça na concorrência; especificamente para Java e Scala, recomenda-se a versão justa dos mecanismos que também possuem versão injusta.

Por fim, considera-se que Java e Kotlin foram as linguagens que obtiveram os melhores resultados gerais. Ambas foram muito justas e tiveram pouco *overhead* para o uso de seções críticas. Java se destaca em relação a Kotlin no *speedup*, enquanto Kotlin se destaca em relação a Java no *overhead* para a criação de tarefas concorrentes.

Logo, dois questionamentos de projeto auxiliam a escolher uma das duas linguagens: se o principal objetivo da concorrência é proporcionar *speedup* por meio da criação de poucas tarefas concorrentes, sugere-se dar preferência a Java; se, por outro lado, o programa exige a criação de dezenas ou centenas de milhares de tarefas concorrentes, como servidores que devem ser capazes de lidar com centenas de milhares de conexões simultâneas sem degradação de performance (BEHREN et al., 2003), recomenda-se optar por Kotlin.

Já numa comparação direta entre os mecanismos de C++, notou-se que os mecanismos de OpenMP tiveram desempenho melhor ou igual aos mecanismos da biblioteca padrão em todos os testes. Portanto, apesar de as *threads* da biblioteca padrão oferecerem mais flexibilidade de programação, recomenda-se utilizar OpenMP sempre que possível.

Para trabalhos futuros recomenda-se analisar outros mecanismos de concorrência e sincronização das linguagens já avaliadas, como *thread pools* e barreiras cíclicas. Além disso, sugere-se estender a análise por meio da adição de novas linguagens de programação populares

e com amplo suporte à concorrência, como Rust. Por fim, apesar de alguns mecanismos de sincronização, como barreiras e variáveis de condição, não terem sido avaliados por esse trabalho devido aos problemas utilizados, eles são mecanismos importantes para a resolução de outros problemas; portanto, recomenda-se a avaliação de desempenho de mecanismos de sincronização não avaliados por esse trabalho.

REERÊNCIAS

ASADOLLAH, Sara *et al.* A model for systematic monitoring and debugging of starvation bugs in multicore software. **SCTDCP 2016: Proceedings of the 1st International Workshop on Specification, Comprehension, Testing, and Debugging of Concurrent Programs**, New York, USA, p. 7–11, sept., 2016.

AYGUADÉ, Eduard *et al.* The design of OpenMP tasks. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v. 20, n. 3, p. 404–418, mar., 2009.

BARBOSA, Valmir *et al.* Deadlock models in distributed computation. **SAC '16: Proceedings of the 31st Annual ACM Symposium on Applied Computing**, [S.l.], p. 538–541, apr., 2016.

BEHREN, Rob *et al*. Capriccio: scalable threads for internet services. **ACM SIGOPS Operating Systems Review**, New York, USA, v. 37, n. 5, p. 268–281, oct., 2003.

BRESLAV, Andrey. **Kotlin 1.0 Released: Pragmatic Language for JVM and Android**, 15 de fev. de 2016. Disponível em: <https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/>. Acesso em: 8 de abr. de 2021.

CAPRETZ, Luiz. A Brief History of the Object-Oriented Approach. **ACM SIGSOFT Software Engineering Notes**, New York, USA, v. 28, n. 2, p. 1–10, mar., 2003.

CARULLO, Giuliana. **Implementing Effective Code Reviews**: How to Build and Maintain Clean Code. 1.ed. Berkeley, CA: Apress, 2020.

CARVER, Richard; TAI, Kuo-Chung. **Modern Multithreading**: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs. 1.ed. Hoboken, NJ: Wiley-Interscience, 2005.

CASSEL, Lillian *et al.* Concurrency and parallelism in the computing ontology. **ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education**, New York, USA, p. 402, july, 2009.

CHAPMAN, Barbara; JOST, Gabrielle; PAS, Ruud van der. **Using OpenMP**: Portable Shared Memory Parallel Programming. 1.ed. USA: The MIT Press, 2007.

CHEUNG, King. **Augmented Marked Graphs**. 1.ed. [S.l]: Springer, 2014.

DIJKSTRA, Edsger. Hierarchical ordering of sequential processes. **Acta Informatica**, [S.l], v. 1, n. 2, p. 115–138, june, 1971.

DONOVAN, Alan; KERNIGHAN, Brian. **The Go Programming Language**. 1.ed. USA: Addison-Wesley Professional, 2015.

EAGER, Derek.; ZAHORJAN, John; LAZOWSKA, Edward. Speedup versus efficiency in parallel systems. **IEEE Transactions on Computers**, [S.l.], v. 38, n. 3, p. 408–423, mar., 1989.

FREITAS, Luiz; MATOS, Fernando; EDUARDO, Paulo. Estudo experimental sobre paralelismo na linguagem Go usando *Goroutines*. **XIII Encontro Anual de Computação**, [S.l], p. 194–201, maio, 2017.

HALLER, Philipp et al. ***FUTURE*S AND PROMISES**. Disponível em: <https://docs.scala-lang.org/overviews/core/*Future*s.html>. Acesso em: 8 de abr. de 2021.

HSIEH, Chyuan; UNGER, Elizabeth. Resource scheduling: Specification and proof techniques. **CSC '88: Proceedings of the 1988 ACM sixteenth annual conference on Computer science**, New York, USA, p. 429–437, feb., 1988.

HUNT, John. **Advanced Guide to Python 3 Programming**. 1.ed. Cham, Switzerland: Springer, 2019.

IBM CORPORATION. **Example: Using *mutex*es in a Java program**. Disponível em: <https://www.ibm.com/docs/en/i/7.4?topic=threads-example-using-*mutex*es-in-java-program>. Acesso em: 9 de maio de 2021.

JAIN, Raj. **The Art of Computer Systems Performance Analysis**: Techniques for Experimental Design, Measurement, Simulation, and Modeling. 1.ed. [S.l.]: Wiley, 1991.

JONES, Micah. **Mastering Scala: *Future*s**, 18 de abr. de 2017. Disponível em: <https://www.credera.com/insights/mastering-scala-*Future*s>. Acesso em: 8 de abr. de 2021.

KOSARAJU, Sambasiva. Limitations of Dijkstra's Semaphore Primitives and Petri Nets. **SOSP '73: Proceedings of the fourth ACM symposium on Operating system principles**, New York, USA, p. 122–136, jan., 1973.

KOTLIN FOUNDATION. **Coroutines basics**. Disponível em: <https://kotlinlang.org/docs/coroutines-basics.html>. Acesso em: 8 de abr. de 2021.

KOTLIN FOUNDATION. **Default Dispatcher**. Disponível em: <https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-default.html>. Acesso em: 8 de abr. de 2021.

KOTLIN FOUNDATION. **FAQ**. Disponível em: <https://kotlinlang.org/docs/faq.html>. Acesso em: 8 de abr. de 2021.

KOTLIN FOUNDATION. **Semaphore**. Disponível em: <https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.sync/-semaphore/index.html/>. Acesso em: 8 de abr. de 2021.

KOTLIN FOUNDATION. **synchronized**. Disponível em: <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/synchronized.html>. Acesso em: 8 de abr. de 2021.

KRAMER, Jeff; MAGEE, Jeff. **Concurrency**: State Models and Java Programs. 2.ed. [S.l]: Wiley, 2004.

KRIEMANN, Ronald. **Implementation and Usage of a Thread Pool based on POSIX Threads**. Max-Planck-Institute for Mathematics in the Sciences, 2004, Leipzig, Germany.

KUHN, Bob; PETERSEN, Paul; O'TOOLE, Eamonn. OpenMP versus threading in C/C++. **Concurrency: Practice and Experience**, [S.l.], v. 12, n. 12, p. 1165–1176, nov., 2000.

LASSER, Daniel J. Productivity of multiprogrammed computers—progress in developing an analytic prediction method. **Communications of the ACM**, New York, USA, v. 12, n. 12, p. 678–684, dec., 1969.

LEE, Edward; SESHIA, Sanjit. **Introduction to Embedded Systems**: a Cyber-Physical Systems Approach. 2.ed. [S.l]: MIT Press, 2017.

LEVITUS, Marcia. **Mathematical Methods in Chemistry**. 1.ed. [S.l.]: LibreTexts, 2020.

LEWIS, Bil; BERG, Daniel. **PThreads Primer**: A Guide to Multithreaded Programming. 1.ed. Mountain View, CA: SunSoft Press, 1996.

LING, Yibei; MULLEN, Tracy; LIN, Xiaola. Analysis of optimal thread pool size. **ACM SIGOPS Operating Systems Review**, New York, USA, v. 34, n. 2, p. 42–55, apr., 2000.

MATTSON, Timothy G. How good is OpenMP. **Scientific Programming**, [S.l.], v. 11, n. 2, p. 81–93, apr., 2003.

MEYERSON, Jeff. The go programming language. **IEEE Software**, [S.l.], v. 31, n. 5, p. 101–104, sept., 2014.

MOQVIST, Erik. **Counting semaphores**. Disponível em: <https://simba-os.readthedocs.io/en/latest/library-reference/sync/sem.html>. Acesso em: 3 de maio de 2021.

MULLER, Stefan; WESTRICK, Sam; ACAR, Umut. Fairness in responsive parallelism. **Proceedings of the ACM on Programming Languages**, New York, USA, v. 3, n. ICFP, aug., 2019.

NICHOLS, Bradford et al. **Pthreads Programming**. 1.ed. Sebastopol, CA: O'Reilly, 1996.

NIEMEYER, Patrick; LEUCK, Daniel. **Learning Java**. 4.ed. Sebastopol, CA: O'Reilly, 2013.

OAKS, Scott; WONG, Henry. **Java Threads**. 3.ed. [S.l.]: O'Reilly, 2004.

ODERSKY, Martin. **The Scala Language Specification**. Version 2.9. Switzerland: 2011.

ODERSKY, Martin *et al.* **An Overview of the Scala Programming Language**. École Polytechnique Fédérale de Lausanne, 2004, Lausanne, Switzerland.

ORACLE CORPORATION. **Class ReentrantLock**. Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html>. Acesso em: 5 de abr. de 2021.

ORACLE CORPORATION. **Class Semaphore**. Disponível em: <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html>. Acesso em: 5 de abr. de 2021.
ORACLE CORPORATION. **Defining Multithreading Terms**. Disponível em: <https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html>. Acesso em: 5 de abr. de 2021.

ORACLE CORPORATION. **Interface Lock**. Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html>. Acesso em: 5 de abr. de 2021.

PARSONS, David. **Foundational Java**: Key Elements and Practical Programming. 2.ed. Cham, Switzerland: Springer, 2020.

PRABHAKAR, Raghu; KUMAR, Rohit. **Concurrent Programming in Go**. University of California, Los Angeles, 2011, Los Angeles, USA.

SANDÉN, Bo. Coping with Java threads. **Computer**, Washington, D.C, USA, v. 37, n. 4, p. 20–27, apr., 2004.

SCROUSTRAP, Bjarne. A history of C++: 1979-1991. **HOPL-II: The second ACM SIGPLAN conference on History of programming languages**, New York, USA, p. 271–297, apr., 1993.

SILBERSCHATZ, Abraham; GALVIN, Peter; GAGNE, Greg. **Operating System Concepts**. 9.ed. USA: Wiley, 2018.

SILVEIRA, Kaue. **A Comparative Study of Programming Models for Concurrency**. UFRGS, 2012. Monografia (Bacharelado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2012.

SNYDER, Herbert; DAVENPORT, Elisabeth. What does it really cost? Allocating indirect costs. **The Bottom Line**, [S.l], v. 10, n. 4, p. 158–164, dec., 1997.

SOTTILE, Matthew; RASMUSSEN, Craig; MATTSON, Timothy. **Introduction to Concurrency in Programming Languages**. 1.ed. [S.l]: Chapman & Hall/CRC, 2009.

TOGASHI, Naohiro; KLYUEV, Vitaly. Concurrency in Go and Java: Performance analysis. **ICIST 2014 - Proceedings of 2014 4th IEEE International Conference on Information Science and Technology**, Shenzhen, China, p. 213–216, apr., 2014.

TU, Tengfei *et al.* Understanding Real-World Concurrency Bugs in Go. **ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems**, New York, USA, p. 865–878, apr., 2019.

VARGAS, William. **Avaliação de concorrência e de sincronização no Android**. UFRGS, 2019. Monografia (Bacharelado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2019.

ANEXO A – 25 LINGUAGENS DE PROGRAMAÇÃO MAIS POPULARES

De acordo com a pesquisa realizada em 2020 pelo site Stack Overflow, que obteu 57.378 respostas, a Tabela 2 apresenta as 25 linguagens de programação mais populares, assim como a porcentagem de respondentes que as utilizam.

Tabela 2 — 25 linguagens de programação mais populares

Posição	Linguagem	Porcentagem de utilizadores
1	JavaScript	67,7%
2	HTML/CSS	63,1%
3	SQL	54,7%
4	Python	44,1%
5	Java	40,2%
6	Bash/Shell/PowerShell	33,1%
7	C#	31,4%
8	PHP	26,2%
9	TypeScript	25,4%
10	C++	23,9%
11	C	21,8%
12	Go	8,8%
13	Kotlin	7,8%
14	Ruby	7,1%
15	Assembly	6,2%
16	VBA	6,1%
17	Swift	5,9%
18	R	5,7%
19	Rust	5,1%
20	Objective-C	4,1%
21	Dart	4,0%
22	Scala	3,6%
23	Perl	3,1%
24	Haskell	2,1%
25	Julia	0,9%

ANEXO B – MEDIANA DOS TEMPOS DE EXECUÇÃO DA MULTIPLICAÇÃO DE MATRIZES

A seguir são apresentadas as medianas dos tempos de execução obtidos para o experimento com a multiplicação de matrizes. Eles foram omitidos da análise pois o foco desse trabalho é avaliar métricas estritamente relacionadas à concorrência; no caso da multiplicação de matrizes, a métrica de interesse foi o *speedup*. A Tabela 3 apresenta as medianas de tempo de execução obtidas para a multiplicação de matrizes quadradas de tamanho 256 em M1.

Tabela 3 — Tempos de execução da multiplicação de matrizes 256x256 em M1

Mecanismo / Tarefas	*Threads* (OpenMP/C++)	*Threads* (C++)	*Goroutines* (Go)	*Threads* (Java)	*Coroutines* (Kotlin)	*Futures* (Scala)
Sequencial	0,0080s	0,0070s	0,0312s	0,0308s	0,0301s	0,0390s
1	0,0082s	0,0073s	0,0346s	0,0317s	0,0586s	0,0740s
4	0,0026s	0,0026s	0,0114s	0,0206s	0,0485s	0,0624s
8	0,0021s	0,0022s	0,0096s	0,0213s	0,0481s	0,0712s

A Tabela 4 apresenta as medianas de tempo de execução obtidas para a multiplicação de matrizes quadradas de tamanho 1024 em M1.

Tabela 4 — Tempos de execução da multiplicação de matrizes 1024x1024 em M1

Mecanismo / Tarefas	*Threads* (OpenMP/C++)	*Threads* (C++)	*Goroutines* (Go)	*Threads* (Java)	*Coroutines* (Kotlin)	*Futures* (Scala)
Sequencial	1,21s	1,21s	5,66s	5,02s	4,87s	4,25s
1	1,21s	1,27s	5,99s	5,03s	4,90s	4,33s
4	0,40s	0,42s	2,11s	1,65s	1,62s	1,43s
8	0,29s	0,31s	1,70s	1,53s	1,53s	1,50s

A Tabela 5 apresenta as medianas de tempo de execução obtidas para a multiplicação de matrizes quadradas de tamanho 2048 em M1.

Tabela 5 — Tempos de execução da multiplicação de matrizes 2048x2048 em M1

Mecanismo / Tarefas	*Threads* (OpenMP/C++)	*Threads* (C++)	*Goroutines* (Go)	*Threads* (Java)	*Coroutines* (Kotlin)	*Futures* (Scala)
Sequencial	30,08s	30,05s	60,62s	54,31s	57,82s	57,74s
1	29,95s	31,12s	63,82s	53,08s	57,96s	58,35s
4	8,80s	9,12s	19,73s	13,71s	14,93s	15,03s
8	6,27s	6,58s	10,31s	7,05s	7,72s	7,74s

A Tabela 6 apresenta as medianas de tempo de execução obtidas para a multiplicação de matrizes quadradas de tamanho 256 em M2.

Tabela 6 — Tempos de execução da multiplicação de matrizes 256x256 em M2

Mecanismo / Tarefas	*Threads* (OpenMP/C++)	*Threads* (C++)	*Goroutines* (Go)	*Threads* (Java)	*Coroutines* (Kotlin)	*Futures* (Scala)
Sequencial	0,0182s	0,0178s	0,0483s	0,0642s	0,0607s	0,0799s
1	0,0185s	0,0182s	0,0479s	0,0662s	0,1183s	0,1467s
4	0,0211s	0,0222s	0,0201s	0,0528s	0,1041s	0,1310s
8	0,0262s	0,0258s	0,0161s	0,0594s	0,1160s	0,1440s

A Tabela 7 apresenta as medianas de tempo de execução obtidas para a multiplicação de matrizes quadradas de tamanho 1024 em M2.

Tabela 7 — Tempos de execução da multiplicação de matrizes 1024x1024 em M2

Mecanismo / Tarefas	*Threads* (OpenMP/C++)	*Threads* (C++)	*Goroutines* (Go)	*Threads* (Java)	*Coroutines* (Kotlin)	*Futures* (Scala)
Sequencial	2,10s	2,08s	5,21s	12,06s	12,07s	11,25s
1	2,06s	2,06s	5,46s	11,63s	12,02s	11,38s
4	2,90s	2,84s	1,41s	3,12s	3,29s	2,94s
8	3,24s	3,16s	0,76s	1,69s	1,76s	1,56s

A Tabela 8 apresenta as medianas de tempo de execução obtidas para a multiplicação de matrizes quadradas de tamanho 2048 em M2.

Tabela 8 — Tempos de execução da multiplicação de matrizes 2048x2048 em M2

Mecanismo / Tarefas	*Threads* (OpenMP/C++)	*Threads* (C++)	*Goroutines* (Go)	*Threads* (Java)	*Coroutines* (Kotlin)	*Futures* (Scala)
Sequencial	74,96s	72,89s	125,27s	155,59s	156,61s	157,02s
1	74,64s	75,50s	131,38s	154,51s	156,38s	157,11s
4	21,92s	22,12s	58,24s	48,60s	48,32s	48,83s
8	15,62s	15,97s	52,22s	38,89s	38,87s	39,67s