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

WILLIAM WILBERT VARGAS

Avaliação de concorrência e de sincronização no Android

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

Orientador: Prof. Dr. Claudio Fernando Resin Geyer
Coorientador: Dr. Julio Cesar Santos dos Anjos

Porto Alegre
2019

AGRADECIMENTOS

Agradeço primeiramente à minha família, em especial agradeço aos meus pais Carla Cristina Wilbert Vargas e Alexandre Wasen Vargas, por todo apoio e motivação que me deram durante todas as etapas de minha vida.

Agradeço ao meu orientador Claudio Fernando Resin Geyer e a todos os professores que encontrei durante a minha graduação na UFRGS por todos os valiosos conselhos oferecidos tanto para esse trabalho quanto para as mais diversas situações durante o curso. Agradeço também ao meu coorientador Julio Cesar Santos dos Anjos pela revisão desse texto.

RESUMO

A consolidação do mercado de smartphones ampliou o acesso das pessoas à computação. Um dos principais objetivos das aplicações desenvolvidas para esse mercado é proporcionar uma boa experiência ao usuário. Uma aplicação precisa manter a sua interface fluída com feedback imediato para que possa fornecer uma boa experiência de usuário. A partir do momento que tarefas custosas passam a ser executadas nesse ambiente, essas características somente se tornam possíveis quando a aplicação inicia fluxos de execução concorrentes para que essas tarefas executem em *background* sem interromper a interação da aplicação com o usuário. Dessa forma, o objetivo desse trabalho é avaliar alguns mecanismos para criação e gerenciamento desses fluxos de concorrência no sistema operacional Android, que é o sistema operacional mais utilizado para smartphones. Para tanto, considera diferentes testes para avaliar mecanismos como Threads, framework HaMeR, Thread Pool, Kotlin Coroutines, IntentServices e AsyncTasks. Considerando características como sua escalabilidade frente aos recursos paralelos de hardware, o uso eficiente de recursos de rede, a sobrecarga introduzida para realizar o gerenciamento de muitos fluxos concorrentes e a sua justiça na distribuição de recursos entre os fluxos concorrentes. Além disso, serão avaliados mecanismos tradicionais de sincronização, como Semáforos, Lock e Condition, blocos Synchronized de Kotlin e variáveis atômicas, no contexto de aplicações móveis para Android. Os resultados obtidos indicam que existe um ganho de escalabilidade no uso de recursos paralelos ao se usar Kotlin coroutines, contudo existe uma alta sobrecarga durante a execução dos fluxos concorrentes ao se usar esse mecanismo. Indicam também que o HaMeR framework tende a ser mais justo para gerenciar um cenário com muitos fluxos competindo pela obtenção de recursos. Indicam ainda que o uso de semáforos introduz uma sobrecarga menor do que os outros mecanismos de sincronização quando tarefas com diferentes funções operam em um ambiente cooperativo.

Palavras-chave: Android. Concorrência. Sincronização. HaMeR framework. Kotlin coroutines.

Concurrency and Synchronization Evaluation on Android

ABSTRACT

The consolidation of the smartphone market has expanded people's access to computation. One of the main goals of the applications developed for this market is to provide a good user experience. An application needs to provide a fluid user interface and immediate feedback in order to provide a good user experience. Once costly tasks are executed in mobile environments, these characteristics only become possible when the application starts concurrent executions streams allowing these tasks to execute in the background without disrupting the application's interaction with the user. Thus, the purpose of this work is to evaluate some mechanics for creating and managing these concurrent execution streams in the Android operating system, that is the most used operating system for smartphones. Therefore, it considers different tests to evaluate mechanisms such as Threads, HaMeR framework, Thread Pool, Kotlin Coroutines, IntentServices and AsyncTasks. Considering characteristics such as its scalability against parallel hardware resources, the efficient use of network resources, the overhead introduced to perform the management of many concurrent executions and fairness of resource distribution among concurrent execution stream. In addition, traditional synchronization mechanisms, such as Semaphores, Lock and Condition, Kotlin's Synchronized blocks and atomic variables, will be evaluated in the context of Android mobile applications. The obtained results indicate that there is a scalability gain in the use of parallel resources when using Kotlin coroutines, however there is an introduction of overhead when running concurrent execution streams using this mechanism. They also indicate that HaMeR framework tends to be fairer to managing a scenario with multiples execution streams competing for resources. The results also show that using semaphores introduces a lower overhead than other synchronization mechanisms when tasks with different functions operate in a cooperative environment.

Keywords: Android. Concurrency. Synchronization. HaMeR framework. Kotlin coroutines.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ART	Android Runtime
CPU	Central Processing Unit
DEX	Dalvik Executable
DVM	Dalvik Virtual Machine
ELF	Executable and Linkable Format
FFT	Transformada Rápida de Fourier
IFFT	Transformada Rápida Inversa de Fourier
IDE	Integrated Development Environment
JVM	Java Virtual Machine
LCS	Longest Common Subsequence
MVC	Model-View-Controller
QoS	Quality of Service
RAM	Random Access Memory
SDK	Software Development Kit
SO	Sistema Operacional
UI	User Interface
VM	Virtual Machine
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Com bilhões de usuários ativos ao redor do mundo, o mercado dos smartphones cresceu em complexidade e em diversidade de uso. Isso foi possível graças à melhoria dos hardwares disponíveis e pela flexibilização dos mecanismos de desenvolvimento. Por outro lado, aplicações móveis têm como um de seus objetivos principais o fornecimento de uma boa experiência ao usuário. Esse objetivo somente pode ser atingido se as aplicações forem capazes de proporcionar um feedback imediato às ações do usuário em uma interface fluída. Apesar disso parecer simples, com a diversificação das aplicações, tarefas custosas computacionalmente passam a ser executadas nos hardwares desses dispositivos. Assim novos nichos de mercado de aplicações móveis podem ser atendidos. A conciliação dessas tarefas custosas com a manutenção da interface fluida pode se tornar um desafio para os desenvolvedores. Exemplos de tarefas que podem atrapalhar a fluidez da interface por possuírem alta demanda computacional são: Jogos que exigem um amplo suporte gráfico, aplicações de redes neurais, interações com a rede, aplicações Big Data, entre outras.

A saída para esse problema é o uso de fluxos de execução concorrentes para permitir que outras tarefas sejam executadas no dispositivo ao mesmo tempo que o usuário é atendido por outra tarefa e continua recebendo feedback imediato. Junto com a necessidade de fluxos concorrentes vem a necessidade de mecanismos de sincronização desses fluxos para manter a corretude do programa como um todo. Esse problema não é novo na computação, possuindo soluções de concorrência disponíveis há várias décadas. Dessa forma, desde o começo da computação móvel em smartphones, alguma forma de desenvolver fluxo concorrente já estava presente.

Contudo, diferentes tarefas exigem diferentes modos de lidar com a concorrência e com a sincronização para manter um uso eficiente dos recursos de hardware. Sabendo disso, os sistemas operacionais móveis oferecem aos desenvolvedores diferentes APIs que lidam com concorrência e sincronização de formas diferentes no baixo nível. Nesse contexto, o objetivo desse trabalho é avaliar os diferentes mecanismos de concorrência e de sincronização do sistema operacional Android. O sistema operacional Android é o mais utilizado para smartphones comerciais e possui uma ampla gama de opções quando se trata de concorrência, uma vez que permite o uso de soluções de concorrência de linguagens de programação de propósitos gerais, como Java e Kotlin, e introduz outras soluções específicas da sua plataforma.

1.1 Objetivos

O objetivo desse trabalho é realizar uma avaliação comparativa de alguns mecanismos de concorrência e de sincronização disponíveis para o desenvolvimento de aplicações Android. Essa avaliação busca testar os seguintes aspectos dos mecanismos:

- Escalabilidade frente a recursos paralelos de hardware;
- Sobrecarga introduzida para gerenciamento de fluxos concorrentes;
- Justiça no compartilhamento de recursos de software;
- Eficiência em ambientes cooperativos que exigem sincronização de grupos de tarefas diferentes.

Assim, esse trabalho busca realizar a contribuição de facilitar a escolha dentre os mecanismos de concorrência e de sincronização disponíveis dependendo do cenário de programação concorrente que se apresenta no desenvolvimento de aplicações Android.

Os mecanismos de concorrência avaliados nesse trabalho foram escolhidos considerando sua popularidade e diferenças de suas especificações. Assim, os seguintes mecanismos foram escolhidos:

- AsyncTask;
- HaMeR *framework*;
- IntentService;
- Thread de Java;
- Thread Pool;
- Kotlin coroutines.

Por sua vez, os mecanismos de sincronização avaliados nesse trabalho foram escolhidos dentre os mecanismos mais tradicionais presentes na área de sincronização e pela sua popularidade. Desse modo os seguintes mecanismos foram escolhidos:

- Lock e Condition;
- Semáforos;
- Synchronized de Kotlin;
- Variáveis atômicas.

1.2 Estrutura desse Documento

Esse trabalho segue a seguinte organização. No capítulo 2, será realizada uma contextualização na programação concorrente e na sincronização dos dispositivos Android, bem como será apresentada uma breve descrição dos mecanismos de concorrência e de sincronização que serão considerados nesse trabalho. No capítulo 3, a metodologia de avaliação utilizada nesse trabalho será descrita. No capítulo 3, também são apresentadas as implementações dos testes desenvolvidas ao longo desse trabalho para explorar diferentes aspectos das soluções propostas. Por fim, no capítulo 4, os resultados dos testes serão apresentados juntamente com observações sobre os mecanismos em específico e observações comparativas entre os mecanismos que puderam ser feitas a partir dos resultados dos testes. Essas observações serão suportadas usando testes de hipóteses estatísticas com valor α de 0,05.

2 CONCEITOS BÁSICOS

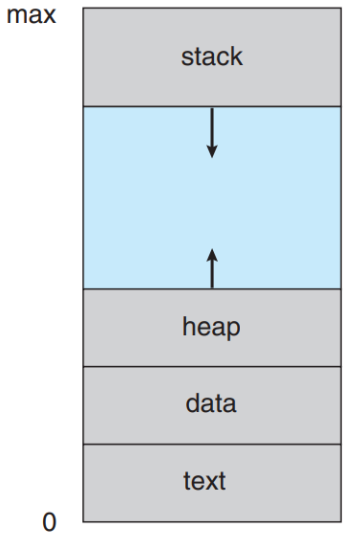
Este capítulo aborda conceitos e definições teóricas importantes para esse trabalho. Os conceitos se iniciam como uma breve descrição da ideia de processos e de threads no contexto de processadores na seção 2.1. Em seguida, na seção 2.2, alguns conceitos básicos relativos à programação concorrente e à sincronização são apresentados. Após isso, na seção 2.3, a linguagem Kotlin que foi utilizada para o desenvolvimento dos programas de teste é apresentada. Na seção 2.4, conceitos do sistema operacional Android como sua arquitetura, seu mecanismo de compilação, como realiza a execução das aplicações e conceitos relativos ao desenvolvimento de aplicações Android são apresentados. A seção 2.4 também apresenta conceitos de concorrência e de sincronização da plataforma Android. Por fim, as seções 2.5 e 2.6 explicam os mecanismos de concorrência e de sincronização avaliados nesse trabalho.

2.1 Processos e Threads

Para poder realizar uma análise de concorrência e de sincronização de um sistema computacional, é importante entender como um sistema operacional moderno lida com esses conceitos e que estruturas usa para permitir que técnicas de execução não sequencial sejam aplicadas. Essa seção aborda brevemente conceitos como processos e threads inseridos no contexto de um sistema operacional moderno. Boa parte dos conceitos dessa seção foram retirados de Silberschatz (2013) e de Tanenbaum (2016).

De acordo com Silberschatz (2013), um processo é um programa em execução, logo, não somente é o código do programa, mas também inclui o *program counter*, o conteúdo dos registradores do processador, geralmente inclui uma pilha, que armazena dados temporários, uma seção de dados com variáveis globais, e também pode incluir um *heap* para memória dinamicamente alocada, como ilustrado na Figura 2.1.

Figura 2.1: Processo em memória



(a) Imagem retirada de (SILBERSCHATZ, 2013)

Processos dão suporte à possibilidade de haver operações (pseudo) concorrentes mesmo quando há apenas uma CPU disponível. Desse modo vários processos podem estar em memória ao mesmo tempo, mesmo que apenas um subconjunto deles esteja efetivamente em execução no conjunto de CPUs do computador.

Threads de controle, ou simplesmente threads, por sua vez, são componentes de processos que permitem execuções quase paralelas, exceto pelo espaço de endereçamento que é compartilhado, como se fossem processos independentes. O termo *multithread* é usado para descrever a situação de permitir múltiplas threads no mesmo processo.

Existem vários benefícios no uso de *multithread* conforme Silberschatz (2013) lista:

1. **Responsividade**: Um programa constituído de múltiplas threads pode ter um conjunto de threads bloqueadas e mesmo assim seguir executando outras threads com tarefas que não dependem dos bloqueios;
2. **Compartilhamento de recursos**: threads de um mesmo processo compartilham código e dados por padrão;
3. **Economia**: o gerenciamento de threads é menos custoso que o gerenciamento de processos;
4. **Escalabilidade**: conforme aumenta a quantidade de recursos paralelos de uma arquitetura multiprocessador, os benefícios do uso de *multithreading* também aumentam, pois, mais threads podem executar em paralelo.

Outro componente fundamental para suporte a existência de processos e threads em um SO é o escalonador, que realiza a escolha de qual processo e de qual thread devem executar assim que uma CPU fica livre. O escalonador também pode bloquear temporariamente um processo que estava em execução para que outro possa executar. É responsabilidade do escalonador a escolha dos processos que devem executar, que é feita a partir do conjunto de processos que estão prontos para a execução. Essa escolha pode ser baseada em algoritmos de prioridade de tarefas ou utilizando algoritmos como o *Round-Robin*, que executa as tarefas por um período de tempo fixo após o qual realiza a troca da contexto, ou como o *Shortest Job First*, que estima o tempo de execução de cada tarefa e executa as mais curtas primeiro (ARPACI-DUSSEAU, 2018). O valor da prioridade de cada processo depende de como o sistema operacional avalia a importância de cada um. O desenvolvedor também pode assinalar valor de prioridade para discriminar os fluxos de execução de sua aplicação quanto a sua importância, esses valores são usados pelo escalonador para escolher as próximas tarefas da aplicação que devem executar.

2.2 Programação Concorrente e Sincronização

Essa seção trata de definições e conceitos sobre a concorrência e a sincronização de fluxos de execução, que são conceitos centrais desse trabalho.

"Um programa concorrente contém dois ou mais processos que trabalham juntos para realizar uma tarefa" (ANDREWS, 2000, s 1.1, p 2). Nessa definição cada processo executa uma sequência de comandos. Diferente de um programa sequencial que possui apenas uma thread de controle, um programa concorrente pode apresentar múltiplas threads de controle que podem se comunicar para atingir um objetivo comum.

Existem vários benefícios ao se usar programas concorrentes, como anteriormente citados, contudo, também existem problemas que aparecem em programas concorrentes e não aparecem em programas sequenciais, dentre os quais destacam-se:

1. **Condição de corrida**: ocorre quando dois ou mais processos estão lendo ou escrevendo dados compartilhados e o resultado final depende da ordem em que os processos executam. (TANENBAUM, 2016).
2. **Deadlocks**: Ocorre quando um grupo de processos está esperando por eventos que não ocorrem. Isso geralmente acontece quando existem dependências cíclicas entre os processos ou threads em execução quanto a recursos de acesso exclusivo.

3. **Starvation**: Ocorre quando uma thread ou um processo que precisa de um recurso para prosseguir nunca consegue obtê-lo, logo fica indefinidamente parado. Isso pode acontecer, por exemplo, quando a thread está competindo com outra de maior prioridade para acessar um recurso.

A maioria das aplicações do mundo real não podem ser paralelizadas de forma eficiente sem que sejam adicionados custos para a comunicação e o gerenciamento entre processos (HERLIHY, 2008). Logo, técnicas de sincronização entre diferentes fluxos de execução reduzem a concorrência entre as tarefas, mas fazem isso com objetivo de garantir a corretude do sistema como um todo, buscando permitir a resolução dos problemas de concorrência apresentados anteriormente.

Um conceito central da sincronização é a atomicidade de uma ação, ou seja, uma ação é atômica se o fluxo em que ela está executando não puder ser interrompido até que a ação seja finalizada. Passando a garantir que, uma vez que uma operação seja iniciada, nenhum outro processo possa acessar recursos que estão sendo utilizados pela ação atômica. Assim os demais processos concorrentes deverão esperar até que a ação tenha sido concluída ou bloqueada (TANENBAUM, 2016). Essa atomicidade é essencial para solucionar problemas de sincronização e evitar a condição de corrida. Conseguir certo nível de atomicidade no acesso a recursos é o objetivo de mecanismos de sincronização como locks, semáforos, monitores e variáveis atômicas.

No contexto de sincronização de fluxos de execução, uma seção crítica é um segmento de código no qual um processo pode alterar variáveis comuns, atualizar tabelas, escrever em arquivos e assim por diante. Tal região precisa dos seguintes requisitos (SILBERSCHATZ, 2013):

1. **Exclusão mútua**: se um processo está em sua seção crítica, então nenhum dos outros processos do mesmo sistema podem estar executando sua seção crítica.
2. **Progresso**: se nenhum processo está executando na seção crítica e existem processos que desejam entrar na sua seção crítica, o escalonador não deve postergar indefinidamente a escolha de qual processo vai receber permissão para entrar na seção crítica.
3. **Espera limitada**: Deve existir um limite para a espera de um processo até obter acesso à sua seção crítica.

2.3 Kotlin

Kotlin é uma linguagem de programação moderna estaticamente tipada que busca resolver muitos problemas de Java, como *null pointer exceptions* e códigos excessivamente verbosos (MOSKALA, 2017). Kotlin usa a JVM, portanto seu compilador gera *bytecode* Java o que permite que Kotlin realize chamadas para códigos Java e vice-versa (SAMUEL, 2017). A linguagem começou a ser desenvolvida pela JetBrains em 2011, e teve sua primeira versão estável lançada apenas em 2016. Em 2017 foi anunciada pela Google como uma das linguagens oficiais para Android e em 2019 passou a ser a linguagem preferida para desenvolvimento Android.

Segundo Moskala (2017) as principais características de Kotlin são:

1. **Segurança**: Kotlin oferece segurança em termos da existência de valores imutáveis e verifica em tempo de compilação a possibilidade da existência de erros do tipo *null pointer exception*. Possui tipagem estática estrita, o que significa que o programador precisa, explicitamente, informar ao compilador se uma variável pode armazenar valores nulos.
2. **Debugging fácil**: ao oferecer a possibilidade de distinguir entre dados mutáveis de imutáveis e incentivar o uso de variáveis imutáveis, Kotlin pode detectar em tempo de compilação problemas que apareceriam se um valor que não deve mudar durante a execução do problema pode ter seu valor alterado.
3. **Concisão**: Boa parte da verbosidade de Java foi eliminada, logo precisa-se de menos código para completar tarefas comuns o que torna a linguagem mais fácil de ler e de entender.
4. **Interoperabilidade**: Uma vez que Kotlin compila para *bytecode* Java, pode interoperar com a ampla gama de bibliotecas Java já existentes sem a introdução de qualquer sobrecarga de tempo. Algumas bibliotecas de Java possuem versões específicas para Kotlin que permitem um uso mais idiomático da linguagem. De forma semelhante, código Java também pode chamar funções de bibliotecas desenvolvidas em Kotlin.
5. **Versatilidade**: Muitas plataformas podem ser atingidas por programas desenvolvidos em Kotlin. A linguagem pode ser usada não somente para Android, mas também para aplicações *backend* e *frontend*, aplicações desktop e para a construção de sistemas com Gradle.

Favorecida pelo fato de ser desenvolvida pela JetBrains, que também desenvolve IDEs, Kotlin possui um amplo suporte dos mais variados IDEs incluindo Android Studio, IntelliJ Idea e Eclipse além da existência de plugins para vários outros ambientes.

2.4 Android

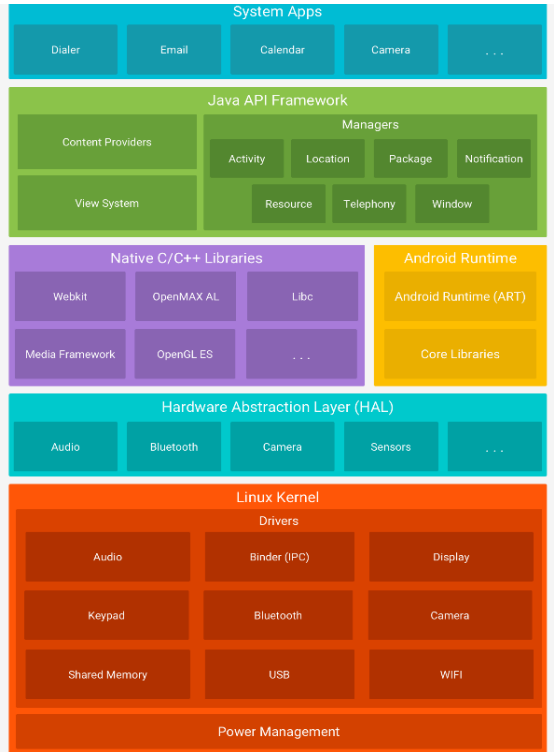
Android é um sistema operacional para dispositivos móveis que foi lançado originalmente em 2003 e comprado pela Google em 2005. É um sistema baseado em Linux de código aberto e gratuito (LENGURE, 2015).

O Android como sistema operacional é escrito majoritariamente em Java (interface de usuário) e C/C++ (núcleo e algumas bibliotecas), bem como trechos estruturais definidos em XML. Define as linguagens Java, C++ e Kotlin como linguagens oficiais para desenvolvimento de suas aplicações. Em 2019, Kotlin foi escolhida pela Google como a linguagem preferida para desenvolvimento de aplicações Android e é a linguagem usada para o desenvolvimento dos testes nesse trabalho.

Uma vez que esse trabalho busca realizar uma avaliação sobre componentes do sistema Android, a seguir são explicados alguns conceitos sobre esse sistema operacional.

2.4.1 Arquitetura Android

Figura 2.2: Arquitetura Android



(a) Imagem retirada de (GOOGLE, 2019b)

A Figura 2.2 apresenta a arquitetura do Android, a qual consiste de (KAUR, 2014):

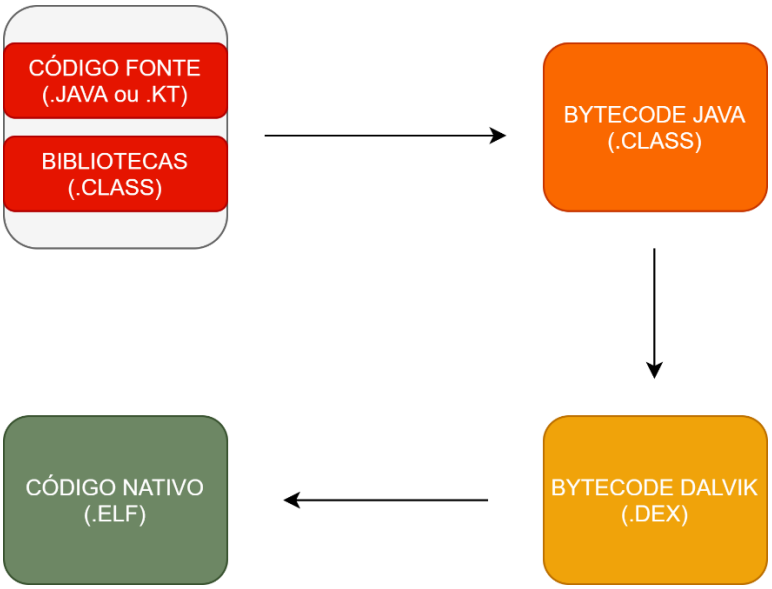
1. **Camada de aplicação**: camada usada pelas aplicações instaladas no dispositivo
2. **Framework de Aplicação**: inclui classes e serviços necessários para as aplicações da camada superior, tais serviços estão relacionados ao gerenciamento de recursos, notificações, activities, entre outros.
3. **Android Runtime e bibliotecas**: O Android runtime é responsável pela execução das aplicações Android. As bibliotecas Android são escritas em C/C++ e não podem ser acessadas diretamente pelas aplicações, somente através do uso da camada de framework de aplicação. Dentre essas bibliotecas existem funcionalidades para acessar recursos web, recursos multimídia, entre outros.
4. **Camada de abstração de Hardware**: fornece interfaces padrão para facilitar o acesso a funcionalidades do hardware por frameworks Java de mais alto nível.
5. **Kernel do Linux**: Núcleo da arquitetura Android com serviços de mais baixo nível do sistema operacional, como gerenciamento de memória, de energia e de segurança.

Vale notar que as estruturas Android da camada de aplicação e do framework de aplicação são implementadas tanto em Java quanto em Kotlin. E as camadas de bibliotecas nativas, Android Runtime e HAL são implementadas em C/C++.

Antes do Android 5.0 a camada de abstração de Hardware não estava definida e o acesso às funcionalidades de Hardware era feito pelas bibliotecas do Android acessando diretamente os recursos do Kernel Linux. Os testes realizados nesse trabalho foram desenvolvidos e executados em dispositivos que seguem a nova arquitetura do Android, ou seja, a arquitetura mostrada na Figura 2.2.

2.4.2 Execução das Aplicações

Figura 2.3: Compilação de uma aplicação Android



A Figura 2.3 ilustra o processo de compilação de uma aplicação Android do código fonte até se tornar um código nativo, essa figura se refere a versões do Android posteriores a versão 5.0. Antes dela, até a versão 4.4 do Android, lançada em junho de 2014, a execução das aplicações Android ocorria sobre a Dalvik Virtual Machine (DVM), uma máquina virtual que suporta apenas um processo. Desse modo, quando um aplicativo era iniciado uma instância de DVM era criada para gerenciar esse aplicativo e essa instância era encerrada junto com o aplicativo. Esse esquema foi adotado para proporcionar isolamento de processos e um ambiente para a execução das aplicações baseados no *bytecode* Dalvik. Os aplicativos, que eram geralmente escritos em Java, eram compilados para *bytecode* da JVM e em seguida esse

bytecode era traduzido para o *bytecode* Dalvik e armazenados como arquivos DEX pronto para ser invocado quando necessário (KAUR, 2014).

A partir da versão 5.0 do Android, lançada em 2014, o uso da DVM se tornou obsoleto e as aplicações passaram a executar sobre o Android Runtime (ART). Aplicativos executando sobre o ART são compilados para código de máquina nativo seguindo o modelo mostrado na Figura 2.3. Os principais benefícios da mudança estão no aumento da eficiência pela eliminação da interpretação do *bytecode* Dalvik e na redução do consumo de energia. Apesar da mudança ainda existe retro compatibilidade com arquivos DEX. De fato, esses arquivos ainda são utilizados no pipeline de compilação de uma aplicação Android para ART, que ganhou mais um passo. Inicialmente o *bytecode* da JVM é traduzido para *bytecode* DVM (arquivos DEX) que, por sua vez, é traduzido para arquivos ELF para serem instalados e executados sobre o Android Runtime.

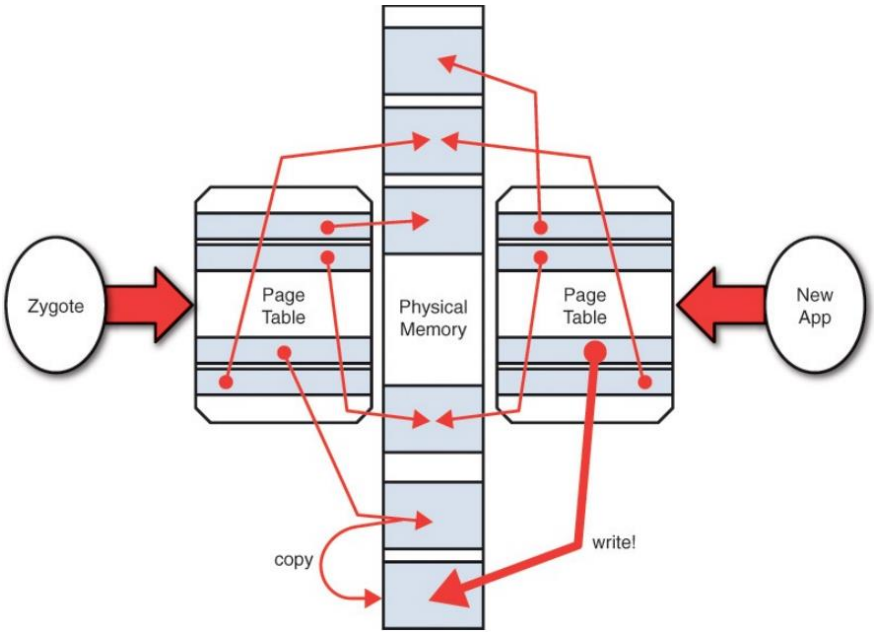
2.4.3 Modelo de Aplicações Android

Um aplicativo Android é formado por um conjunto de componentes: *Activities*, *Services*, *Providers* e *Receivers*. O sistema operacional garante que esses componentes estarão disponíveis quando forem necessários. Para cada componente que precise ser executado é iniciado um novo processo para a aplicação e para cada processo é mantido um valor de prioridade que varia dinamicamente conforme características do componente, de modo que componentes visíveis aos usuários possuam maior prioridade do que componentes executando tarefas não visíveis aos usuários. Desse modo, quando o Android precisa de mais memória para novos componentes ele finaliza os processos de menor prioridade (MEIKE, 2016).

Cada aplicação Android é um clone de um processo base chamado Zigoto. Uma única instancia Zigoto é iniciada quando o sistema é ligado para realizar tarefas de inicialização. Quando outras aplicações são iniciadas, elas se conectam a sockets do Zigoto que realiza um clone de si mesmo criando um novo processo no nível do kernel do SO. Esse novo processo, que passa a ser propriedade da aplicação iniciada, compartilha memória com o Zigoto. Esse compartilhamento de memória é gerenciado pelo SO de modo diferente do compartilhamento de memória entre threads. A memória que uma aplicação compartilha com o Zigoto possui objetivo de permitir uma inicialização mais rápida das aplicações sem a necessidade de recarregar bibliotecas geralmente usadas. Contudo, quando a nova aplicação tenta mudar um valor do espaço compartilhado ocorre uma alocação de memória que copia a página da memória que está sendo acessada para um espaço de memória exclusivo do novo processo. Essa nova

página não é mais compartilhada, para evitar que a memória do Zigoto seja alterada por processos de aplicação, como ilustrado na Figura 2.4 (MEIKE, 2016). Outra vantagem desse esquema é que pouca memória é necessária para que uma aplicação seja iniciada, já que o compartilhamento com o Zigoto permite a alocação de páginas de memória por demanda, somente quando algum endereço de memória compartilhado está prestes a ser alterado pela aplicação. Entretanto, nota-se que essa solução não segue nem o modelo padrão de processos, que não compartilham memória, nem o modelo padrão de threads, com compartilhamento de memória, mas é uma exceção implementada pelo Android por questões de desempenho.

Figura 2.4: Endereçamento do processo Zigoto e de um novo processo.



(a) A figura destaca o que ocorre quando o processo de aplicação precisa escrever em área compartilhada com o Zigoto, ocorre uma cópia dessa página para outras posições da memória e a aplicação passa a referenciar essa cópia ao invés da página original. A escrita, no entanto, não é necessária em páginas de frameworks carregados pelo Zigoto, esses continuam compartilhados entre todas as aplicações. Imagem retirada de (MEIKE, 2016).

A configuração básica das aplicações Android usa apenas uma thread chamada de UI Thread ou Thread principal, essa thread funciona como um loop processando tarefas que são postas em uma fila. Cada tarefa é processada até que seja completa antes de iniciar outra tarefa. Muito embora esse modelo preveja apenas o uso de uma thread, quase todas as aplicações Android não triviais precisam iniciar múltiplas threads (MEIKE, 2016).

2.4.4 Desenvolvimento

O desenvolvimento de aplicações Android ocorre com o uso do Android *software development kit* (SDK), que inclui *debugger*, bibliotecas com códigos prontos, emuladores de dispositivos, documentação e códigos de exemplo. Uma aplicação Android geralmente é escrita com alguma das três linguagens oficiais (Kotlin, Java ou C++) e com a UI descrita em XML. Contudo, outras linguagens que não compilam para JVM também podem ser usadas para desenvolvimento Android ou desenvolvimento multiplataforma, mas esses casos, geralmente possuem um suporte restrito da API.

Como mencionado anteriormente, uma aplicação Android é um conjunto de componentes, dentre os quais vale destacar as *Activities* que são objetos que representam telas da aplicação e que podem ficar visível ao usuário. Cada *Activity* representa uma tela e possuirá métodos para interação com o usuário e atualizações dos seus itens (DIMARZIO, 2008). Durante o desenvolvimento, todos os componentes da aplicação precisam ser registrados no manifesto da aplicação (MEIKE, 2016). Esse manifesto é um arquivo XML que descreve a aplicação ao SO.

2.4.5 Concorrência e Sincronização no Android

Como explicado na subseção 2.4.3, uma aplicação Android possui uma thread chamada de UI Thread (Thread principal), que funciona processando tarefas e eventos sequencialmente conforme são postos na sua fila de execução. Assim, nesse esquema básico, não existe a possibilidade de múltiplos fluxos concorrentes pertencentes à mesma instância de uma aplicação. O que torna necessário a existência de outros mecanismos para que uma execução concorrente possa acontecer no Android, tais mecanismos permitem a definição de tarefas que executam em outra thread da aplicação que não é a thread principal. Todas as outras threads da aplicação, são chamadas de *worker* thread (GOOGLE, 2016), esse termo é usado de forma intercambiável com *background* thread.

Como os códigos Android construídos em Java ou em Kotlin compilam para *bytecode* Java, uma alternativa possível é o uso dos mecanismos de concorrência e de sincronização já presentes nessas linguagens que serão compilados para o *bytecode* da JVM e em seguida traduzidos ou para a execução na DVM, ou para código nativo e execução no Android Runtime. Outra alternativa é o uso das bibliotecas do Android como por exemplo *AsyncTask*, *HandlerThread*, *IntentService*, entre outros. É objetivo desse trabalho realizar uma avaliação

de alguns desses mecanismos buscando determinar como eles podem ser comparados e em quais contextos são mais eficientes.

O modelo de concorrência presente no Android possui algumas limitações causadas pela existência da thread principal, uma vez que essa thread suporta todo o processamento da UI da aplicação. Muitos métodos da interface de usuário verificam se estão executando na thread principal e caso contrário lançam uma exceção. Isso significa que qualquer fluxo de execução concorrente a essa thread, que tenha sido iniciado pela aplicação, não pode, diretamente, realizar atualizações na UI. Esse modelo foi adotado pelo SO Android porque é difícil construir uma interface de usuário multithread que não entre em deadlock (MEIKE, 2016). A documentação Android afirma que o Android UI *toolkit*, utilizado para realizar modificações na interface de usuário, não é *thread-safe*, ou seja, não existiria garantia de que as threads se comportariam de acordo com suas especificações se pudessem manipular os dados da interface como compartilhados. Outras razões que podem ser citadas para a escolha de manter a UI como responsabilidade de apenas uma thread é a simplicidade, pois nenhuma solução de sincronização precisa ser introduzida, e existe atomicidade entre operações que podem afetar os mesmos objetos da UI.

Essa limitação da concorrência imposta pelo Android obriga que qualquer *worker* thread que precise atualizar a UI, realize requisições explícitas para que suas atualizações sejam executadas pela thread principal. Desse modo, a concorrência no Android é útil para desafogar a thread principal de tarefas que não envolvem atualização da interface do usuário, ou que ficam um longo tempo executando até que seu resultado fique pronto para atualizar a UI.

2.5 Mecanismos de Concorrência

Essa seção tem objetivo de apresentar os mecanismos de concorrência que serão avaliados nesse trabalho.

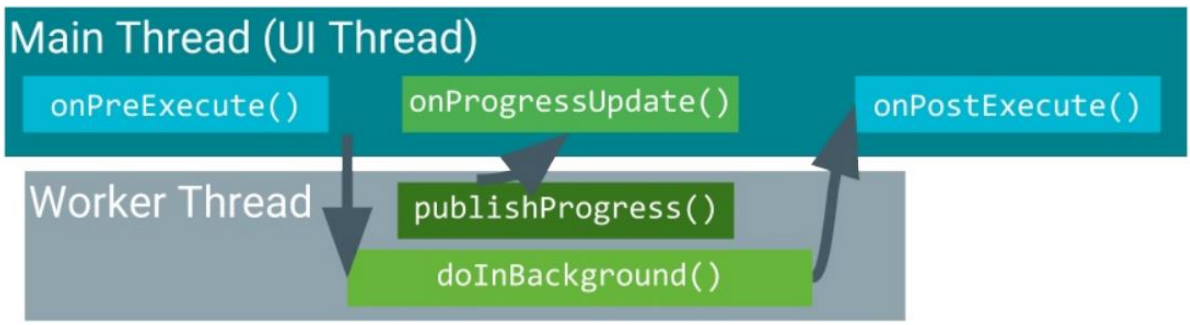
2.5.1 AsyncTask

AsyncTask é o primeiro mecanismo de concorrência para Android que a maioria dos desenvolvedores utiliza (MEIKE, 2016). A classe AsyncTask deve ser usada para implementar tarefas assíncronas que executarão em uma *worker* thread, essa classe permite que se realize operações em *background* e que se publique os resultados na UI thread sem que o programador precise manipular diretamente threads ou handlers (GOOGLE, 2016).

A classe AsyncTask possui quatro métodos principais que podem ser sobrescritos por qualquer classe que estenda a classe AsyncTask. Esses métodos seguem o modelo apresentado na Figura 2.5 e realizam as seguintes tarefas:

1. *onPreExecute()*: chamado na UI thread antes da tarefa ser executada, pode ser usado para configurar a tarefa, como por exemplo, iniciar uma barra de progresso na UI.
2. *doInBackground(Params ...)*: invocado na *background* thread após a execução de *onPreExecute()*, esse método executa a operação de *background* de fato e retorna um resultado que é passado para *onPostExecute()*. Também pode realizar chamadas da função *publishProgress(Progress ...)* usada para avisar a thread principal do progresso atual.
3. *onProgressUpdate()*: executa na UI thread após chamada da função *publishProgress(Progress ...)*, usada para reportar progresso para a thread principal enquanto as operações em *background* estão sendo executadas.
4. *onPostExecute(Result)* executa na UI thread após a computação em *background* ter finalizado.

Figura 2.5: Modelo de execução de uma AsyncTask.



(a) Imagem retirada de (GOOGLE, 2016).

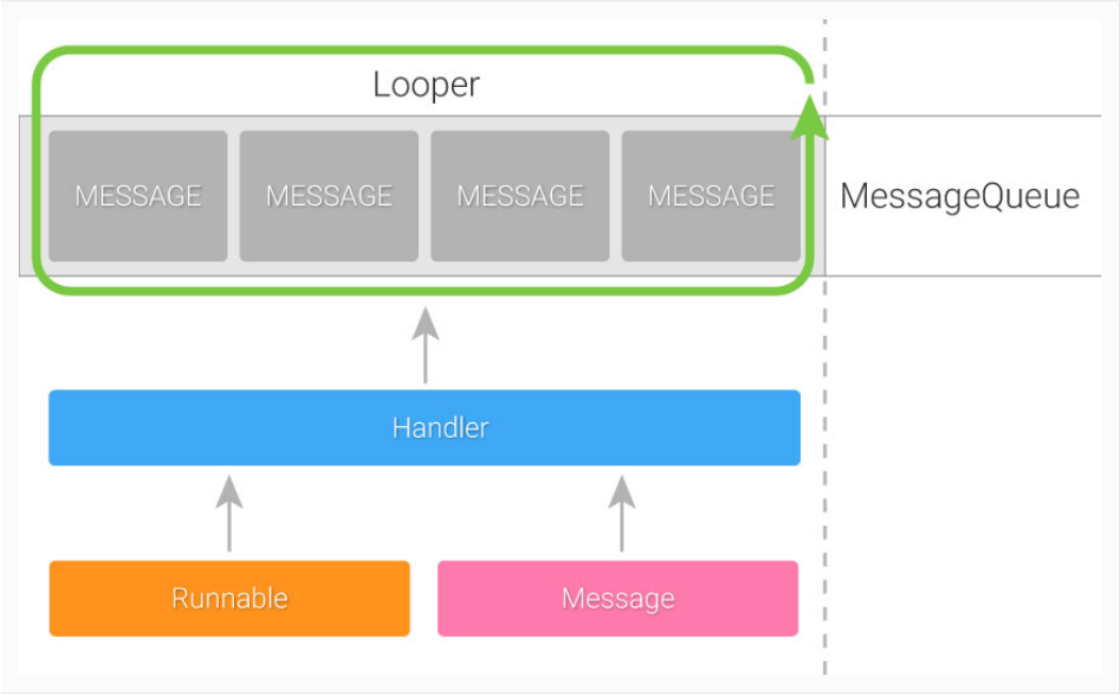
O objetivo da classe AsyncTask é servir como uma classe auxiliar às classes Thread e Handler permitindo o uso fácil e adequado da UI thread estabelecendo fluxos concorrentes de modo simples. Contudo não constitui um *framework* genérico para geração de fluxos concorrentes, uma vez que quando uma AsyncTask é invocada, usando o método *execute(Params... params)*, ela é enfileirada em uma única thread de *background*. O principal motivo para que uma AsyncTask seja implementada dessa forma é fornecer um método simples de desafogar a UI Thread sem que o programador precise se preocupar com os problemas que

podem surgir em decorrência da execução das suas AsyncTasks de forma concorrente entre si (GOOGLE, 2019a).

2.5.2 HaMeR framework

O nome HaMeR vem de três estruturas importantes para a expressão da concorrência nesse mecanismo (*Handler, Message* e *Runnable*). Muito embora o nome do *framework* mencione apenas essas três estruturas, existem outras classes envolvidas como a classe *Looper* e a classe *MessageQueue*. Apesar de AsyncTask ser a primeira construção de concorrência que a maioria dos desenvolvedores encontra, a construção de concorrência básica do Android é baseada nas classes *Looper* e *Handler* (MEIKE, 2016). O modelo de execução desse *framework* é ilustrado na Figura 2.6.

Figura 2.6: Modelo de execução do HaMeR framework.



(a) Imagem retirada de (MEGALI, 2016).

O Looper é um mecanismo cujo objetivo é transferir uma tarefa de uma thread para outra, funciona como uma fila de eventos que está vinculada a uma única *worker* thread. Para realizar esse processo o Looper de uma *worker* thread é associado a uma instância da classe *MessageQueue* que armazena uma lista de mensagens a serem entregues pelo Looper para que sejam executadas.

Para que a thread que originou a tarefa possa usar esses mecanismos, ela precisa criar um objeto da classe *Handler* para o Looper desejado. Esse Handler pode ser usado para enfileirar tantas tarefas quanto forem necessárias nesse Looper e pode fazer isso de duas formas básicas:

1. Por meio de objetos da classe *Message* que entrarão na fila do Looper e serão processados. Esse processamento pode variar dependendo de como a resposta às mensagens está definida no Handler.
2. Por meio de implementações da interface *Runnable* que entrarão na fila e serão normalmente executados pela *worker* thread quando chegar a sua vez na fila do Looper.

Se um Handler é criado sem a especificação explícita de um Looper como argumento, esse Handler vai usar o Looper da thread em execução, o que de fato não acarreta na criação de um fluxo concorrente. Para que a concorrência ocorra de fato, é necessário criar um novo Looper e associá-lo a uma nova *worker* thread. Um jeito simples de realizar isso é criando uma instância da classe *HandlerThread* que inicia uma *worker* thread nova com um Looper e um Handler prontos para receber requisições.

2.5.3 IntentService

Para entender o uso de IntentServices como um mecanismo de concorrência, é necessário compreender alguns outros conceitos do Android como *Services* e *Intents*.

Um objeto do tipo *Service* pode ser entendido como uma *Activity* sem UI (MEIKE, 2016). E, como qualquer outro componente, deve ser descrito no manifesto da aplicação. Um objeto do tipo *Service* pode ser configurado para receber solicitações de outras aplicações, contudo, como os outros componentes da aplicação, existe no máximo uma instância de um Service a cada momento, tal instância é criada com a primeira requisição para esse serviço e é destruída quando a última entidade requisitante não precisa mais do *Service*. Assim como uma Activity, um *Service* possui seu ciclo de execução baseado em call-backs, que podem ser sobrescritas e são chamadas quando o serviço é criado, quando recebe solicitação, quando é destruído, entre outros.

Dentre as principais razões para se usar *Service* está o fato de que a prioridade do processo que executa um *Service* é maior do que a prioridade de um processo executando uma Activity não visível, embora essa prioridade seja menor do que uma Activity visível (MEIKE, 2016).

Um objeto da classe *Intent*, por sua vez, é usado para descrever uma tarefa dentro da aplicação, faz isso armazenando informações que serão passadas de um componente para outro funcionando de forma similar a uma mensagem passada entre os processos que executam cada um dos componentes (DIMARZIO, 2008).

De acordo com Meike (2016), um IntentService é provavelmente a forma mais comum de se usar um *Service*, pois é uma extensão simples, atrativa e elegante da classe *Service* e fornece as funcionalidades esperadas pelos desenvolvedores quando usam AsyncTask. Um *Service*, por si só, não oferece concorrência à aplicação pois seu código executa sobre a UI thread. Contudo, ao se usar um IntentService, as tarefas são executadas no método *onHandleIntent*, que executa em uma *worker* thread. IntentService usa um Looper para o enfileiramento das tarefas que chegam a essa thread.

2.5.4 Threads

A classe *Thread* de Java serve como base para o sistema multithread da linguagem e, juntamente com implementações da interface *Runnable*, representa o básico da programação concorrente em Java. Uma Thread de Java encapsula uma thread em execução (SCHILDT, 2014). Dentre os métodos da classe Thread destacam-se:

1. *start()*: Inicia a execução da Thread chamando seu método *run()*.
2. *run()*: Código que será executado pela Thread.
3. *sleep()*: Suspende a execução da Thread por um período de tempo.
4. *join()*: Espera até que a Thread termine.

Uma Thread pode ser criada de duas formas:

1. Implementando a interface *Runnable*. Uma classe *A* implementa a interface *Runnable* e um objeto da classe *A* é passado como argumento para a criação de um objeto *Thread* que vai executar a tarefa definida no método *run* da classe *A*.
2. Estendendo a classe *Thread.* Uma classe *B* estende a classe *Thread* e em uma instância da classe *B* o método *start* pode ser chamado para a execução do método *run* da classe *B*.

2.5.5 Thread Pool

O uso de thread pool em Java permite a execução de tarefas sobre um conjunto de threads que usa uma fila para armazenar as tarefas que precisam ser executadas

(HORSTMANN, 2013). A classe *ThreadPoolExecutor* fornece a implementação básica para a execução de thread pools, onde as tarefas são submetidas por meio da função *execute* que recebe como parâmetro a tarefa descrita como um objeto *Runnable* (GÖETZ, 2006).

Um objeto da classe *ThreadPoolExecutor* pode ser criado de várias formas, dentre elas destacam-se os métodos:

1. *newCachedThreadPool*: Cria e destrói threads dinamicamente de acordo com a demanda de tarefas existente.
2. *newFixedThreadPool*: Recebe como argumento a quantidade de threads que devem ser criadas para a implementação do thread pool e mantem essa quantidade de threads constante, ou seja, a quantidade de threads independe da quantidade de tarefas existente nas filas de execução, isso introduz menor sobrecarga para o gerenciamento de threads. Para os experimentos realizados nesse trabalho essa foi a opção utilizada.

2.5.6 Kotlin coroutines

Coroutines são implementadas em Kotlin na biblioteca *kotlinx.coroutines* e podem ser vistas como threads leves, no sentido de consumo de memória. Executam um bloco de código e podem completar retornando um valor ou uma exceção. Coroutines são instâncias de uma computação que pode ser suspensa e não estão ligadas a uma thread específica, assim, podem ser suspensas em uma thread e voltar a executar em qualquer outra thread que estiver disponível (ARIAS, 2018). Para a execução de coroutines são definidos os seguintes termos:

1. *runBlocking*: função que cria uma coroutines e bloqueia a thread atual até que essa coroutine termine de executar, podendo retornar um valor correspondente à execução.
2. *launch*: função que cria uma nova coroutines sem bloquear a thread atual e retorna uma instância de *Job* que pode ser posteriormente usada para obter o valor resultante da coroutine.
3. *delay*: função que suspende a coroutine atual por um determinado tempo sem bloquear a thread atual.
4. *suspend*: modificador de função que identifica funções definidas pelo programador que podem suspender a execução das coroutines.

2.6 Mecanismos de Sincronização

Essa seção tem objetivo de apresentar os mecanismos de sincronização usados durante a implementação dos testes e avaliados nesse trabalho.

2.6.1 Lock e Condition

Uma alternativa bastante usada para a sincronização de fluxos concorrentes é o uso de implementações de locks existentes no pacote *java.util.concurrent.locks*. Dentre as implementações presentes nesse pacote uma escolha comum é a classe *ReentrantLock*. Um lock reentrante permite que o dono atual do lock requisite esse lock novamente sem causar um deadlock. Isso simplifica o código em caso de recursão e em passagens do lock para outras funções. Outro ponto positivo dessa classe é a existência do método *tryLock()* que apenas adquire o lock se ele está disponível, se não retorna imediatamente indicando que não pode obter o lock (SAMUEL, 2017), isso evita o bloqueio do fluxo quando ele pode executar outras tarefas se não conseguir obter o lock.

A interface *Lock* prevê a existência do método *newCondition()* nas implementações de lock, esse método permite que se obtenha um objeto do tipo *Condition* que permite controle detalhado do lock usando sinalização com os métodos *await()* e *signal()* que operam sobre objetos Condition (SCHILDT, 2014).

2.6.2 Semáforos

Semáforos são usados para controlar a quantidade de tarefas que podem acessar certo recurso ou realizar determinada ação em algum momento. Podem ser usados para implementar um grupo limitado de recursos ou para impor limites a uma coleção (GÖETZ, 2006). Dessa forma, um semáforo é um mecanismo que mantém um inteiro que indica a quantidade de recursos disponíveis e permite que esse inteiro seja acessado e modificado, em exclusão mútua, por um conjunto de threads (SAMUEL, 2017).

Em Java, semáforos são implementados na classe *java.util.concurrent.Semaphore* e a quantidade de recursos disponíveis é chamada de *permits*. Logo, para acessar um recurso, uma thread precisa conseguir um *permit* do semáforo (SCHILDT, 2014). Os métodos usados pelas threads para interagir com o semáforo são:

1. *acquire()*: Usado para obter um ou mais *permits*.

2. *release()*: Usado para liberar o uso de um ou mais *permits* que estavam sendo mantidos pela thread que realizou a chamada do método.
3. *tryAquire()*: Semelhante ao *tryLock()* para locks, ou seja, tenta obter *permits* que estão imediatamente disponíveis.

2.6.3 Synchronized Kotlin

A palavra-chave *synchronized*, de Java, pode ser aplicada como modificador de método ou como um bloco de código e garante que somente uma thread poderá executar esse código por vez, ou seja, a execução desse código ocorre em exclusão mútua garantindo consistência às variáveis que podem ser escritas por diversas threads (BLOCH, 2018).

Em Kotlin, *synchronized* é implementada como uma função *inline* que recebe um objeto para ser usado como lock para garantir acesso exclusivo ao código implementado dentro do synchronized.

Na JVM, todas as instâncias de objetos possuem um monitor associado, esse monitor somente pode pertencer a uma thread a cada momento, contudo qualquer thread pode requisitar o monitor de um objeto que resultará ou na entrega do monitor da instância para a thread ou no bloqueio da thread até que possam obter esse monitor. Para requisitar o monitor é possível utilizar *synchronized* como uma função passando como argumento o objeto do qual se deseja obter o monitor (SAMUEL, 2017).

Desse modo a palavra-chave *synchronized* pode ser usada das seguintes formas em Java:

1. **Modificador de método de instância**: impõe exclusão mútua na execução do método para esse objeto ou da própria classe, se o método for estático.
2. **Bloco de comandos**: nesse caso obtém o monitor do objeto, ao qual o bloco pertence.
3. **Função para obter acesso ao monitor de uma instância de objeto**: esse uso é bastante semelhante ao bloco de comandos, a diferença é que após a palavra-chave *synchronized* se coloca como parâmetro o objeto do qual se deseja obter acesso ao monitor. Esse uso também está presente em Kotlin, que define *synchronized* como uma função implementada na sua biblioteca padrão.

2.6.4 Variáveis atômicas

O pacote *java.util.concurrent.atomic* oferece classes que representam tipos primitivos de dados com objetivo de proporcionar métodos de leitura e de escrita atômicos sobre essas variáveis. Esses métodos buscam ser meios eficientes de atualizar o valor de variáveis sem o uso de ferramentas de sincronização que possam bloquear a execução de um fluxo (SCHILDT, 2014).

Exemplos de classes presentes no pacote *java.util.concurrent.atomic* são:

- *AtomicInteger*;
- *AtomicLong*;
- *AtomicBoolean*;
- *AtomicReference*.

3 METODOLOGIA DE AVALIAÇÃO

Esse capítulo introduz alguns aspectos da metodologia de avaliação utilizada para a realização desse estudo. Em seguida, na seção 3.1, os problemas usados para avaliação são apresentados considerando seus aspectos teóricos no sentido de como podem ser usados para avaliar mecanismos de concorrência e de sincronização. Além disso, sua implementação realizada para esse trabalho e o projeto dos experimentos são discutidos nessa seção. Na seção 3.2, as métricas para avaliação são explicadas. Na seção 3.3, são explicados os testes de hipóteses estatísticas utilizados durante esse trabalho.

Para atingir o objetivo de avaliar os mecanismos de concorrência e de sincronização que foram apresentados nas seções 2.5 e 2.6, foi montada uma metodologia de avaliação que leva em consideração alguns passos para sistematização de um projeto de avaliação de desempenho apresentados por Jain (1991). São eles:

1. **Objetivos do estudo e definição do sistema**: Como descrito na seção 1.1, o objetivo desse trabalho é realizar uma avaliação comparativa de alguns mecanismos de concorrência e de sincronização disponíveis para o desenvolvimento de aplicações Android considerando:
   - Escalabilidade frente a recursos paralelos de hardware;
   - Sobrecarga introduzida para gerenciamento de fluxos concorrentes;
   - Justiça no compartilhamento de recursos de software;
   - Eficiência em ambientes cooperativos que exigem sincronização de grupos de tarefas diferentes.
2. **Serviços do sistema e possíveis saídas**: Os sistemas avaliados nesse trabalho são os mecanismos de concorrência e de sincronização, seus serviços, ou seja, o modo como permitem a execução de fluxos concorrentes ou como executam a sincronização, foram descritos nas seções 2.5 e 2.6.
3. **Métricas de desempenho para avaliação**: As métricas de avaliação usadas nesse trabalho são as seguintes:
   - Tempo de execução;
   - Speedup;
   - *Throughput*;
   - Justiça.

   Essas métricas são descritas com mais detalhes na seção 3.2.

4. **Parâmetros do sistema e dos testes**: Os parâmetros que afetam o desempenho do sistema são:
    - Quantidade de CPUs no dispositivo.
    - Versão do Android em execução.
    - Latência de rede e QoS das conexões de Internet.

    Com objetivo de tornar as observações realizadas gerais, ou seja, independente dos parâmetros do sistema, foram realizadas variações no hardware por meio de execução remotas no Firebase Test Lab descritas no Apêndice A. Os parâmetros dos testes são específicos para cada problema e são descritos na seção 3.1, alguns desses parâmetros, no entanto, estão na maioria dos problemas testados, como a variação dos mecanismos avaliados e a variação da quantidade de tarefas concorrentes.
5. **Fatores de avaliação**: Para cada um dos problemas é definida uma lista de fatores para avaliação dos mecanismos usados durante as experimentações, que estão detalhados, também, na seção 3.1.
6. **Técnicas de avaliação**: Como técnica de avaliação serão realizadas medições do sistema real (mecanismos de concorrência e de sincronização). Essas medições são apresentadas no capítulo 4.
7. **Programas de teste**: Descritos na seção 3.1.
8. **Projeto dos experimentos**: Realizado na seção 3.1 no contexto da descrição dos problemas.
9. **Análise e interpretação dos dados**: Realizada no capítulo 4.
10. **Apresentação dos resultados**: Realizada no capítulo 4.

Os experimentos realizados nesse trabalho são do tipo fatorial completo, com quantidade de fatores variando entre 2 e 4 dependendo do problema utilizado para avaliação dos mecanismos de concorrência ou de sincronização. Ao todo, foram executados 216 testes diferentes, cada um deles foi repetido pelo menos 30 vezes para aumentar a significância estatística dos resultados.

3.1 Programas de teste

Como os usos reais dos mecanismos de concorrência e de sincronização são bastante variados, os cinco problemas escolhidos para avaliação buscam explorar diferentes aspectos de

cada mecanismo, tentando cobrir o máximo possível de usos e mantendo a simplicidade. Desse modo tenta não realizar suposições sobre o que será executado nos mecanismos em um cenário real, mas busca avaliar características que seriam importantes em alguma medida para qualquer aplicação que envolva o uso de tarefas concorrentes. Embora as aplicações dos sistemas avaliados possam ser mais complexas, a escolha das aplicações usadas neste trabalho objetiva também a redução de parâmetros específicos de um problema para que os parâmetros relativos aos mecanismos de concorrência e de sincronização se sobressaiam nos resultados da experimentação. Por exemplo, se fossem escolhidos problemas como treinamento de redes neurais, ou processamento em Big Data, diversos outros parâmetros que afetam diretamente o desempenho dos problemas, mas não diretamente a concorrência, deveriam ser levados em consideração, como a taxa de aprendizado para redes neurais, o tamanho do conjunto de dados avaliado, entre outros. Alternativamente seria possível manter os parâmetros do problema fixos durante a avaliação. Essa estratégia, contudo, não foi adotada, pois poderia levar a vantagens de algum dos mecanismos que porventura seja melhor em algum conjunto de parâmetros limitado de um problema, mesmo se esse não for o caso com outro conjunto de parâmetros. Dessa forma, foram escolhidas as aplicações descritas a seguir.

3.1.1 Multiplicação de Matrizes

A multiplicação de matrizes consiste em uma operação da álgebra linear que recebe um conjunto de matrizes de entrada com dimensões compatíveis e, no caso da entrada ser apenas duas matrizes, gera como saída uma matriz cujos elementos são a soma dos produtos dos elementos de uma linha da primeira matriz pelos elementos de uma coluna da segunda matriz. Esse processo é ilustrado na Figura 3.1.

Figura 3.1: Exemplo de multiplicação de matrizes quadradas.

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} \begin{vmatrix} j & k & l \\ m & n & o \\ p & q & r \end{vmatrix} = \begin{vmatrix} aj+bm+cp & ak+bn+cq & al+bo+cr \\ dj+em+fp & dk+en+fq & dl+eo+fr \\ gj+hm+ip & gk+hn+iq & gl+ho+ir \end{vmatrix}$$

No âmbito da programação concorrente, o problema da multiplicação de matrizes é altamente paralelizável, uma vez que os valores da matriz resultante são independentes entre si. Existem diferentes modos de paralelizar a operação da multiplicação de matrizes dependendo

de como as tarefas paralelas são definidas. Uma das formas de paralelizar a multiplicação de duas matrizes quadradas de dimensões $n \times n$ é utilizando n^3 tarefas, de modo que cada tarefa executa apenas uma multiplicação e, em seguida, n^2 tarefas realizam a soma desses produtos e armazenam o resultado na matriz resultante. Outro modo é utilizando n^2 tarefas, onde cada uma é responsável por realizar todas as multiplicações e somas necessárias para um elemento da matriz resultante. É possível também usar apenas n tarefas para a paralelização, sendo que cada tarefa calcula os elementos de uma linha ou coluna completa da matriz resultante, essa foi a opção usada para a implementação do problema nesse trabalho. A quantidade de tarefas que se deseja usar pode ser menor de n, nesses casos, basta dividir as tarefas como definidas anteriormente entre a quantidade de tarefas disponíveis para a execução.

No contexto da avaliação desse trabalho, a multiplicação de duas matrizes pode ser utilizada para avaliar a escalabilidade dos mecanismos de concorrência quanto aos recursos paralelos do hardware, visto que o problema pode ser quebrado em problemas menores que são independentes entre si e podem ser executados de forma paralela. Uma vez que os problemas são independentes, podem ser implementados sem que exista a necessidade de sincronização, apenas fazendo que a memória correspondente a matriz resultante não seja a mesma memória das matrizes de entrada. Nesse caso, os problemas de condição de corrida, deadlock ou starvation não ocorrem.

Para a implementação do problema da multiplicação de matrizes, optou-se pelo uso de duas matrizes quadradas de mesmo tamanho para a multiplicação. Essa simplificação permite que se reduza a quantidade de parâmetros que afetam os testes, como por exemplo, quantidade de matrizes e ordem em que a multiplicação é realizada. Essa redução de parâmetros, por sua vez, permite que a análise se foque em parâmetros relacionados mais diretamente com o comportamento geral dos mecanismos de concorrência como a quantidade de fluxos concorrentes criados para a execução.

O algoritmo usado para a implementação da multiplicação das matrizes divide a quantidade de linhas da primeira matriz igualmente, ou com diferença de no máximo uma linha, entre as tarefas concorrentes. Cada uma dessas tarefas realiza a multiplicação dos elementos das linhas que recebe com todas as colunas da segunda matriz, depois dessas multiplicações realiza a soma dos produtos dos elementos de uma linha com os elementos de uma coluna para cada um dos elementos das linhas que recebeu obtendo os elementos correspondentes da matriz resultante. Portanto, cada tarefa é responsável pelo cálculo completo de um conjunto de linhas da matriz resultante. Esse algoritmo, executado sequencialmente, tem complexidade $O(n^3)$, com

n sendo a quantidade de linhas das matrizes quadradas. Novamente, optou-se pelo uso desse algoritmo ao invés do uso de outros algoritmos com complexidade assintótica menor, como o algoritmo de Strassen, por exemplo, pois outros fatores como a instabilidade numérica estariam envolvidos (GOLUB, 2013). Esses fatores adicionariam parâmetros que afetam os testes, mas que não estão relacionados a avaliação desse trabalho. Como o maior uso da pilha causado pela natureza recursiva baseada em divisão e conquista do algoritmo de Strassen e o fato de que a paralelização desse algoritmo é mais complicada pois exige comunicação entre os fluxos concorrentes o que levaria a um envolvimento maior dos mecanismos de sincronização.

A implementação desse problema, para as avaliações desse trabalho, considera apenas variações dos mecanismos de concorrência e não avalia os mecanismos de sincronização. O problema foi implementado usando os seguintes mecanismos de concorrência:

- Framework HaMeR;
- Threads;
- Thread Pool;
- Kotlin coroutines.

Os mecanismos AsyncTask e IntentServices não foram utilizados pois, como explicado na seção 2.5, as tarefas criadas por esses métodos não executam sobre a UI thread, no entanto, executam apenas sobre uma *worker* thread. Assim esses dois mecanismos têm objetivo de desafogar a UI thread e não de utilizar de modo mais eficiente recursos paralelos do hardware gerando múltiplas worker threads que é o que está sendo avaliado nesse problema.

Os parâmetros que afetam os testes podem ser resumidos em:

- Tamanho das matrizes usadas para a multiplicação;
- Quantidade de tarefas criadas para realizar a multiplicação de forma concorrente;
- Mecanismo de concorrência utilizado.

Todos esses parâmetros foram usados como fatores de avaliação para os experimentos realizados com multiplicação de matrizes.

Esses experimentos consideram as seguintes variações dos fatores de avaliação:

- Tamanho das matrizes: 128x128, 256x256 e 512x512;
- Tarefas concorrentes: 1, 2, 8 e 64;

- Mecanismos de concorrência: Thread, Thread Pool, framework HaMeR e Kotlin coroutines.

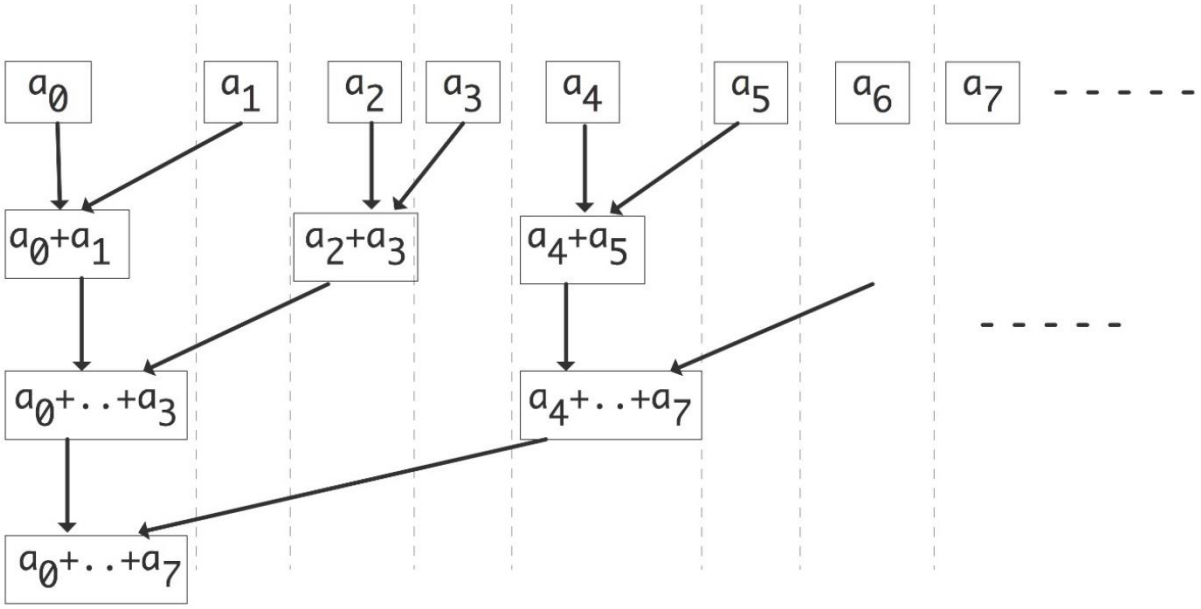
Utilizando essas variações dos fatores de avaliação foram realizados um total de 48 testes para a implementação do problema de multiplicação de matrizes. Esses valores foram utilizados para avaliar a redução no tempo de execução de cada um dos mecanismos com o aumento da quantidade de tarefas concorrentes, variando também o tamanho da matriz para explorar diferentes casos do problema.

Foram utilizados tamanhos de matrizes em potências de dois, pois, nesse caso, cada tarefa iniciada pelo mecanismo de concorrência, que também estão em potências de dois, recebe a mesma quantidade de linhas para realizar o processamento da multiplicação de matrizes. Caso a quantidade de linhas das matrizes não fosse divisível pelo número de tarefas gerado a carga de tarefas estaria desbalanceada com algumas tarefas sendo responsáveis pelo processamento de mais linhas do que outras. Nesses casos as tarefas com mais linhas tenderiam a possuir um tempo de execução maior e a determinar o tempo de execução do problema como um todo.

3.1.2 Soma concorrente

A soma sequencial de valores inteiros é um problema que consiste em iterar por um vetor usando uma variável acumuladora para somar o valor de cada um dos seus inteiros. A definição desse problema de forma paralela é um pouco mais complexa. A soma de um conjunto de valores em paralelo define tarefas que executam apenas uma soma de dois elementos, essas tarefas são dependentes entre si, de modo que o processamento ocorre em níveis de tarefas, onde em cada nível a metade das tarefas do nível anterior é necessária. Esse processo é ilustrado na Figura 3.2.

Figura 3.2: Ilustração da execução do problema da soma paralela de elementos.



(a) Imagem retirada de (EIJKHOUT, 2014).

O problema da soma de elementos organizado de forma paralela apresenta dependência de dados entre as tarefas de um nível em relação àquelas do nível anterior, uma vez que as operações do nível corrente dependem do resultado das operações de níveis anteriores. Essa dependência de dados exige que exista algum mecanismo de sincronização entre a execução das tarefas para que o resultado da soma seja correto, ou que as tarefas estejam confinadas a um nível de execução e a cada novo nível novas tarefas concorrentes sejam iniciadas. A paralelização ocorre definindo as tarefas em relação a seu nível de execução, de modo que, no primeiro nível cada tarefa realiza a soma de dois dos elementos de entrada, nos níveis seguintes cada tarefa realiza a soma dos resultados de duas tarefas do passo anterior até que, no último nível, sobrem apenas dois valores que serão somados por uma tarefa resultando na soma de todos os elementos iniciais.

Embora essa definição gere um algoritmo de complexidade *O(log n)*, se existem *n/2* CPUs, a execução do algoritmo paralelo para a soma de um conjunto de elementos geralmente é mais lenta do que a execução sequencial, uma vez que a sobrecarga introduzida pelo gerenciamento entre as tarefas e pela a sincronização é alta e, dificilmente, o hardware vai possuir *n/2* CPUs disponíveis. Desse modo, a definição do problema de forma paralela possui poucas aplicações práticas, contudo, suas aplicações para as avaliações realizadas nesse trabalho são válidas pois permite avaliar como os mecanismos de concorrência lidam com

muitas tarefas que possuem dependências entre si e como a introdução de sincronização a cada nível pode afetar o desempenho da execução do algoritmo.

O principal objetivo dos experimentos com o problema da soma concorrente é determinar o quão bem os mecanismos de concorrência lidam com a existência de muitas tarefas simples. Muito embora as tarefas desse problema possuem dependência de dados entre si, o problema pode ser resolvido controlando quando uma tarefa deve ser iniciada. Desse modo, se a implementação usa tarefas que operam somente em um nível da soma basta iniciar todas as tarefas do nível quando o nível anterior terminou e iniciar as tarefas do nível seguinte apenas quando as tarefas do nível atual terminarem.

Em função disso, boa parte das implementações dos experimentos desse problema utilizam a definição de tarefas que estão vinculadas a um nível da execução do algoritmo. Com objetivo de comparar qual seria a diferença de se usar tarefas que operam em todos os níveis de forma sincronizada ao invés de usar tarefas que operam somente em um nível, foi realizada uma implementação utilizando Threads com o uso de barreiras para a sincronização a cada nível, ou seja, são iniciadas tarefas no primeiro nível que serão mantidas nos níveis seguintes, sendo sincronizadas com o uso de barreiras ao final de cada nível.

Desse modo os mecanismos utilizados para implementação foram:

- Threads;
- Thread Pool;
- Framework HaMeR;
- Kotlin coroutines;
- Threads com barreiras.

Os parâmetros que afetam os testes para o problema da soma concorrente são:

- Quantidade de números para somar;
- Tarefas concorrentes em cada nível;
- Mecanismos de concorrência entre as tarefas;
- Mecanismos de sincronização utilizados quando as tarefas são mantidas em múltiplos níveis.

Vale notar que a quantidade de tarefas concorrentes a cada nível não corresponde a quantidade de tarefas totais utilizadas durante a execução do algoritmo, mas a quantidade máxima utilizada em um mesmo nível. Por exemplo, se forem somados 16 valores e usar 4

tarefas concorrentes por nível o total de tarefas utilizadas, nos casos em que as tarefas somente executam em um nível, será de 13 tarefas (4 no primeiro nível + 4 no segundo nível + 2 no terceiro nível + 1 no último nível).

Os fatores de avaliação utilizados para a soma concorrente consideram todos os parâmetros que afetam os testes descritos anteriormente, com exceção do mecanismo de sincronização utilizado. Optou-se por não variar o mecanismo de sincronização utilizado quando as tarefas são mantidas em múltiplos níveis pois a forma como o algoritmo é concebido em níveis de execução é o que se espera para o uso de barreiras como mecanismo de sincronização.

Os experimentos realizados para o problema da soma concorrente realizaram as seguintes variações dos fatores de avaliação:

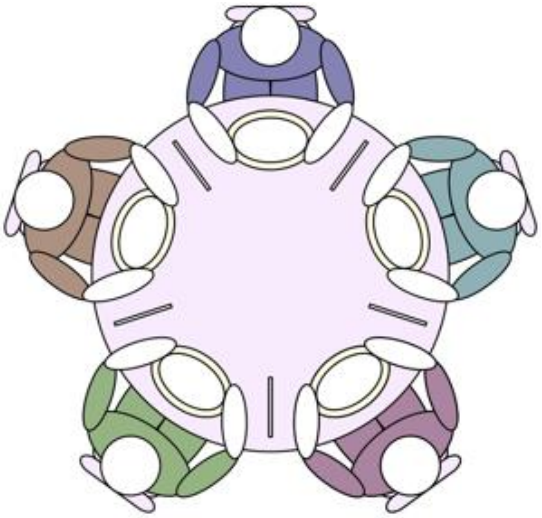
- Quantidade de números para somar: 262144 (2^{18}) e 1048576 (2^{20});
- Tarefas concorrentes: 1, 16 e 256;
- Mecanismos de concorrência: Thread, Thread Pool, framework HaMeR e Kotlin coroutines e Threads usando barreiras como sincronização entre tarefas que executam por diferentes níveis.

Utilizando essas variações dos fatores de avaliação, foram realizados um total de 30 testes para a implementação do problema da soma concorrente.

3.1.3 Problema dos filósofos (dining philosophers)

O problema dos filósofos é considerado um problema clássico de sincronização pois é um exemplo de uma grande classe de problemas de controle de concorrência, sendo uma representação simples da necessidade da alocação de recursos entre diferentes processos de uma forma que previna deadlock e starvation (SILBERSCHATZ, 2013). O problema consiste em *n* filósofos sentados ao redor de uma mesa com *n* garfos dispostos entre si, de tempos em tempo, um filósofo deseja comer, mas para comer precisa pegar o garfo que está a sua direita e o garfo a sua esquerda. Um filósofo pode pegar um garfo de cada vez, mas precisa dos dois para comer e não pode pegar um garfo que está com outro filósofo. Quando consegue os dois garfos que precisa, um filósofo pode comer e ao terminar coloca os dois garfos nas suas posições originais. Um cenário do problema com cinco filósofos é mostrado na Figura 3.3.

Figura 3.3: Exemplo do cenário do problema com 5 filósofos.



(a) Imagem retirada de (YOUHAN, 2019).

Existem várias formas de representar esse problema na computação, por exemplo, pode-se representar cada garfo como um semáforo e fazer cada filósofo ser uma thread que requer dois semáforos em específico para realizar uma tarefa. Desse modo o problema envolve tanto concorrência, pois o processamento de cada filósofo não depende do processamento dos outros (somente depende dos garfos), quanto sincronização no acesso aos recursos compartilhados.

Para a avaliação realizada nesse trabalho, o problema dos filósofos é usado para a avaliação dos mecanismos de concorrência e de sincronização para avaliar tanto o desempenho dos processos (filósofos), quanto a justiça no compartilhamento de recursos (garfos) entre esses processos.

O problema dos filósofos é tipicamente apresentado contendo cinco filósofos, contudo, para atender melhor o objetivo de avaliar a justiça dos mecanismos de concorrência e de sincronização, para os experimentos desse trabalho, o problema foi implementado com um valor arbitrário de filósofos e a quantidade de garfos é sempre igual a quantidade de filósofos.

Para a implementação dos experimentos com o problema dos filósofos, realizou-se uma adaptação no problema clássico. Nessa adaptação os recursos são representados por dados do tipo *String* e os filósofos são tarefas que, quando conseguem obter suas duas Strings, executam o algoritmo LCS (*Longest Common Subsequence*) entre as Strings, esse algoritmo recebe duas Strings e calcula qual a maior substring comum entre elas. Apesar do algoritmo em si não possuir implicações práticas na avaliação dos mecanismos de concorrência e de sincronização que se deseja executar nesse trabalho, ele permite a simulação da execução de uma tarefa pelos processos que estão competindo pelos recursos.

Na implementação realizada, todas as tarefas estão configuradas para executar durante um tempo fixo a maior quantidade possível de vezes o algoritmo do LCS. A avaliação da justiça do mecanismo é realizada por meio de uma contagem de quantas vezes cada tarefa conseguiu executar o algoritmo LCS, utilizando medidas como o coeficiente de variação, média aritmética e valores máximos e mínimos.

Dentre os mecanismos de sincronização utilizados estão semáforos e locks, para ambos foi usado uma implementação de fila justa, ou seja, as tarefas que esperam para a obtenção dos recursos no semáforo ou no lock vão ser atendidas na ordem que chegaram. Essa decisão foi tomada, para buscar tornar as execuções o mais justas possível quanto à distribuição dos recursos em cada mecanismo, já que esse é um dos fatores que se deseja avaliar com esse problema.

Os parâmetros que afetam os experimentos com o problema dos filósofos conforme a adaptação realizada para esse trabalho são:

- Quantidade de tarefas (filósofos);
- Mecanismo de sincronização para acesso aos recursos;
- Mecanismo de concorrência entre as tarefas;
- Tempo de execução das tarefas;
- Tamanho das Strings usadas no algoritmo LCS.

Os fatores de avaliação utilizados para a implementação do problema dos filósofos consideraram a quantidade de filósofos, o mecanismo de sincronização e o mecanismo de concorrência. Optou-se por não realizar variações no tempo de execução das tarefas nem no tamanho das Strings utilizadas no LCS, pois adicionariam um alto custo a avaliação além de não representarem implicações tão importantes para os experimentos como a quantidade tarefas por exemplo. Como o algoritmo LCS executa apenas sobre uma tarefa, a variação do tamanho das Strings é menos importante para as avaliações desse trabalho do que a variação do tamanho das matrizes no problema da multiplicação de matrizes, que implica em diferenças no que as tarefas precisarão executar. Também é menos importante do que a variação da quantidade de números somados na soma concorrente, que implicam diretamente na quantidade de níveis de execução do problema.

Os experimentos realizados para o problema dos filósofos fazem as seguintes variações dos fatores de avaliação:

- Quantidade de tarefas: 5, 11 e 51;
- Mecanismos de sincronização: Semáforos, synchronized e locks;
- Mecanismos de concorrência: Thread, Thread Pool, framework HaMeR e Kotlin coroutines.

Utilizando essas variações dos fatores de avaliação, foram realizados um total de 36 testes para o problema dos filósofos. Para o tempo de execução das tarefas usou-se o valor de 2 segundos e foram usadas Strings com 64 caracteres. Foram escolhidos valores impares para a quantidade de filósofos, pois desse modo a quantidade de recursos também é impar o que permite a consideração de casos em que todos os recursos não podem ser utilizados ao mesmo tempo pela existência de dependência com os outros recursos. Uma vez que esses recursos são utilizados em pares.

3.1.4 Download de arquivos

Frequentemente no contexto da computação móvel, um dispositivo precisa fazer uso da rede para obter algum arquivo, embora o processamento de requisição do arquivo não seja custoso aos processadores, o tempo até que o arquivo esteja pronto pode ser alto. Logo, se fosse executado na thread principal, resultaria no travamento da interação com o usuário até que o arquivo esteja pronto. Para evitar isso, o download deve ser executado de forma concorrente a thread principal que continua livre para a interação com o usuário dentro do aplicativo.

Desse modo, o uso de download de um arquivo como um dos testes para esse trabalho tem como objetivo principal a avaliação de como os mecanismos de concorrência possibilitam que a thread principal não seja bloqueada enquanto que uma operação demorada e que causa bloqueio na execução é executada em segundo plano. Também considera a requisição de atualização da interface de usuário para a thread principal quando o resultado da em segundo plano estiver disponível para ser usado pela aplicação. Assim esse problema consiste na UI thread solicitando o download de um arquivo para uma *worker* thread que vai realizar a requisição e esperar o download ser concluído para que em seguida envie atualizações para a interface de usuário. Isso ocorre sem que a thread principal seja bloqueada, permitindo que execute outras tarefas ou eventos de sua fila.

O download de arquivo possui objetivo de verificar o uso de recursos externos pelo dispositivo sem travar a interface com o usuário. Dessa forma a implementação consiste no download de uma imagem, permite que se escolha uma dentre 5 imagens, realiza o download e

quando este está pronto mostra a imagem ao usuário. A requisição e espera pelo arquivo é realizada em uma *worker* thread para que a UI thread não fique bloqueada enquanto o arquivo não está pronto.

Para a implementação se usa um mecanismo de concorrência para a geração de uma tarefa que não executará na UI thread e nessa tarefa realiza a requisição pela imagem.

Os parâmetros que afetam os testes para o download dessas imagens são:

- Imagem escolhida;
- Qualidade da conexão com a internet;
- Mecanismo de concorrência.

Os fatores de avaliação utilizados nesse caso foram a imagem escolhida, que variam em tamanho e o mecanismo de concorrência utilizado para fazer a requisição da imagem e a atualização da interface de usuário.

Os experimentos realizados para o download dessas imagens utilizam os seguintes valores dos fatores de avaliação:

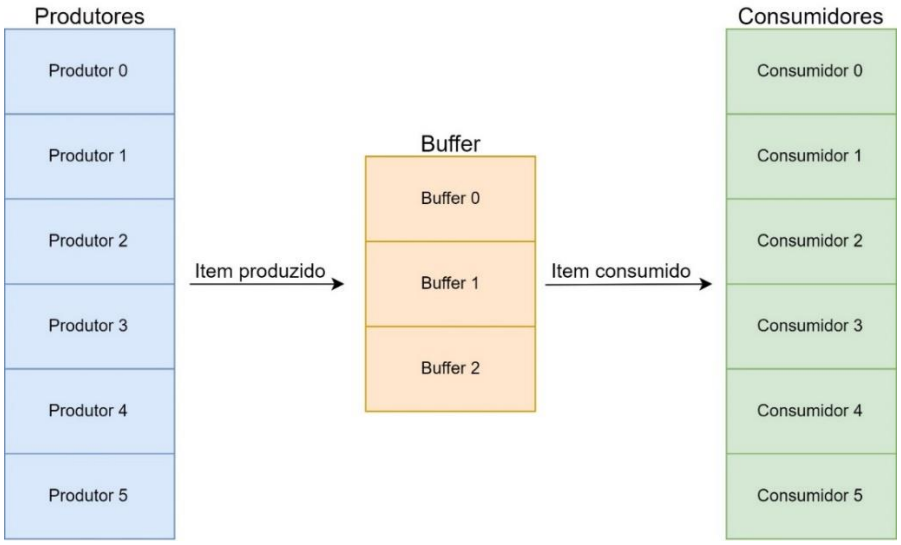
- Imagem escolhida: 5 imagens de diferentes tamanhos;
- Mecanismos de concorrência: Thread, Thread Pool, framework HaMeR, Kotlin coroutines, AsyncTask e IntentService.

Utilizando essas variações dos fatores de avaliação, foram realizados um total de 30 testes para o problema de download de arquivo.

3.1.5 Produtores-consumidores

O problema dos produtores-consumidores ou problema do *buffer* limitado, é definido como um conjunto de processos que compartilham um *buffer*, alguns desses processos produzem dados e colocam no *buffer* (produtores) e outros consomem dados removendo-os do *buffer* (consumidores), como ilustrado na Figura 3.4. O problema surge quando o produtor quer colocar um item novo no *buffer*, mas ele está cheio, ou se o consumidor quer remover um item do *buffer* que está vazio (TANENBAUM, 2016).

Figura 3.4: Problema dos produtores e consumidores.



A implementação desse problema consiste em garantir que os consumidores devam esperar até que exista algum item no *buffer* enquanto que os produtores devem esperar até que exista alguma posição livre para colocar um novo item. Esse tipo de situação é comum entre processos que cooperam entre si, por exemplo, um compilador pode produzir código assembly que é consumido por um montador (SILBERSCHATZ, 2013).

Na avaliação realizada nesse trabalho o problema dos produtores e consumidores é útil para determinar como os diferentes mecanismos de sincronização se comportam quando expostos a diferentes situações em um ambiente cooperativo, como mais produtores que consumidores, mais consumidores que produtores, *buffer* muito pequeno, entre outras. Assim, o principal objetivo das avaliações com esse problema é avaliar os mecanismos de sincronização nesses diferentes cenários que podem surgir no problema, dessa forma o foco das avaliações desse problema está nos mecanismos de sincronização e não nos de concorrência. Assim tanto os produtores quanto os consumidores são implementados estendendo a classe *Thread*.

O problema foi adaptado para que os produtores gerem um vetor aleatório, apliquem a transformada rápida de Fourier (FFT) nesse vetor e coloquem o resultado no *buffer*. Os consumidores, por sua vez, retiram esses valores do *buffer* e aplicam a transformada inversa (IFFT) para obter o vetor originalmente gerado pelos produtores. Semelhante a implementação do LCS no problema dos filósofos, o objetivo dessa implementação é a simulação de uma tarefa real realizada no contexto cooperativo entre processos que produzem dados e processos que consomem dados. As complexidades computacionais das tarefas executadas tanto pelo produtor quanto pelo consumidor são semelhantes, o que significa que a quantidade de acessos ao *buffer*

realizadas pelos produtores e pelos consumidores, após a inserção dos primeiros valores no *buffer*, deve ser proporcional a quantidade de tarefas produtoras e de tarefas consumidoras.

Tanto os produtores quanto os consumidores estão configurados para executarem durante um tempo predeterminado buscando, respectivamente, gerar e consumir a maior quantidade de dados possível. A avaliação dos mecanismos de sincronização ocorre pela quantidade total de produções e consumos realizados pelas tarefas. Assim como no problema dos filósofos, os semáforos e locks utilizados implementam uma fila justa para que os fluxos esperem pela obtenção do acesso.

Para o problema dos produtores-consumidores os parâmetros que afetam os testes são:

- Quantidade de produtores;
- Quantidade de consumidores;
- Tamanho do *buffer*;
- Mecanismo de sincronização;
- Tempo de execução das tarefas;
- Tamanho do vetor aleatório gerado pelos produtores.

Os fatores de avaliação utilizados buscam variar o cenário de execução dos experimentos, variando a quantidade de tarefas produtoras, a quantidade de tarefas consumidoras e o tamanho do *buffer*, os mecanismos de sincronização também são um fator de avaliação utilizado. Os argumentos para a não utilização dos parâmetros de tempo de execução da tarefa e de tamanho do vetor aleatório como fatores de avaliação são semelhantes aos dados para o problema dos filósofos quanto à variação do tempo de execução e do tamanho das Strings. Nesse caso o custo de avaliação introduzido seria maior ainda no que do problema dos filósofos pois já são considerados 4 fatores de avaliação ao invés de 3.

Os experimentos realizados para o problema dos produtores-consumidores realizaram as seguintes variações dos valores dos fatores de avaliação:

- Quantidade de produtores: 5, 10 e 20;
- Quantidade de consumidores: 5, 10 e 20;
- Tamanho do *buffer*: 2 e 8 posições;
- Mecanismos de sincronização: Lock e Condition, semáforos, variáveis atômicas e synchronized.

Utilizando essas variações dos fatores de avaliação, foram realizados um total de 72 testes para o problema dos produtores e consumidores. É importante notar que cada posição no *buffer* corresponde a um espaço de memória suficiente para armazenar todo o resultado da aplicação da FFT sobre o vetor aleatório gerado, portanto um produtor insere todos os dados resultantes da transformada em apenas uma posição do *buffer*. O tempo de execução predeterminado para a execução dos produtores e dos consumidores foi de 2 segundos e o vetor gerado pelos produtores contém 1024 números.

3.2 Métricas

As métricas envolvidas nos testes têm objetivo de abordar aspectos importantes para a avaliação de cada mecanismo. Desse modo, as métricas utilizadas nesse trabalho incluem:

- **Tempo de execução**: Utilizado na multiplicação de matrizes, na soma concorrente e no download de arquivo. Útil por ser uma medida direta e de fácil comparação.
- **Throughput**: Útil para determinar quanto o sistema foi capaz de produzir no geral. Usada no problema dos produtores e consumidores para medir o quão eficientes os mecanismos de sincronização foram durante a execução do problema enquanto que mantinham os estados dos fluxos de execução consistentes.
- **Justiça**: Utilizada para medir como os mecanismos influenciam no escalonamento dos fluxos concorrentes, sendo utilizada no problema dos filósofos e medida utilizando valores estatísticos da quantidade de vezes que cada filósofo conseguiu obter os dois recursos necessários para realizar a sua computação. A principal medida utilizada para avaliar a justiça desses mecanismos foi o coeficiente de variação que é calculado como o desvio padrão dividido pela média de uma população. Essa medida foi utilizada pois ao mesmo tempo que captura desigualdades na distribuição de recursos com o desvio padrão, também captura uma ideia de eficiência levando em consideração a quantidade média de recursos que foi distribuída. Logo, quanto maior for o coeficiente de variação, maior é a razão entre o desvio padrão e a média, indicando que existe uma desigualdade maior entre a distribuição de recursos entre as tarefas. Assim quanto menor for o valor do coeficiente de variação, mais justa foi a execução.
- **Speedup**: O speedup é uma medida utilizada para comparação de dois sistemas com objetivo de determinar o quão mais rápido um sistema é que outro (MARTIN, 2015). No contexto de processamento paralelo a medida geralmente é utilizada para comparar

o algoritmo sequencial com o algoritmo paralelo. O speedup ideal de uma aplicação paralela é igual a quantidade de CPUs sobre a qual os algoritmos estão executando, ou seja, em um ambiente com *p* CPUs, idealmente se espera que o algoritmo paralelo que usa todos os processadores seja *p* vezes mais rápido que o algoritmo sequencial executando nessa mesma máquina.

Dessa forma o speedup de um programa paralelo é calculado como:

$$S_p = \frac{T_s}{T_p}$$

Onde:

S_p : é o speedup utilizando *p* CPUs
T_s : é o tempo de execução do algoritmo sequencial
T_p : é o tempo de execução do algoritmo paralelo usando *p* CPUs

Como dito anteriormente, o speedup ideal para *p* CPUs é igual a *p*, contudo esse valor dificilmente é atingido na prática por diversos motivos como necessidade de sincronização, poucas tarefas paralelas no problema ou sobrecarga introduzida pela criação das tarefas. Esse fenômeno é previsto e modelado teoricamente pela lei de Amdahl e pela lei de Gustafson.

3.3 Testes de Hipótese

Testes de hipótese estatística foram realizados nesse trabalho para casos em que os valores de agregação dos dados obtidos com os experimentos forem muito próximos, pois nesses casos não se tem tanta certeza de que os desempenhos dos diferentes mecanismos sejam realmente distintos. Portanto, para esses casos, se usa como hipótese nula a possibilidade de que os dois mecanismos comparados possuem o mesmo desempenho, logo a hipótese alternativa representa a possibilidade desses desempenhos serem diferentes. O nível de significância escolhidos para todos os testes realizados no capítulo 4 foi de 0,05, esse valor foi escolhido pois corresponde a um valor de α bastante usado em teste de hipótese e por corresponder a um intervalo de confiança relativamente alto com um risco baixo de levar a uma conclusão incorreta sobre os dados.

4 EXECUÇÃO DOS EXPERIMENTOS E AVALIAÇÃO DOS RESULTADOS

Este capítulo apresenta os resultados da execução dos testes descritos no capítulo 3. A avaliação dos resultados está organizada de modo que se faz uma análise individual dos testes de cada problema, apresentando os dados de cada mecanismo avaliado que, inicialmente são comparados consigo mesmos variando os demais fatores de avaliação e em seguida são comparados com os outros modelos. Para cada um dos 216 testes listados no capítulo 3, a execução foi repetida pelo menos 30 vezes e a mediana desses valores é utilizada como medida de agregação reduzindo o efeito de *outliers* que podem surgir nesse tipo de medição pois outras tarefas do SO podem surgir e demandar recursos do hardware em casos pontuais. Contudo ao comparar os mecanismos de concorrência e de sincronização entre si nos testes de hipótese estatística, todas as execuções são levadas em consideração.

Para algumas das comparações realizadas nesse trabalho foi calculado o *p-value* entre dois mecanismos. Esse cálculo foi realizado quando os resultados médios e medianos obtidos com dois mecanismos estavam muito próximos, logo não existia confiança estatística suficiente para afirmar que um era melhor do que outro. Em todos os testes estatísticos realizados foi utilizada a distribuição t de Student com valor α igual a 0.05 a hipótese nula utilizada para as comparações foi que ambos os mecanismos comparados possuem o mesmo desempenho. Logo a hipótese alternativa assume que o desempenho dos mecanismos é diferente.

Para a execução dos testes foi utilizado primariamente o smartphone Samsung Galaxy J5, que possui um processador com 8 núcleos físicos de 1.6 GHz de modelo Cortex-A53. A versão do Android utilizada para os testes possui a arquitetura descrita na subseção 2.4.1. Os dados mostrados nas seções 4.1, 4.2, 4.3, 4.4 e 4.5 correspondem às execuções realizadas nesse dispositivo. De forma adicional, execuções remotas utilizando o *data center* Firebase Test Lab foram realizadas e são apresentadas no Apêndice A.

4.1 Multiplicação de Matrizes

Com o objetivo de avaliar a escalabilidade dos mecanismos de concorrência frente aos recursos paralelos de hardware, usou-se o problema da multiplicação de matrizes calculando o Speedup de cada um dos mecanismos.

4.1.1 Speedup

A seguir, nas Figuras 4.1, 4.2 e 4.3, são mostrados os gráficos obtidos com a execução dos experimentos da multiplicação de matrizes utilizando o Speedup como métrica. O eixo vertical desses gráficos representa o Speedup calculado como descrito na seção 3.2, comparando a versão de cada mecanismo gerando uma tarefa com a versão gerando a quantidade de tarefas especificada no eixo horizontal desse mesmo mecanismo. Nesses gráficos, os quatro mecanismos de concorrência testados com a multiplicação de matrizes são comparados quanto ao seu speedup, junto deles está o speedup ideal para a quantidade de tarefas utilizadas (considerando que o hardware que o executou possui 8 processadores). O speedup foi calculado comparando os mecanismos consigo mesmos, ou seja, o speedup de todos os mecanismos usa como tempo base a sua execução gerando uma única tarefa.

Figura 4.1: Speedup para os testes com multiplicação de matrizes com tamanhos 128x128

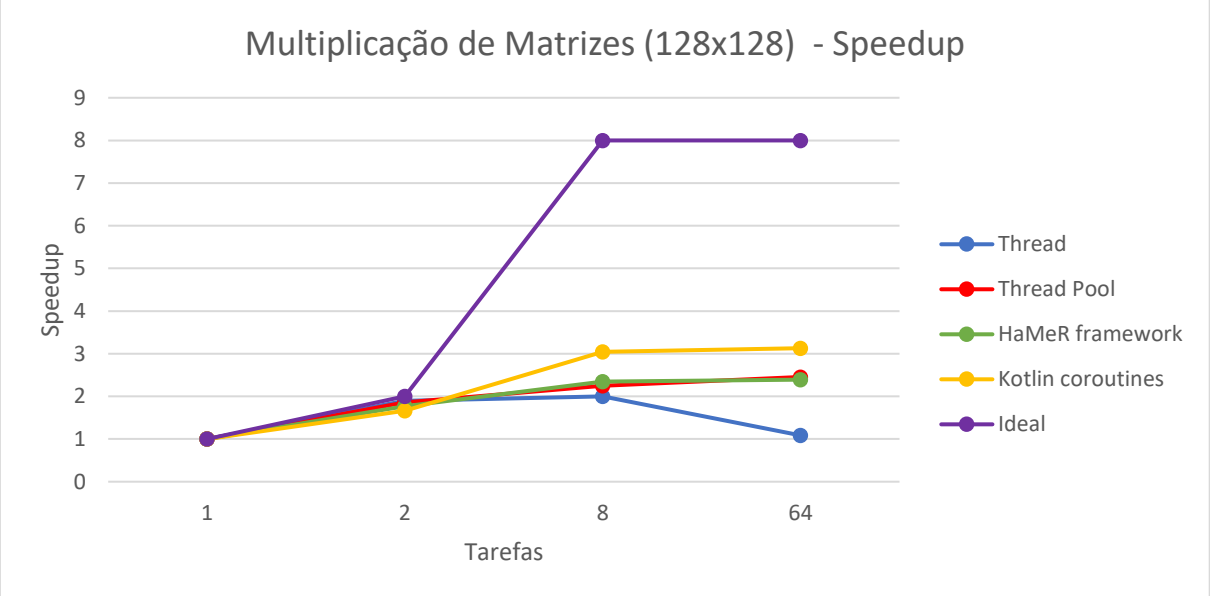


Figura 4.2: Speedup para os testes com multiplicação de matrizes com tamanhos 256x256

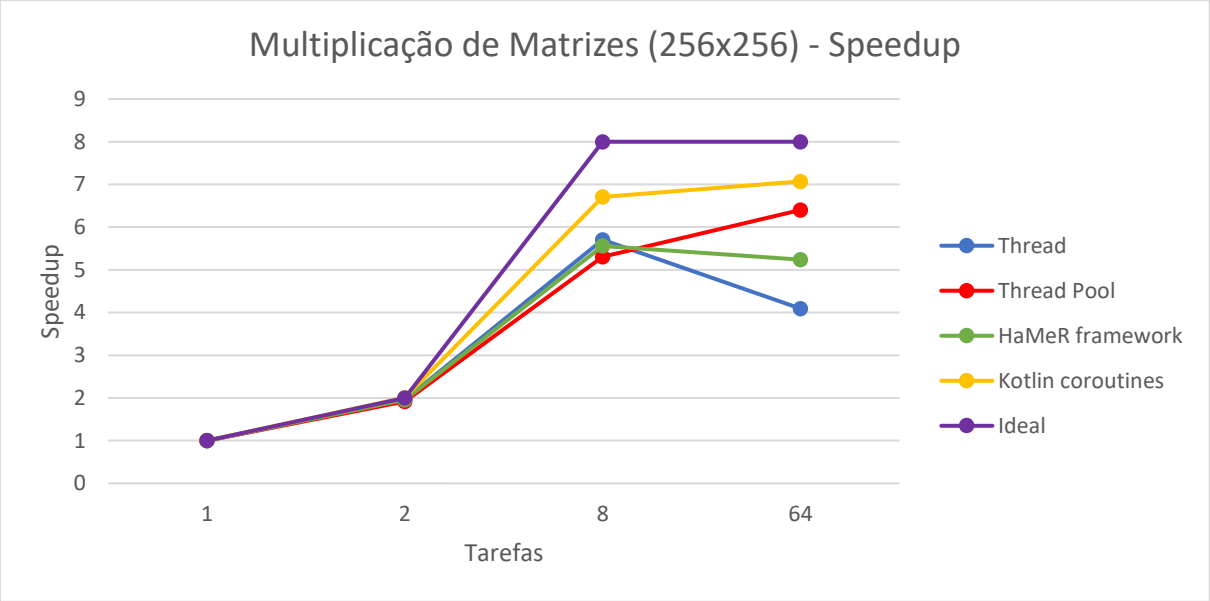
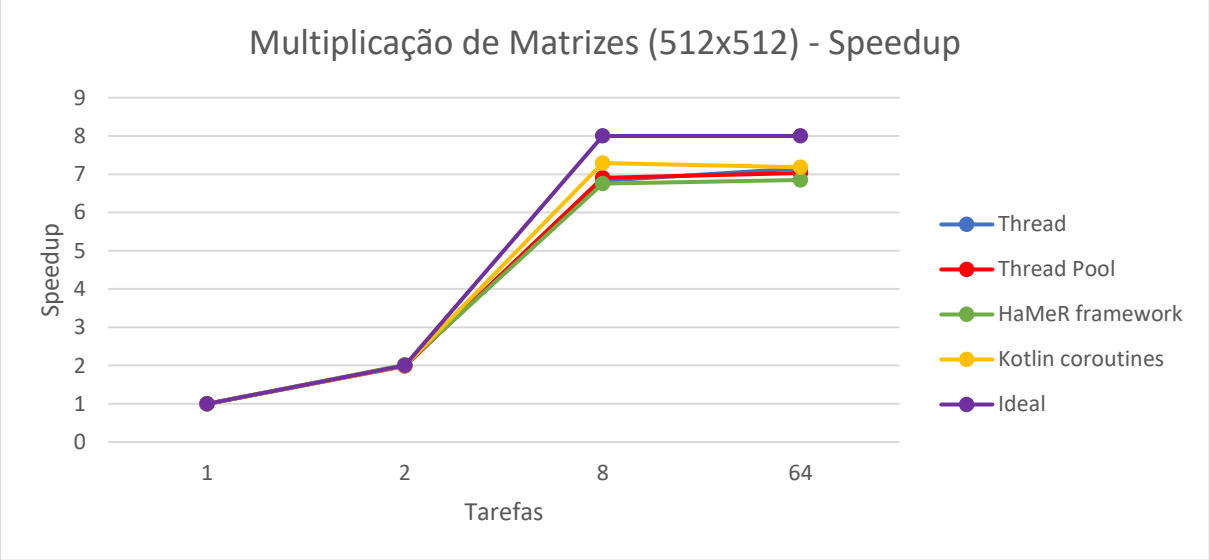


Figura 4.3: Speedup para os testes com multiplicação de matrizes com tamanhos 512x512



Com esses dados algumas observações gerais podem ser realizadas:

- Quanto maiores as matrizes mais o speedup de todos os mecanismos tendem a se aproximar do valor ideal. Isso indica que tempos de gerenciamento das tarefas como criação e destruição se tornam proporcionalmente menores se comparados ao tempo de execução, o que é um fator esperado uma vez que o hardware fica mais tempo ocupado realizando os cálculos do problema do que gerenciando tarefas.
- Para as matrizes 128x128 e 256x256 o mecanismo que apresentou maior speedup comparado com sua própria versão com uma tarefa foi Kotlin coroutines, o que, segundo essa análise o tornaria o mecanismo mais escalável frente a recursos paralelos do

hardware sendo comparado com sua própria versão com uma tarefa, contudo, como será posteriormente discutido, os tempos de execução desse mecanismo foram os piores dentre os quatro testados. Para a matriz 512x512 a mesma afirmação é válida ao se considerar o speedup mediano, no entanto essa afirmação não possui suporte do teste de hipótese, uma vez que o *p-value* comparando as populações de Kotlin coroutines e de threads foi de 9,67e-02 o que não permite a rejeição da hipótese estatística de que o desempenho desses dois mecanismos foi igual no caso de $\alpha = 0,05$. Por outro lado, realizando a mesma comparação entre Kotlin coroutines e os outros dois mecanismos se consegue significância estatística para rejeitar a hipótese tanto para Thread Pool (*p-value* = 1,07e-02) quanto para o framework HaMeR (*p-value* = 4,60e-05) no caso das matrizes de tamanho 512x512.

Analisando cada um dos mecanismos separadamente, verifica-se que o mecanismo tradicional de Threads introduz uma sobrecarga bastante considerável para a criação de cada tarefa, isso pode ser observado claramente nas Figuras 4.1 e 4.2 onde existe uma queda considerável no speedup das execuções que utilizam 8 tarefas para as execuções com 64 tarefas.

O mecanismo de Thread Pool, por sua vez, não introduz uma sobrecarga tão alta quanto o mecanismo de Threads para o gerenciamento das tarefas. O que faz sentido teórico considerando que as tarefas de uma Thread Pool executam sobre Threads que são criadas no início da execução e executam as tarefas conforme chegam na sua fila, portanto existem menos Threads reais para gerenciar no caso de 64 tarefas.

O framework HaMeR, por sua vez, possui speedup praticamente igual ao de Thread Pool nas Figuras 4.1 e 4.3, contudo, na Figura 4.2, mostra uma redução no speedup ao usar 64 tarefas, o que indica que o gerenciamento dos seus Handlers e dos seus Loopers é mais custoso do que o gerenciamento dos tarefas do Thread Pool, porém menos custoso do que objetos Thread.

Kotlin coroutines ficou bem próximo dos valores ideais de speedup nas Figuras 4.2 e 4.3 o que cumpre o objetivo principal do mecanismo de ser uma forma leve de definir tarefas concorrentes. Vale lembrar, contudo, que esse gráfico expressa o speedup calculado com base no próprio mecanismos ao gerar apenas uma tarefa. Como será mostrado na seção seguinte, apesar de Kotlin coroutines apresentar o maior speedup quando aumenta a quantidade de tarefas seu tempo de execução foi o pior dentre os dispositivos analisados.

4.1.2 Tempo de execução

Quanto ao tempo de execução dos experimentos obteve-se os valores medianos apresentados nas Tabelas 4.1, 4.2 e 4.3. A observação mais clara a se fazer desses tempos é a grande diferença entre Kotlin coroutines e os outros mecanismos e, muito embora tenha ficado com bons valores na avaliação pelo speedup, apresentou os piores tempos de execução. Isso indica que existe uma sobrecarga consideravelmente alta durante a execução das tarefas utilizando Kotlin coroutines. No entanto, como indicado pelo speedup, o tempo para gerenciamento de várias tarefas é relativamente baixo, tornando-o um mecanismo indicado para tarefas de curta duração, resultado reforçado na seção 4.2, onde o tempo de gerenciamento das tarefas é proporcionalmente maior. Contudo não é indicado para tarefas longas, onde o tempo de gerenciamento das tarefas se torna quase desprezível.

Tabela 4.1: Tempo de execução em ms para matrizes de tamanho 128x128

Mecanismo	1 tarefa	2 tarefas	8 tarefas	64 tarefas
Threads	56,0	29,5	28,0	51,5
Thread Pool	54,0	29,0	24,0	22,0
HaMeR framework	61,0	34,5	26,0	25,5
Kotlin coroutines	217,5	131,0	71,5	69,5

Tabela 4.2: Tempo de execução em ms para matrizes de tamanho 256x256

Mecanismo	1 tarefa	2 tarefas	8 tarefas	64 tarefas
Threads	673,0	343,0	118,0	164,5
Thread Pool	669,0	349,5	126,0	104,5
HaMeR framework	686,5	350,5	123,5	131,0
Kotlin coroutines	1.973,0	982,0	294,0	279,0

Tabela 4.3: Tempo de execução em ms para matrizes de tamanho 512x512

Mecanismo	1 tarefa	2 tarefas	8 tarefas	64 tarefas
Threads	7.292,0	3.636,5	1.067,0	1.019,5
Thread Pool	7.296,5	3.665,0	1.056,5	1.037,5
HaMeR framework	7.234,5	3.580,0	1.070,5	1.056,0
Kotlin coroutines	16.898,5	8.453,5	2.317,5	2.353,0

4.2 Soma Concorrente

As análises realizadas para o problema da soma concorrente buscam determinar como a existência de muitas tarefas afeta os mecanismos de concorrência. Diferentemente do problema da multiplicação de matrizes explicado na seção 4.1, os resultados dos testes apresentados aqui não possuem um valor próxima do speedup ideal para a realização das somas em paralelo, visto que o tempo de gerenciamento se torna relativamente maior com a necessidade de inclusão de mecanismos de sincronização entre os níveis de execução do algoritmo. Esses tempos podem ser vistos nos gráficos das Figuras 4.4 e 4.5.

Figura 4.4: Tempo de execução para a soma concorrente utilizando 262.144 números

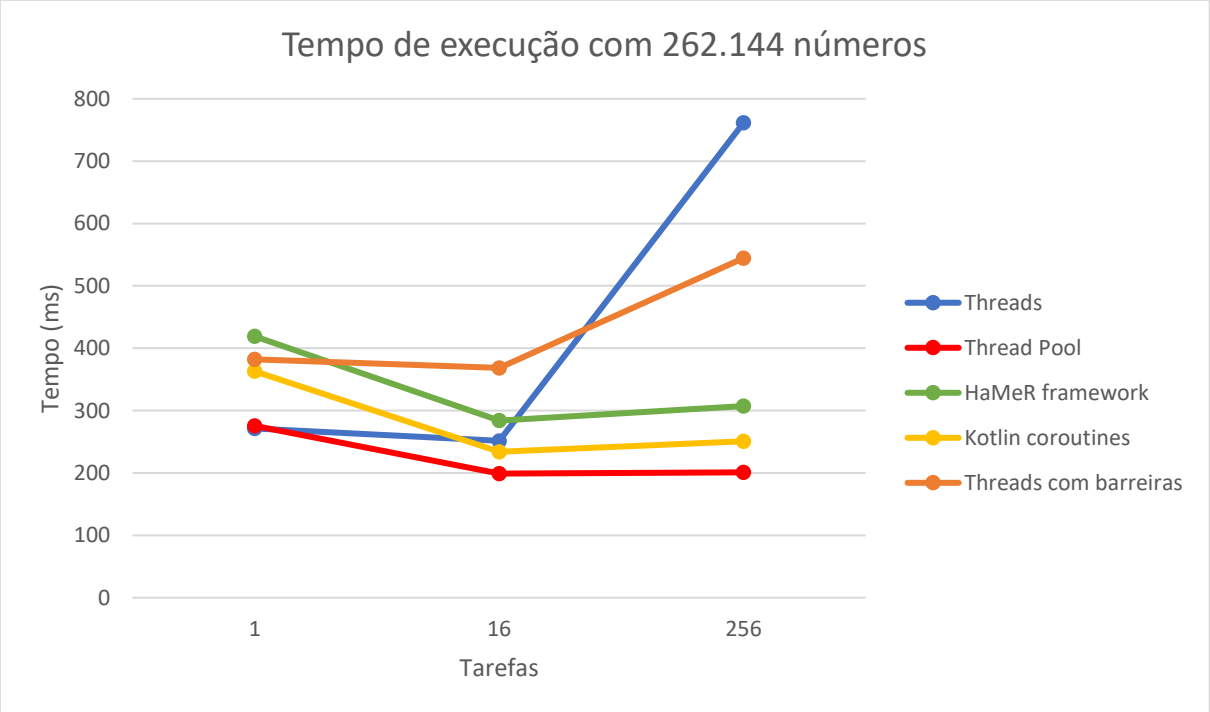
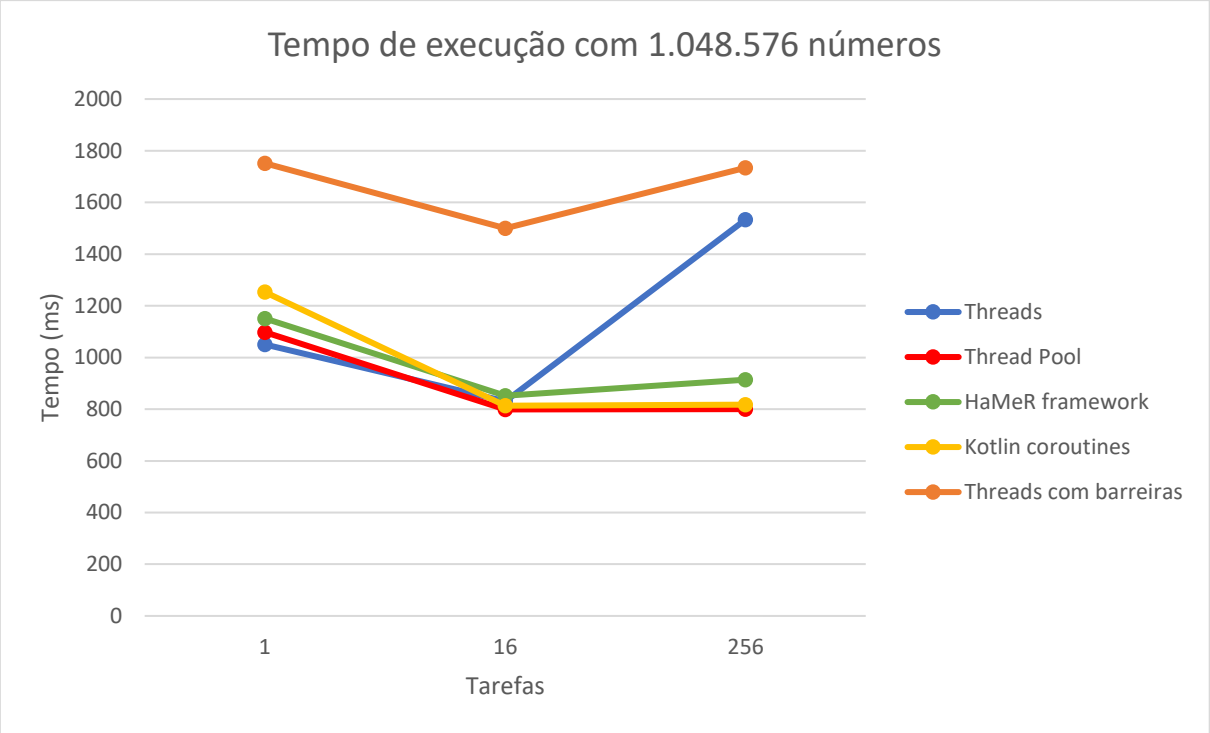


Figura 4.5: Tempo de execução para a soma concorrente utilizando 1.048.576 números



Ao analisar os gráficos das Figuras 4.4 e 4.5 é possível notar que, diferente do caso da multiplicação de matrizes, o mecanismo de Kotlin Coroutines foi um dos que possuiu menor tempo quando 256 tarefas são usadas a cada nível de execução do algoritmo. Isso fortalece a ideia apresentada na seção 4.1 de que esse mecanismo é especialmente útil para tarefas simples de curta duração. Em ambos os gráficos também é possível notar que o mecanismo que apresentou o maior aumento de tempo entre o uso de 16 tarefas e o uso de 256 tarefas foi o mecanismo de Threads, que, apesar de ser bastante eficiente comparado com os outros ao usar apenas uma tarefa, mostrou novamente que introduz um custo alto de gerenciamento quando as tarefas são curtas.

Quanto às execuções com 1.048.576 números, Thread Pool foi o mecanismo de concorrência com menor tempo mediano tanto para 16 quanto para 256 tarefas seguido por Kotlin coroutines em ambos os casos. Isso pode ser afirmado com confiança estatística para o caso de 256 tarefas, onde, apesar de visualmente o valor parecer muito próximo de Kotlin coroutines, o *p-value*, assumindo a hipótese nula como o desempenho de Kotlin coroutines sendo igual ao de Thread Pool, é igual a 2,98e-02 o que é um pouco menor do que o valor α, ou seja, 0,05. Indicando que existe confiança estatística para rejeitar a hipótese nula. Contudo o mesmo não pode ser confirmado para 16 tarefas onde o *p-value* da comparação entre Kotlin coroutnes e Thread Pool foi de 3,65e-01 o que não permite que a hipótese nula seja rejeitada

uma vez que é maior do que o valor α. Portanto não se pode afirmar, com significância estatística, que os desempenhos são diferentes para o caso de 16 tarefas, embora o valor mediano do tempo das execuções com Thread Pool seja menor.

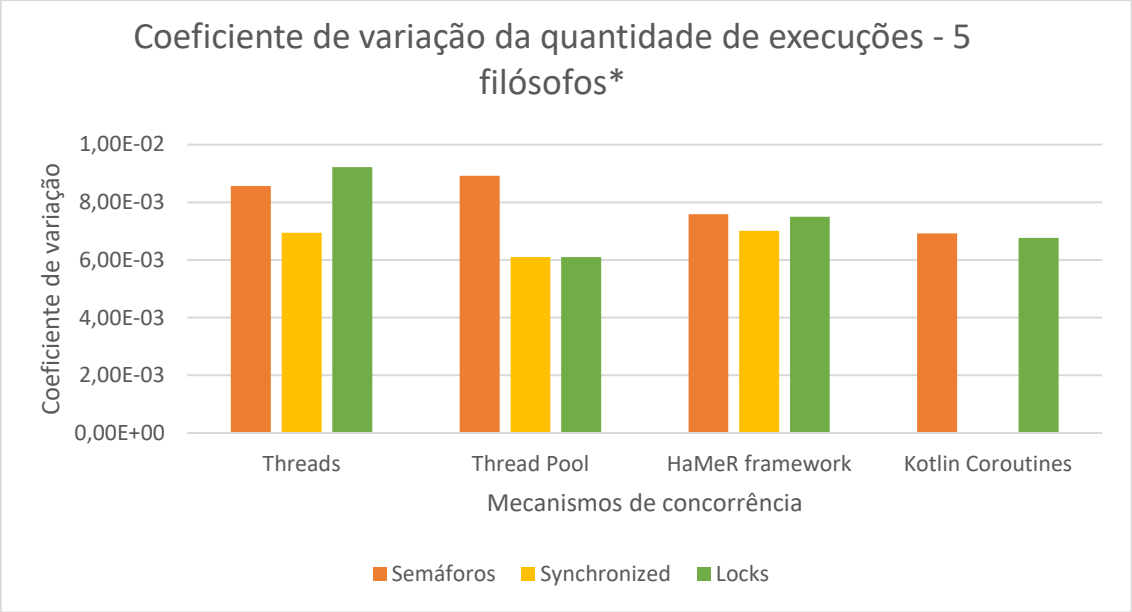
É interessante notar também, que ao se modificar o uso de Threads que operam somente em um nível do algoritmo de soma concorrente e utilizar threads que esperam em uma barreira em cada nível, o aumento de tempo, citado anteriormente, foi menor, contudo o uso de barreiras parece introduzir uma sobrecarga alta para a sincronização dos níveis, isso é especialmente visível na Figura 4.5, onde o uso de threads com barreiras possuiu o pior dentre os desempenhos apresentados.

4.3 Problema dos Filósofos

Ao realizar os experimentos do problema dos filósofos, o foco foi na avaliação da justiça dos mecanismos de concorrência e de sincronização para o compartilhamento de recursos de hardware, como uso do processador, e de recursos de software, como os garfos do problema dos filósofos. Para realizar essa avaliação a principal medida usada foi o coeficiente de variação da quantidade de vezes que cada tarefa conseguiu obter os recursos, implementados como Strings, e executar o algoritmo do LCS. O coeficiente de variação é uma medida estatística calculada como a razão entre o desvio padrão e a média de uma população, essa medida foi escolhida pois oferece um indicativo relativo à média da população de quanto se espera que cada amostra varie. Dessa forma, quanto menor for o coeficiente de variação mais justa foi a execução. Como medidas auxiliares foram verificados os valores máximos e mínimos da quantidade de execução de cada uma das tarefas envolvidas no problema, esses valores não são mostrados em gráficos, mas contribuem para as conclusões posteriores.

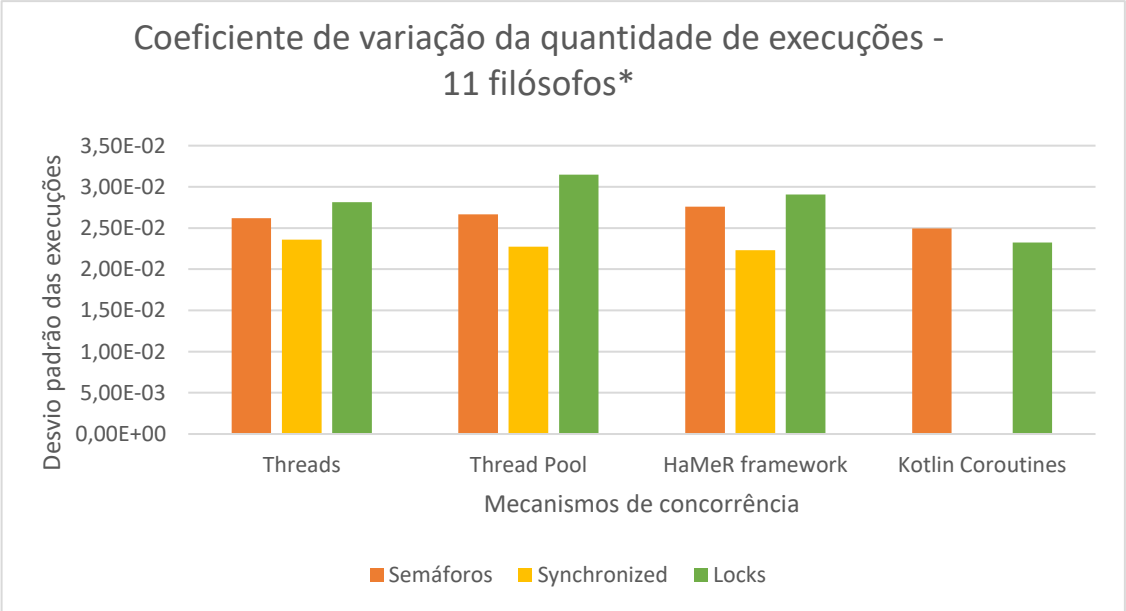
As Figuras 4.6, 4.7 e 4.8 relacionam cada combinação de mecanismo de concorrência com mecanismo de sincronização com seu respectivo coeficiente de variação para a quantidade de execuções no problema dos filósofos. Nesses gráficos, existem doze barras verticais agrupadas em grupos de três, cada grupo corresponde a um mecanismo de concorrência e cada uma das três barras de um grupo corresponde a um mecanismo de sincronização. O coeficiente de variação é mostrado no eixo vertical usando notação científica para facilitar a visualização.

Figura 4.6: coeficiente de variação da quantidade de execuções para 5 filósofos



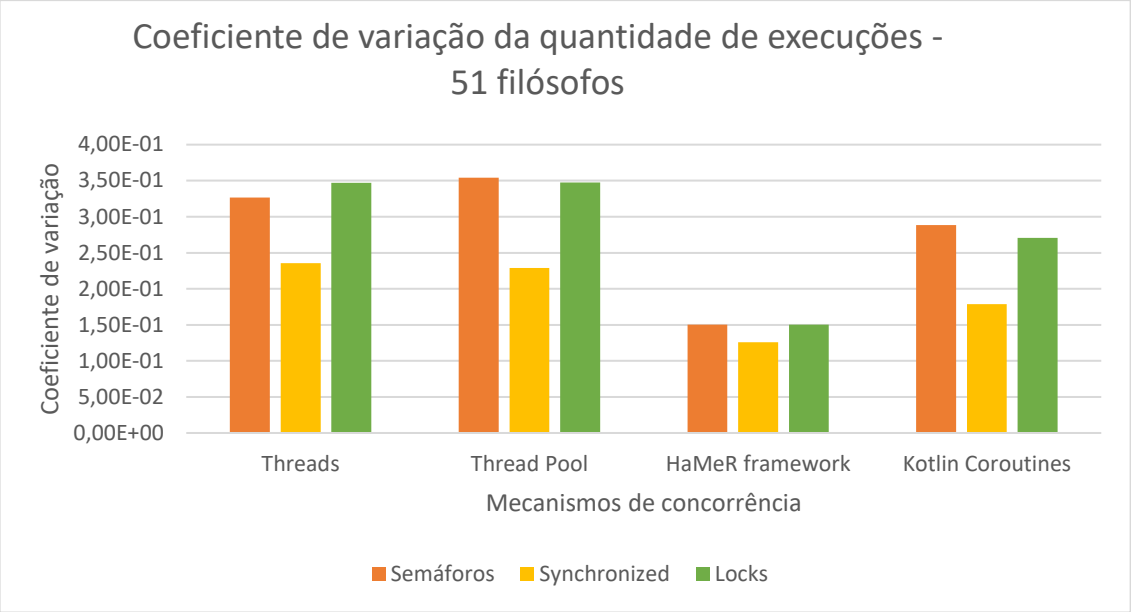
(*): O coeficiente de variação para a execução com blocos synchronized e Kotlin coroutines não é mostrada pois seu valor é mais do que 10 vezes maior do que os demais.

Figura 4.7: Coeficiente de variação da quantidade de execuções para 11 filósofos



(*): O coeficiente de variação para a execução com blocos synchronized e Kotlin coroutines não é mostrada pois seu valor é mais do que 10 vezes maior do que os demais.

Figura 4.8: Coeficiente de variação da quantidade de execuções para 51 filósofos



O fato mais notável nos dados referentes às execuções com 5 e com 11 filósofos foi a grande diferença da execução utilizando blocos synchronized com Kotlin coroutines que, ao mesmo tempo, possui um coeficiente de variação mais do que 10 vezes maior do que as demais combinações. Também possui a maior média de execuções por filósofo que foi uma ordem de grandeza maior do que os demais e, nessas execuções, os valores máximos foram cerca de 10 vezes maiores do que os valores mínimos, fato que não ocorre nas outras execuções onde o valor máximo raramente é maior do que o dobro do valor mínimo. Esse fenômeno não aparece nas execuções com 51 filósofos, não pela redução no coeficiente de variação da execução de Kotlin coroutines com blocos synchronized, mas pelo aumento desse valor para os outros mecanismos. De fato, o aumento do coeficiente de variação dos demais mecanismos foi suficiente para fazer que a implementação com synchronized e com Kotlin coroutines passasse a possuir o menor valor dentre as implementações com Kotlin coroutines. Contudo essa implementação (Kotlin coroutines com synchronized) continua sendo a que possui maior média de execuções ainda uma ordem de grandeza a frente das outras. É difícil aplicar um juízo ao uso de Kotlin coroutines com synchronized, pois, embora tenha sido a combinação menos justa, permitiu a maior quantidade de execuções no total, mostrando que o uso dessa combinação se adequa a cenários em que se deseja a maior quantidade possível de execuções no sistema e a justiça entre os serviços que estão executando está em segundo plano.

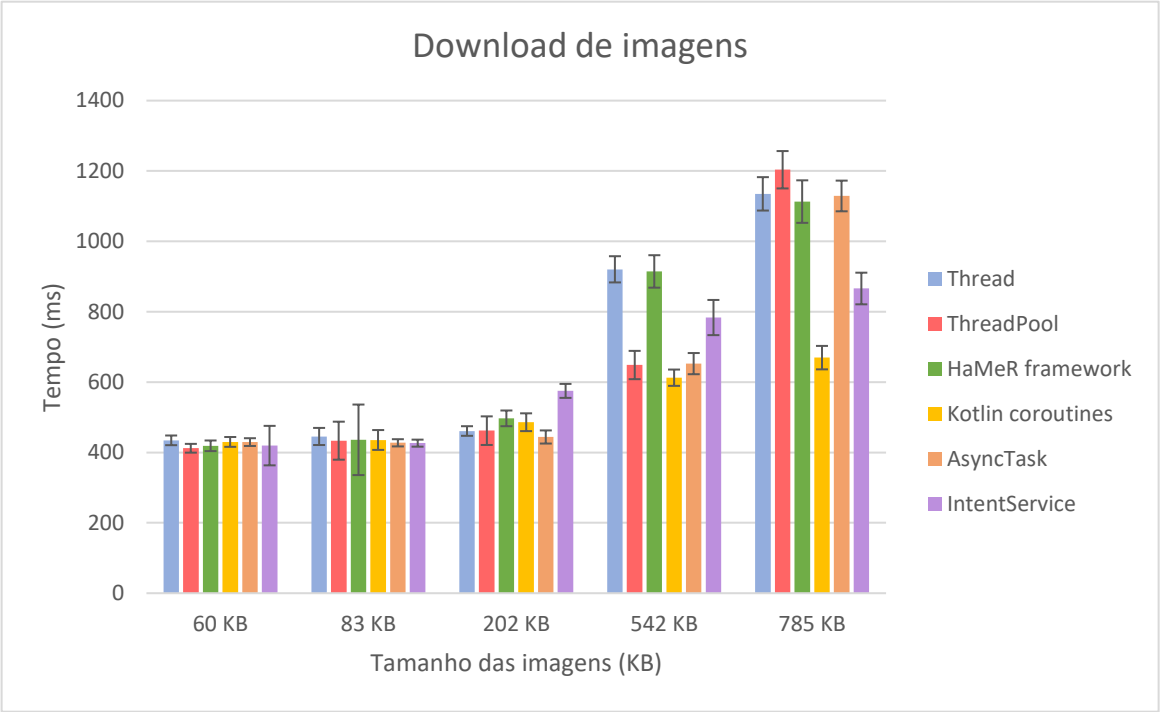
Quanto às outras implementações, é possível notar que em quase todos os casos, independente do mecanismo de concorrência, o mecanismo de sincronização que é mais justo em termos do coeficiente de variação é o synchronized. Isso foi observado nos experimentos principalmente pelo aumento da média de execuções em comparação com os outros mecanismos de sincronização enquanto que os valores do desvio padrão desses mecanismos eram bastante similares, nesses casos cada um dos filósofos consegue executar, em média 48.06%, mais vezes ao utilizar blocos synchronized para sincronização, mantendo o desvio padrão semelhante aos outros casos.

Dentre os mecanismos de concorrência, não foi possível observar muitas diferenças, com exceção da execução com 51 filósofos que indica que o uso de HaMeR framework foi, no geral, mais justo que os outros mecanismos, independente do mecanismo de sincronização utilizado, indicando que esse mecanismo pode ser especialmente útil para casos onde vários processos precisam competir pelo uso de recursos. Essa observação possui suporte estatístico realizando o teste de significância estatística com hipótese nula de que os mecanismos possuem o mesmo desempenho e valor $\alpha = 0,05$. Uma vez que a hipótese é rejeitada em todos os casos de comparação do framework HaMeR com os demais mecanismos de concorrência para o caso com 51 filósofos, onde o maior *p-value* observado (1,18e-04) ocorreu quando foi comparado com Kotlin coroutines usando locks e Condition e foi muito menor do que o valor α.

4.4 Download de arquivos

Tendo como objetivo a verificação de como os mecanismos de concorrência descritos na seção 2.5 permitem o uso de recursos externos pelo dispositivo de forma concorrente à UI thread, esse problema testou a realização do download de 5 imagens de diferentes tamanhos que são mostradas ao usuário quando o download é finalizado. No gráfico da Figura 4.9, o tempo para a realização de todo esse processo é mostrado. Vale notar, que, por utilizar recursos da rede, as variações do tempo em cada uma das repetições realizadas foram relativamente altas se comparadas aos outros experimentos, dessa forma, as barras do gráfico da Figura 4.9 são determinadas pelo valor mediano do tempo e possuem barras de erro determinadas pelo erro padrão do tempo de cada uma das execuções. As barras de erro foram adicionadas somente nesse gráfico e não nos gráficos das avaliações dos outros mecanismos pois o erro padrão dos outros experimentos foi muito baixo, logo as barras de erro seriam praticamente imperceptíveis.

Figura 4.9: Tempo para os experimentos com download de imagens



Pouco pode ser dito sobre os testes realizados com as imagens menores além de que todos os mecanismos de concorrência apresentaram tempos muito semelhantes, contudo se nota que na imagem de 83 KB, o erro padrão sofrido pela implementação com HaMeR framework é muito mais alto do que os demais mecanismos para a mesma imagem. Isso ocorre, pois para esse caso em específico a quantidade de repetições que se comportam como *outliers* é maior do que nos outros casos.

Quanto às duas imagens maiores, é possível notar que em ambas o mecanismo de Kotlin coroutines possuiu o menor dos tempos e o menor erro padrão, essa diferença, no entanto, é baixa na imagem de 542 KB se comparada com Thread Pool e com AsyncTask, onde as barras de erro desses mecanismos possuem intersecções de tempo. Essa superioridade de Kotlin coroutines é confirmada estatisticamente para a imagem de 785 KB onde o maior *p-value* realizando a comparação de Kotlin coroutines com os demais mecanismos usando como hipótese nula que ambos possuem o mesmo desempenho é de 2,30e-04 o que é bem menor que o valor α (0,05) indicando que a hipótese deve ser rejeitada. Isso, no entanto, não se repete na imagem de 542 KB, onde a significância estatística não é confirmada ao comparar Kotlin coroutines com Thread Pool (*p-value* = 1,44e-01) nem com AsyncTask (*p-value* = 1,43e-01).

Muito embora essas observações puderam ser feitas sobre esse problema, um fator importante para os resultados é o efeito da rede no tempo de execução que acaba ocupando uma boa parte do tempo total medido. Mesmo que todas as requisições de download foram feitas de modos equivalente, isso acabou tornando esse o problema menos conclusivo dos cinco testados pelo alto erro padrão se comparado com os demais testes e pela dependência da qualidade da rede.

4.5 Produtores-Consumidores

O objetivo dos experimentos com o problema dos produtores-consumidores é de avaliar os impactos dos mecanismos de sincronização em diferentes cenários de uma computação cooperativa onde um conjunto de tarefas produz dados que serão consumidos por outro conjunto de tarefas. Os dados produzidos são armazenados em um *buffer* onde aguardam até que alguma das tarefas consumidores requisite seu consumo. Nesse ambiente a quantidade de itens produzidos nunca é menor do que a quantidade consumida e a diferença entre a quantidade produzida e a quantidade consumida nunca é maior do que o tamanho do *buffer*. Realizando essas considerações os gráficos das figuras 4.10, 4.11 e 4.12 foram gerados, de modo que cada grupo de barras verticais representa uma configuração diferente do tamanho do *buffer* (B) e da quantidade de tarefas consumidoras (C). A quantidade de itens produzidos é usada como medida, pois todas as execuções executaram durante o mesmo tempo, isto é, dois segundos, e uma vez que a diferença entre esse valor e a quantidade de itens consumidos nunca ser maior do que o tamanho do *buffer*.

Figura 4.10: Itens produzidos utilizando 5 tarefas produtoras

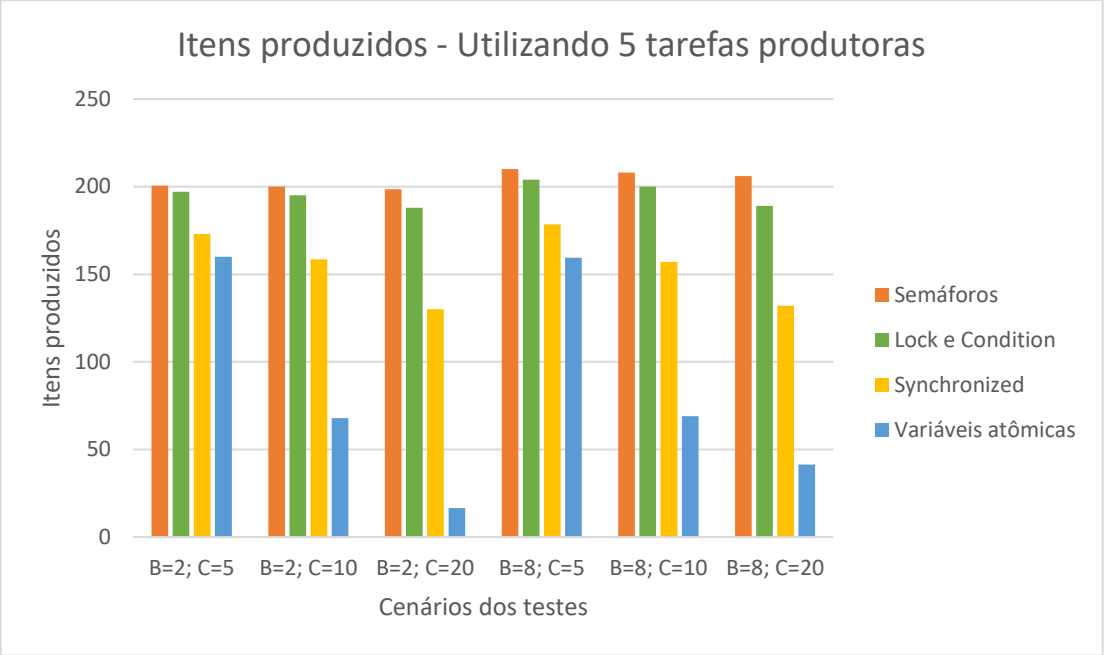


Figura 4.11: Itens produzidos utilizando 10 tarefas produtoras

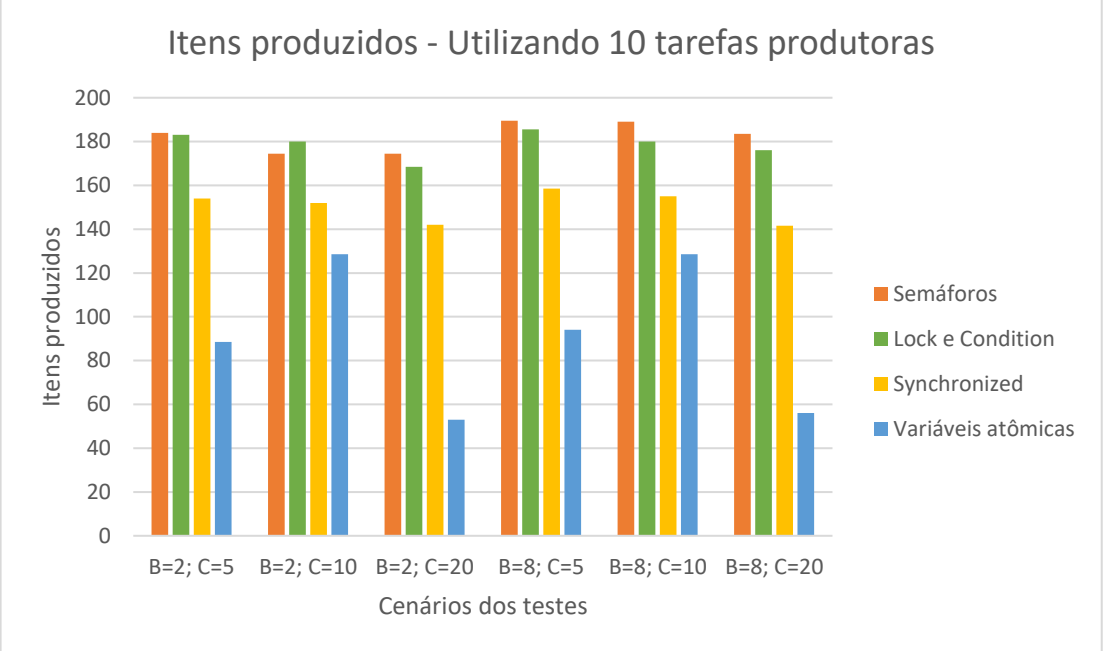
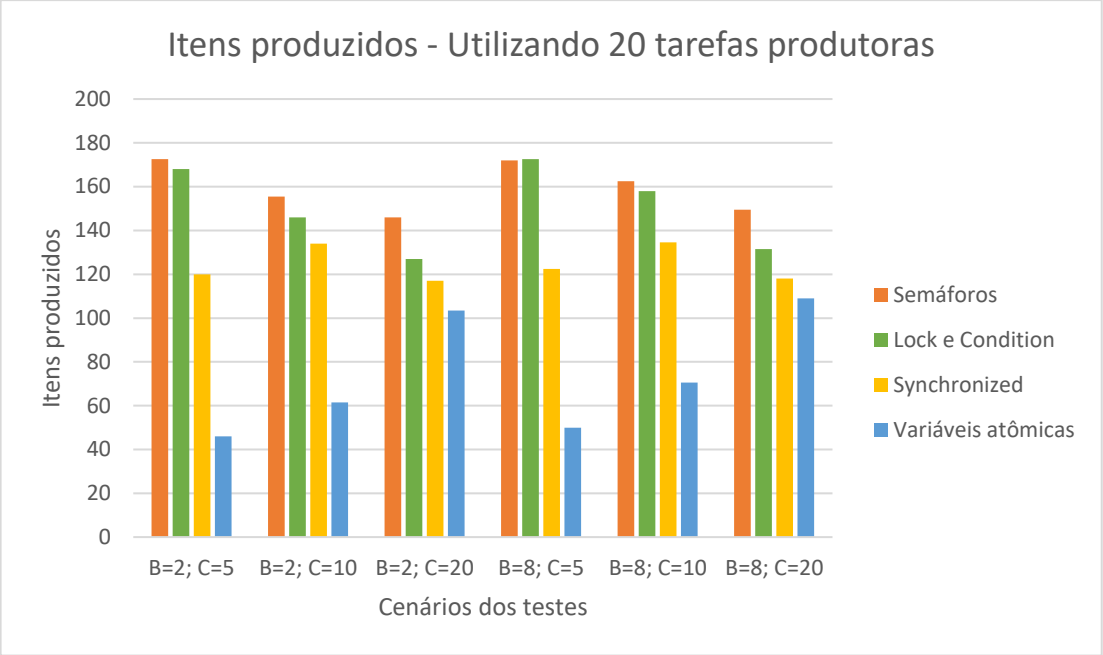


Figura 4.12: Itens produzidos utilizando 20 tarefas produtoras



Na maioria dos resultados mostrados nas Figuras 4.10, 4.11 e 4.12, a ordem de quantidade de itens produzidos é a mesma, ou seja, a sincronização com semáforos produzindo mais itens, seguida de perto pelos testes com lock e Condition seguidos pelas execuções com synchronized e por último, várias vezes com uma ampla desvantagem, as execuções com variáveis atômicas. Isso indica que a sobrecarga introduzida pelos mecanismos de sincronização para as execuções das tarefas segue aproximadamente essa ordem independe da quantidade de tarefas produtoras, de tarefas consumidoras e do tamanho do *buffer*. Considerando essas observações, ao executar o teste de significância estatística entre os mecanismos de sincronização em cada cenário, utilizando o valor 0,05 para α e assumindo como hipótese nula que não existe diferença entre os mecanismos de sincronização, obteve-se sempre *p-values* muito baixos. Isso indica que a hipótese nula deve ser rejeitada e que, portanto, as distribuições não devem ser consideradas iguais. Uma exceção para isso foi a comparação entre semáforos e locks com Condition, cujos *p-values* são mostrados nas Tabelas 4.4 e 4.5.

Tabela 4.4: *p-value* assumindo semáforos = lock e Condition utilizando *buffer* de tamanho 2.

Mecanismo	**5 Produtores**	**10 Produtores**	**20 Produtores**
5 Consumidores	7,96744e-01	3,66205e-01	1,26342e-01
10 Consumidores	6,26898e-01	1,41830e-02	9,73594e-03
20 Consumidores	5,91661e-05	1,55025e-01	2,71764e-12

Tabela 4.5: *p-value* assumindo semáforos = lock e Condition utilizando *buffer* de tamanho 8.

Mecanismo	5 Produtores	10 Produtores	20 Produtores
5 Consumidores	2,03968e-02	4,12653e-02	8,86926e-01
10 Consumidores	7,12951e-06	1,09133e-03	2,56399e-01
20 Consumidores	1,28531e-07	8,87654e-04	1,48300e-10

Nota-se nas Tabelas 4.4 e 4.5 que, enquanto que em alguns cenários como quando existe 20 tarefas produtoras e 20 tarefas consumidoras, o *p-value* indica fortemente que a hipótese nula de que os desempenhos foram iguais deve ser rejeitada. Por outro lado, em outros casos, como quando existe 20 produtores e 5 consumidores com *buffer* de tamanho 2 o *p-value* é muito maior que o valor de α (0,05), indicando fortemente que a hipótese não deve ser rejeitada e afirmando que não se tem evidências para dizer que uma distribuição é diferente da outra, para esse caso em especial os valores são quase indistinguíveis na Figura 4.12. Com isso, a comparação entre semáforos e lock e Condition não pode ter um resultado em que um é melhor que o outro sempre. Muito embora em 11 dos 18 cenários a hipótese nula foi rejeitada e apenas em um deles (10 produtores, 10 consumidores e *buffer* de tamanho 2) existe indicação de superioridade de lock e Condition sobre semáforos, nos outros 10 existe indicação de que semáforos permitiu um desempenho melhor.

Outra característica interessante que pode ser percebida com os resultados das Figuras 4.10, 4.11 e 4.12 é que as execuções com variáveis atômicas sempre foram mais eficientes quando a quantidade de tarefas produtoras era igual a quantidade de tarefas consumidoras indicando que a manutenção de um *buffer* por variáveis atômicas funciona melhor quando a quantidade de pedidos para inserção é semelhante a quantidade de pedidos para remoção de itens.

Vale notar ainda que, diferente do que se imagina em uma primeira impressão do problema, a quantidade de itens produzidos durante os testes tendeu a reduzir com o aumento de tarefas produtoras, o que, apesar de parecer contra intuitivo, faz sentido, uma vez que, quando existem muitas tarefas, o tempo de execução é dividido entre elas e existe mais esforço necessário para sincronização.

5 CONCLUSÃO

A programação concorrente é uma área antiga na computação e com soluções bem consolidadas possuindo objetivo de não somente permitir a redução do tempo de execução de uma aplicação ao utilizar recursos paralelos, mas também de permitir que um programa possa ser composto de diferentes fluxos de execução com objetivos diferentes, mas que contribuem para a execução global da aplicação.

O ambiente de desenvolvimento de aplicações móveis, por sua vez, tem como um de seus principais objetivos oferecer ao usuário aplicações que possam ser usadas de forma fácil e ofereçam uma boa experiência de usuário. Contudo, com a diversificação das tarefas executadas em um smartphone, tarefas custosas poderiam impedir o oferecimento dessa boa experiência para o usuário pois levariam ao bloqueio da interação dele com a interface e da possibilidade de que receba feedback.

Desse modo, o desenvolvimento de aplicações móveis não pode ser visto separado da programação concorrente, dado que, por meio da criação de fluxos concorrentes, um desenvolvedor de aplicações móveis pode executar tarefas custosas em segundo plano.

Assim, esse trabalho realizou uma avaliação de alguns mecanismos de concorrência e de sincronização disponíveis para a plataforma Android. Possuindo objetivo de determinar em quais ocasiões cada um deles deve ser utilizado e como a escolha pode afetar o desempenho geral da aplicação em termos da sua escalabilidade frente a recursos paralelos, sobrecarga introduzida para o gerenciamento das tarefas concorrentes, justiça na distribuição de recursos, entre outros.

Este capítulo apresenta as conclusões sobre esse trabalho, inicialmente apontando observações sobre os resultados dos mecanismos de concorrência, na seção 5.1. Em seguida, na seção 5.2, apresenta algumas conclusões sobre os mecanismos de sincronização. Por fim, na seção 5.3, são feitas algumas recomendações para trabalhos futuros.

5.1 Mecanismos de Concorrência

A Tabela 5.1 sumariza as conclusões sobre os mecanismos de concorrência avaliados nesse trabalho.

Tabela 5.1: Comparação dos mecanismos de concorrência.

Características	K. Coroutines	Threads	Thread Pool	HaMeR
Escalabilidade	✓	✗	−	−
Execução	✗	✓	✓	✓
Sobrecarga	✓	✗	✓	−
Justiça	−	−	−	✓

Sobre os mecanismos de concorrência, concluiu-se que Kotlin coroutines apresentou-se mais escalável quanto ao uso de recursos computacionais, no sentido que seu Speedup foi o mais próximo do speedup ideal, contudo, apresenta uma sobrecarga alta durante a execução das tarefas o que aumenta seu tempo de execução, sendo mais indicada para tarefas de curta duração. Pôde-se concluir também que o mecanismo de Threads de Java apresenta um bom tempo de execução se comparado aos outros mecanismos quando poucas Threads são iniciadas, contudo apresenta um tempo de gerenciamento alto para esses fluxos tornando-o menos recomendável para cenários com muitas tarefas de curta duração, mas uma boa alternativa quando existem poucas tarefas de longa duração.

Sobre a justiça na distribuição de recursos, constatou-se que o uso de HaMeR framework com muitas tarefas concorrentes foi o mais justo dentre os mecanismos de concorrência avaliados, permitindo que as tarefas competidoras executassem de forma mais eficiente e com pouca desigualdade na distribuição de recursos.

Ao utilizar recursos da rede, foi possível perceber que existe pouca diferença entre os mecanismos avaliados uma vez que nesses casos sua tarefa é de realizar a requisição pelos arquivos e enviar pedido de atualização da interface de usuário quando esses arquivos estiverem disponíveis. Nesse cenário existiu um erro padrão alto da eficiência dos mecanismos devido à interferência da rede.

5.2 Mecanismos de Sincronização

Sobre a justiça na distribuição de recursos em um ambiente com muitas tarefas, os mecanismos de sincronização testados, ou seja, Semáforos, Locks e synchronized, foram semelhantes entre si com uma leve vantagem a synchronized que permitiu uma execução mais eficiente das tarefas competidoras mantendo um desvio padrão baixo da quantidade de recursos atribuído a cada uma. Também se nota que o uso de Kotlin Coroutines com blocos synchronized apresenta um comportamento interessante quando a quantidade de tarefas não é alta, pois a eficiência geral do sistema foi maior do que os demais, contudo a justiça na distribuição dos recursos foi menor, indicando que nesse cenário existe mais priorização a certas tarefas sobre outras.

Observou-se também que, em um ambiente onde tarefas com funções diferentes no programa operam em um esquema como o problema dos produtores-consumidores, o uso de variáveis atômicas para gerenciamento do *buffer* se apresenta como a pior das opções. Pois introduz uma sobrecarga considerável para a sincronização da estrutura de armazenamento em especial quando a quantidade de tarefas produtoras era diferente da quantidade de tarefas consumidoras. Nesse caso o mecanismo que permitiu maior eficiência na maioria dos casos testados foi o de semáforos, seguido de perto por locks e Condition seguido por synchronized e por último variáveis atômicas.

5.3 Trabalhos futuros

Para trabalhos futuros sugere-se testes que comparem ferramentas que geram código nativo com tecnologias para desenvolvimento móvel multiplataforma como o framework Flutter para a linguagem Dart, as ferramentas do Microsoft Visual Studio Xamarin, entre outros.

Também se sugere testes para campos mais específicos de aplicações móveis, como jogos, e avaliações dos efeitos que diferentes modos de interagir com o usuário podem causar sobre os mecanismos de concorrência e de sincronização.

REFERÊNCIAS

AMDAHL, Gene M. **Validity of Single Processor Approach to Achieving Large-Scale Computing Capabilities.** IBM Sunnyvale, CA: AFIPS spring joint computer conference, 1967.

ANDREWS, G. R. **Foundations of Multithreaded, Parallel, and Distributed Programming**. University of Arizona, USA: Wesley, 2000.

ARIAS, M.; CHAKRABORTY R. **Functional Kotlin:** Extend your OOP skills and implement functional techniques in Kotlin and Arrow. Birmingham, UK: Packt, 2018.

ARPACI-DUSSEAU, R. H.; ARPACI-DUSSEAU, A. C. **Operating Systems:** Three Easy Pieces. [S.l.], Arpaci-Dusseau Books, v. 1, c. 7. 2018.

BLOCH, J., **Effective Java:** Best practices for the Java Platform. 3.ed. USA: Addison-Wesley, 2018. p.311-338.

BORNSTEIN, Dan. **Dalvik VM Internals**. [S.l.: s.n.], 2008.

COFFMAN, E. G.; ELPHICK, M. J.; SHOSHANI, A. System Deadlocks. **Computing Surveys**, [S.l.: s.n], v. 3, n. 2, p. 67-77. 1971.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T.; BLAIR, G. **Distributed Systems:** Concepts and Design. 5.ed. USA: Addison-Wesley, 2012. p. 279-318.

DIMARZIO, Jerome J. F. **Android™:** A Programmer's Guide. USA: McGraw-Hill, 2008.

DUBOIS, Michel; SCHEURICH, Christoph. Synchronization, Coherence, and Event Ordering in Multiprocessors. **IEEE,** University of Southern California, USA, [s.n.], Fevereiro de 1988.

EIJKHOUT, Victor. **Introduction to High Performance Scientific Computing**, Texas Advanced Computing Center, USA, [s.n.], 2014. c. 2.

GÖETZ, B.; PEIERLS, T.; BLOCH, J.; BOWBEER, J.; HOLMES, D.; LEA, D.; **Java Concurrency in Practice**. [S.l.], Addison-Wesley, 2006.

GOLUB, G. H.; LOAN, C.F. Van, **Matrix Computations** 4.ed. Baltimore, USA: Johns Hopkins University Press, 2013. p.2-62.

GOOGLE AND OPEN HANDSET ALLIANCE. **Android API Guide**. Disponível em: <https://developer.android.com/guide/index.html>. Acesso em: 06 de novembro de 2019.

GOOGLE AND OPEN HANDSET ALLIANCE. **Platform Architecture**, Disponível em: < https://developer.android.com/guide/platform>. Acesso em: 30 de novembro de 2019.

GOOGLE DEVELOPER TRAINING TEAM. **Android Developer Fundamentals Course**: Concept Reference [S.l.], 2016.

GUSTAFSON, John L. Reevaluating Amdahl's Law. **Communication of the ACM**. Maio de 1988. Disponível em: < http://www.johngustafson.net/pubs/pub13/amdahl.htm >. Acesso em: 17 de Dezembro de 2019.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Archtecture:** A Quantitative Approach. 6.ed. Cambridge, MA, USA: Morgan Kauffmann, 2019. p. 368-442.

HERLIHY, Maurice; SHAVIT, Nir. **The Art of Multiprocessor Programming.** Burlington, MA, USA: Morgan Kaufmann, 2008.

JAIN, R. K. **The Art of Computer Systems Performance Analysis:** techniques for experimental design, measurement, simulation, and modeling. 1.ed. [S.l.]: Willey, 1991.

JEMEROV, Dmitry; ISAKOVA, Svetlana. **Kotlin in Action.** [S.l.], Manning publications, 2016.

KAUR, Parmjit; SHARMA, Sumit. Google Android a Mobile Platform: A Review. **RAECS UIET**., Chandigarh, Índia. Março de 2014.

LENGURE, Swapnil A. Android Operating System and Contemporary, A Brief Comparison. **International Journal of Electrical and Electronics Research**, [S.l.: s.n.], 2015.

LIN, Yu; RADOI, Cosmin; DIG, Danny. Retrofitting Concurrency for Android Applications Through Refactoring. **ACM Comput. Surv**. Hong Kong, China, Novembro de 2014.

LYNCH, Nancy A., **Distributed Algorithms.** San Francisco, CA, USA: Morgan Kaufmann, 1996. p. 397 – 448.

MAIYA, Pallavi; KANADE, Aditya; MAJUMDAR, Rupak. Race Detection for Android Applications. **ACM Comput. Surv.**, Edinburgh, UK, Junho de 2014.

MARTIN, Milo; ROTH, Amir. **CIS 501: Computer Architecture**: Unit 4: Performance & Benchmarking, University of Pennsylvania, USA, 2015.

MEGALI, Tin. **Understanding Concurrency on Android Using HaMeR**, Setembro de 2016. Disponível em: < https://code.tutsplus.com/tutorials/concurrency-on-android-using-hamer-framework--cms-27129>. Acesso em: 30 de Novembro de 2019.

MEIKE, G. Blake. **Android Deep Dive:** Android Concurrency. USA: Addison-Wesley, 2016.

MOSKALA, Marcin; WOJDA, Igor. **Android Development with Kotlin**. Birmingham, UK: Packt, 2017.

PARHAMI, Berhrooz. **Introduction to Parallel Processing:** Algorithm and Architectures. University of California, Santa Barbara, CA, USA: Kluwer Academic, 2002.

SAMUEL, Stephen; BOCUTIU, Stefan., **Programming Kotlin**: Familiarize yourself with all of Kotlin's features with this in-depth guide. Birmingham, UK: Packt, 2017.

SCHILDT, Herbert. **Java: The Complete Reference**: Comprehensive Coverage of the Java Language. 9.ed. USA: McGraw-Hill, 2014.

SILBERSCHATZ, Abraham; GALVIN, Peter Baer; GAGNE, Greg. **Operating System Concepts**. 9.ed. USA: Wiley, 2013.

SUN, Xian-He; NI, Lionel M., Another View on Parallel Speedup. **ACM Comput. Surv**, New York, USA, Novembro de 1990.

TANG, H.; WU, G.; WEI, J.; ZHONG, H. Generating Tests Cases to Expose Concurrency Bugs in Android Applications. **ACM Comput. Surv**, Singapore, Setembro de 2016.

TIS COMMITTEE, **Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification**. [S.l.:s.n.], 1995. p.1-23.

WITTE, Robert S.; WITTE, John S. **Statistics**. 9.ed. USA: Willey, 2010.

YOUHAN, Zeena. **Dining Philosopher Problem and Solution**. Maio de 2019. Disponível em: < https://medium.com/@zinayouhan33/dining-philosopher-problem-and-solution-9a273a5fa614>. Acesso em: 30 de Novembro de 2019.

APÊNDICE A – EXECUÇÕES REMOTAS

O objetivo dos experimentos com execuções remotas é de validar as observações realizadas nas seções anteriores. os experimentos foram executados também em outros dispositivos disponíveis pelo Firebase Test Lab, que é um *data center* com dispositivos que podem ser utilizados para teste de aplicativos de forma remota. A seguir os dispositivos utilizados para a execução remota são descritos. Após a descrição desses dispositivos, são apresentadas algumas observações dos resultados desses experimentos, observações que foram realizadas com objetivo de confirmar ou refutar as observações feitas nas seções anteriores desse capítulo.

Os dispositivos selecionados para as execuções remotas foram:

1. **Pixel 2 XL**:
    - CPUs: Oito núcleos físicos organizados em dois conjuntos (4x2.35 GHz Kryo & 4x1.9 GHz Kryo);
    - RAM: 4GB;
    - Android 8.0.
2. **Samsung Galaxy Note9:**
    - CPU: Oito núcleos físicos organizados em dois conjuntos (4x2.8 GHz Kryo 385 Gold & 4x1.7 GHz Kryo 385 Silver);
    - RAM: 6 GB;
    - Android 9.0.
3. **Motorola Nexus 6:**
    - CPU: Quatro núcleos físicos (2.7 GHz Krait 450);
    - RAM: 3 GB;
    - Android 6.0.

Para o problema da multiplicação de matrizes foi possível verificar em todos os dispositivos o aumento no tempo de execução quando utilizando o mecanismo de Kotlin coroutines se comparado com os demais. Esse tempo maior, contudo, foi proporcionalmente menor quando executado no dispositivo 3 para matrizes 128x128 (o dispositivo com maior tempo de execução dentre os dispositivos usados para testes). De mesmo modo, foi possível verificar que em quase todos os casos Kotlin coroutines se mantém sendo o mecanismo com speedup mais próximo do ideal. Uma exceção a isso, no entanto, ocorreu ao executar com o

dispositivo 1 com matrizes de 512x512 onde tanto Kotlin coroutines quanto o framework HaMeR apresentaram um speedup menor do que os outros dois mecanismos de concorrência, esse fenômeno, contudo, não aconteceu ao executar com matrizes de 128x128 e de 256x256 onde Kotlin coroutines se manteve com o speedup mais próximo do ideal. Assim como nos testes mostrados na seção 4.1, o mecanismo de Threads introduziu uma alta sobrecarga para as matrizes menores, o que foi especialmente sentido no dispositivo 2 (o dispositivo com menor tempo de execução dentre os testados), onde a execução com 64 tarefas foi 3 vezes mais lenta do que a execução com 1 tarefa. Tudo isso contribui para as afirmações realizadas na seção 4.1 de que Threads, apesar de possuir um bom desempenho para tarefas longas, possui uma sobrecarga alta quando as tarefas são de curta duração, sendo mais recomendável para tarefas mais longas, enquanto que Kotlin coroutines é mais recomendável para tarefas curtas, pois mesmo nesses casos consegue aproveitar de recursos paralelos possuindo o speedup mais próximo do ideal.

Quanto as execuções dos experimentos com a soma concorrente, foi possível confirmar o baixo tempo de Kotlin coroutines, assim como observado na seção 4.2, especialmente para os dispositivos mais rápidos (dispositivo 1 e dispositivo 2) reforçando sua indicação para tarefas de curta duração. De mesmo modo, Thread Pool apresentou o menor dos tempos para a maioria dos cenários possuindo maior vantagem sobre Kotlin coroutines no dispositivo 3, mas sendo superada por Threads em alguns dos cenários onde se usou 1 tarefa. Um elemento que apareceu em todos os testes remotos e que apareceu apenas parcialmente nos experimentos da seção 4.2. Foi o fato da execução com Threads, ao utilizar 256 tarefas foi mais lento do que todos os outros mecanismos possuindo um aumento muito alto do tempo de execução ao passar de 16 para 256 tarefas. A observação realizada na seção 4.2 sobre o uso de Threads com barreiras introduzir uma sobrecarga alta para a sincronização dos níveis, somente se confirmou totalmente no dispositivo 3, nos outros dois dispositivos, essa sobrecarga é percebida somente ao utilizar 16 ou 256 tarefas. Pois o tempo desse mecanismo foi similar aos demais ao utilizar 1 tarefa, possuiu o maior tempo ao utilizar 16 tarefas (com exceção do dispositivo 2 ao somar 2^{18} números) e possui seu tempo ultrapassado por Threads ao utilizar 256 tarefas.

Ao executar remotamente os experimentos do problema dos filósofos, foi possível confirmar o comportamento de Kotlin coroutines com blocos Synchronized de possuir um coeficiente de variação muito maior para as execuções com 5 e com 11 filósofos. Esse valor se mantém na mesma ordem de grandeza quando executa com 51 filósofos enquanto os demais aumentam seu coeficiente de variação consideravelmente. De modo geral, o comportamento

dos mecanismos de concorrência se manteve semelhante entre si e com as observações feitas na seção 4.3, com exceção do framework HaMeR que, no dispositivo 2 ainda foi mais justo que os demais para 51 filósofos, porém com menos diferença do que os resultados obtidos na seção 4.3 e para o dispositivo 1 foi mais justo ao ser utilizado com semáforos e com locks, mas não quando usado com blocos synchronized, ainda assim se manteve o mecanismo mais justo, em média, para os cenários de execução com 51 filósofos.

Para as execuções do download de arquivos, os resultados do dispositivo 2 foram semelhantes aos obtidos na seção 4.4, mostrando pequena vantagem de Kotlin coroutines para imagens maiores. Contudo, o dispositivo 1 apresentou resultado oposto, com o tempo de Kotlin coroutines sendo semelhante aos demais para imagens pequenas, mas possuindo o pior tempo para a maior imagem. No dispositivo 3, por sua vez, esse mecanismo possuiu um tempo mediano. Essa inconsistência nos resultados indica que as comparações de tempos para esse problema são inconclusivas para a avaliação dos mecanismos de concorrência que não apresentam diferenças consistentes em performance para esse problema.

As execuções do problema dos produtores-consumidores confirmaram a superioridade do uso de semáforos e de locks sobre os outros dois mecanismos. Como foi notado na seção 4.5, semáforos e locks possuíram desempenhos muito próximos em muitos casos com leve vantagem para o uso de semáforos, esses dois são seguidos em questão de desempenho pelas execuções com blocos synchronized e, por último, variáveis atômicas, que, somente em casos raros não ficou com o pior desempenho. Também foi possível notar a melhora do desempenho de variáveis atômicas quando a quantidade de tarefas produtoras e consumidoras foi igual. Essas exceções mencionadas ocorreram nos casos em a quantidade de tarefas produtoras em execução era igual a quantidade de tarefas consumidoras, onde o desempenho de variáveis atômicas foi próximo do desempenho de blocos synchronized superando-o em poucos casos.