INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Análise de Ferramentas para Deteção de Problemas de Concorrência em Java: Um Estudo Comparativo

LUÍS FERNANDO DA SILVA NUNES VIGÁRIO
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Análise de Ferramentas para Deteção de Problemas de Concorrência em Java: Um Estudo Comparativo

Luís Fernando da Silva Nunes Vigário

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Orientador: Luís Miguel Nogueira

Porto, junho 2025

Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, 26 de junho de 2025

Resumo

A programação concorrente em Java é fundamental para o desenvolvimento de sistemas modernos, especialmente em ambientes que exigem alta performance, escalabilidade e eficiência no uso de recursos. Apesar da robustez das ferramentas e *frameworks* existentes, desafios como *race conditions*, *deadlocks* e violações de atomicidade continuam a impactar negativamente a fiabilidade e o desempenho dos sistemas. A presente dissertação analisa as soluções disponíveis, identificando limitações em termos de precisão, escalabilidade e facilidade de utilização.

Através de uma revisão do estado da arte, foram exploradas ferramentas como *Java Pathfinder*, *ThreadSanitizer*, e *frameworks* como *Executors Framework* e *Fork-Join Framework*. Paralelamente, propõem-se melhorias como a adoção de algoritmos de *machine learning* para deteção preditiva de problemas, técnicas para mitigar *state space explosion*, e interfaces simplificadas que promovam a acessibilidade e a integração com arquiteturas modernas.

Os resultados obtidos apontam para avanços significativos na gestão de concorrência, com impacto positivo em áreas críticas como sistemas financeiros e plataformas de *microservices*. Este trabalho destaca ainda a relevância da deteção proativa de erros e da escalabilidade em sistemas distribuídos, propondo direções futuras que incluem a adaptação a tecnologias emergentes.

Palavras-chave: Programação concorrente, Java, *Race conditions, Deadlocks*, Ferramentas de deteção, Violações de atomicidade.

Abstract

Concurrent programming in Java is fundamental for the development of modern systems, particularly in environments requiring high performance, scalability, and efficient resource utilization. Despite the robustness of existing tools and frameworks, challenges such as race conditions, deadlocks, and atomicity violations continue to negatively impact system reliability and performance. This dissertation critically analyzes the available solutions, identifying limitations in terms of precision, scalability, and ease of use.

Through a detailed review of the state of the art, tools such as Java Pathfinder, ThreadSanitizer, and frameworks like Executors Framework and Fork-Join Framework were explored. In parallel, improvements are proposed, including the adoption of machine learning algorithms for predictive problem detection, advanced techniques to mitigate state space explosion, and simplified interfaces that promote accessibility and integration with modern architectures.

The results achieved point to significant advancements in concurrency management, with a positive impact on critical areas such as financial systems and microservices platforms. This work also highlights the relevance of proactive error detection and scalability in distributed systems, proposing future directions that include adaptation to emerging technologies such as serverless computing.

Keywords: Concurrent programming, Java, Race conditions, Deadlocks, Detection tools, Atomicity violations.

Agradecimentos

Primeiramente quero agradecer ao Instituto Superior de Engenharia do Porto por me acolher durante o meu percurso académico e aos docentes com quem tive o prazer de compartilhar a sala de aula, em particular ao professor Luís Nogueira que me acompanhou nesta última etapa desta minha jornada.

Aos amigos que trouxe do ensino secundário, da licenciatura e àqueles que fiz ao longo destes anos de mestrado, o meu especial obrigado por me acompanharem tanto nos momentos felizes, como nos de maior aperto. Sem o vosso apoio e amizade não teria chegado onde estou agora.

Quero agradecer à minha avó Ana, que me criou em tenra idade e sempre me ajudou quando precisei.

Quero também agradecer à minha irmã, Íris, que apesar das nossas diferenças, tento sempre ser um exemplo para ela e assim faz-me crescer como pessoa.

Quero deixar o meu obrigado à Inês, nos momentos mais difíceis nunca me deixou desistir e sempre incentivou a fazer e a ser melhor, sendo sempre um porto de abrigo.

Por fim quero agradecer aos meus pais, que embarcaram nesta aventura comigo. Apesar de nunca terem experimentado a vida académica, sempre me deram espaço para eu a conseguir aproveitar ao máximo. Se sou o que sou hoje devo-lhes a eles por todos os ensinamentos e virtudes que me passaram.

Orientador: Luís Miguel Nogueira

Porto, junho 2025

Índice

Lista de Figuras

Lista de Tabelas

Lista de Excertos de Código

Acrónimos e Símbolos

Lista de Acrónimos

ACID	*Atomicity, Consistency, Isolation, Durability*
API	*Application Programming Interface*
CAS	*Compare-and-Swap*
CI/CD	*Continuous Integration and Continuous Delivery*
CSV	*Comma Separated Values*
CPU	*Central Processing Unit*
HTTP	*Hypertext Transfer Protocol*
ISEP	*Instituto Superior de Engenharia do Porto*
IoT	*Internet of Things*
JIT	*Just-In-Time*
JMM	*Java Memory Model*
JPF	*Java Pathfinder*
JVM	*Java Virtual Machine*
MEI-ES	*Mestrado Engenharia Informática – Engenharia Software*
MTTR	*Mean Time To Resolution*
REST	*Representational State Transfer*
WBS	*Work Breakdown Structure*

1 Introdução

1.1 Contexto

A programação concorrente é uma área de desenvolvimento de sistemas modernos, caracterizada pela execução simultânea de múltiplas tarefas, com o objetivo de melhorar a eficiência e o desempenho das aplicações. Este paradigma é especialmente relevante em contextos onde a velocidade de processamento e a capacidade de resposta são fatores críticos, como em sistemas distribuídos, aplicações financeiras, arquiteturas de *microservices* e ambientes baseados em *cloud computing*. Contudo, a programação concorrente apresenta desafios significativos, como *race conditions*, *deadlocks* e violações de atomicidade, que podem comprometer a integridade e a consistência dos sistemas.

Com o aumento da complexidade das aplicações e da procura por escalabilidade, a necessidade de ferramentas, técnicas e *frameworks* eficazes para gerir a concorrência tornou-se evidente. Estes desafios exigem soluções robustas que garantam a correta gestão de *threads* e o acesso seguro a recursos partilhados, minimizando falhas e otimizando a performance em ambientes críticos. Assim, a programação concorrente não é apenas uma necessidade técnica, mas também um domínio estratégico para a inovação e o desenvolvimento de sistemas confiáveis e eficientes.

1.2 Problema

A crescente necessidade de melhor desempenho nos sistemas computacionais modernos tem impulsionado a utilização de processamento paralelo e concorrente em aplicações Java. Embora este paradigma traga melhorias significativas em eficiência e capacidade de processamento, introduz também desafios técnicos complexos. Problemas como *race conditions*, *deadlocks* e violações de atomicidade, a serem abordadas na seção 3.1.1, são inerentes à programação concorrente, resultando em comportamentos imprevisíveis e difíceis de diagnosticar, comprometendo a fiabilidade e a eficiência dos sistemas.

Para mitigar estas dificuldades, foram desenvolvidas diversas ferramentas e *frameworks* para a deteção de problemas de concorrência em programas Java. Contudo, estas soluções apresentam limitações, como falsos positivos e negativos frequentes, bem como dificuldades de escalabilidade em sistemas de grande porte. Adicionalmente, a harmonização entre a deteção de problemas e a sua resolução prática constitui um desafio significativo, especialmente em ambientes diversificados onde as questões de concorrência se manifestam de formas distintas.

A ausência de *benchmarks* padronizados que representem cenários do mundo real dificulta ainda mais a avaliação abrangente das ferramentas existentes. Este cenário evidencia a necessidade de uma solução universal que aborde, de forma eficaz, todos os aspetos relacionados com a deteção e resolução de problemas de concorrência, especialmente em sistemas complexos e escaláveis.

Assim, o problema analisado não possui uma solução óbvia e permanece parcialmente abrangido pelas normas e códigos atuais, que fornecem orientações gerais, mas falham em atender às exigências dos sistemas modernos. As consequências deste problema são amplas, afetando setores críticos como o financeiro, o da saúde e o das telecomunicações, o que reforça a necessidade de desenvolver soluções inovadoras para detetar e gerir problemas de concorrência em aplicações Java.

1.3 Objetivos

O principal objetivo deste trabalho é analisar e comparar ferramentas para a deteção de problemas de concorrência em Java, propondo melhorias que aumentem a eficiência e a precisão na identificação e resolução de *race conditions*, *deadlocks* e violações de atomicidade. Para alcançar este objetivo, foram formuladas as seguintes perguntas de investigação, que orientam o desenvolvimento da dissertação:

1. Quais são as principais limitações e desafios das ferramentas existentes para a deteção de problemas de concorrência em Java, nomeadamente *race conditions*, *deadlocks* e violações de atomicidade, especialmente em sistemas de grande escala?
2. Como varia a precisão, a cobertura e o desempenho das ferramentas de deteção de concorrência em Java em diferentes cenários de execução e que impacto têm os falsos positivos e os falsos negativos na eficácia dessas ferramentas?
3. Como podem as ferramentas de deteção de problemas de concorrência em Java ser otimizadas, através de novas técnicas ou melhorias nas abordagens atuais, para aumentar a eficiência, escalabilidade e precisão, reduzindo os falsos positivos e negativos, especialmente em ambientes com elevada concorrência?
4. Quais são os impactos das melhorias nas ferramentas de deteção de concorrência no ciclo de vida de desenvolvimento de software e na capacidade de assegurar a fiabilidade em sistemas críticos?

Para responder a estas perguntas, foram delineados os seguintes objetivos específicos:

- **Revisão de ferramentas e *frameworks* existentes:** realizar uma revisão detalhada das principais ferramentas e *frameworks* utilizados na deteção de problemas de concorrência em Java, como por exemplo, o *Java Pathfinder* (JPF), o *ThreadSanitizer* e o *FindBugs* com o *Concurrency Bug Detector*.
- **Avaliação comparativa:** avaliar a eficácia destas ferramentas em diferentes cenários de concorrência, utilizando *benchmarks* reconhecidos e estudos de caso reais. A avaliação incluirá uma análise comparativa em termos de cobertura, precisão, desempenho e facilidade de utilização.
- **Identificação de limitações:** identificar as principais limitações das ferramentas atuais, com foco nos desafios associados à deteção de problemas em sistemas complexos, bem como na ocorrência de falsos positivos e negativos.

Adicionalmente, planeia-se implementar as otimizações propostas em um ou mais *frameworks* de código aberto, acompanhadas de testes comparativos para validar as melhorias em termos de precisão e eficiência. Caso a implementação prática não seja possível, o trabalho concentrar-se-á numa análise comparativa detalhada das ferramentas existentes, utilizando diversas métricas para fundamentar a avaliação.

1.4 Gestão de Competências

O diagnóstico de competências é uma etapa essencial para identificar as áreas-chave a serem desenvolvidas ao longo do percurso académico e profissional. Este processo permite avaliar de forma crítica as competências existentes, destacar pontos fortes e reconhecer fraquezas que possam ser trabalhadas para potenciar o desempenho. No contexto desta dissertação, foi realizada uma autoavaliação, com o objetivo de delinear ações concretas para o desenvolvimento de competências específicas, essenciais para o sucesso tanto na execução deste trabalho como em futuros desafios profissionais.

1.4.1 Diagnóstico de Competências

A autoavaliação de competências é uma análise detalhada de um conjunto abrangente de competências, classificando-as em dois critérios principais: a sua importância no contexto do percurso académico e profissional, e o nível atual de proficiência com base na autoavaliação.

A Tabela 1 apresenta os resultados desta avaliação, abrangendo competências técnicas e comportamentais. Este diagnóstico permitiu delinear um plano de ações direcionado para o desenvolvimento das competências.

Tabela 1 - Diagnóstico de Competências

Competências	**Importância**	**Autoavaliação**
Análise e resolução de problemas	4	4
Aprendizagem ao longo da vida	4	4
Trabalho em equipa e colaboração	5	4
Motivação para a excelência	4	4
Adaptabilidade e flexibilidade	5	5
Ética e responsabilidade social	4	4
Domínio de línguas estrangeiras	3	3
Competências técnicas da área específica de conhecimento	4	3
Tomada de decisão	4	3
Gestão do tempo	4	3
Capacidade para assumir riscos	4	3
Comunicação oral	5	3
Comunicação escrita	3	3
Resiliência	4	4
Escuta ativa	4	4
Relacionamento interpessoal	4	5
Liderança	5	4
Criatividade e inovação	3	4
Adaptação e flexibilidade	5	5
Pensamento critico	4	4
Planeamento e organização	4	4
Gestão das emoções	4	4
Negociação	4	4
Empatia	3	4
Assertividade	4	4
Gestão do stress	4	3
Proatividade	4	4

1.4.2 Competências mais desenvolvidas

Entre as competências avaliadas no diagnóstico, destacam-se o relacionamento interpessoal e a adaptação e flexibilidade como as mais desenvolvidas. Estas competências apresentam uma autoavaliação máxima, refletindo a capacidade de estabelecer conexões positivas e produtivas com os outros, bem como a habilidade de se ajustar eficazmente a diferentes contextos e situações.

O relacionamento interpessoal é fundamental para promover uma comunicação eficaz, fortalecer laços em ambientes colaborativos e contribuir para um clima de trabalho harmonioso. Esta competência é especialmente relevante em contextos de trabalho em equipa e projetos multidisciplinares, onde a interação constante com diferentes perfis é essencial para o sucesso.

Por outro lado, a adaptação e flexibilidade, demonstram a capacidade de responder proactivamente a mudanças e desafios, uma característica indispensável num ambiente profissional dinâmico. A elevada pontuação nesta competência reflete a aptidão para lidar com imprevistos e ajustar estratégias rapidamente, garantindo uma resposta eficaz às necessidades do momento.

Estas competências são pilares importantes para sustentar o desenvolvimento das demais áreas, servindo como alicerces para alcançar um desempenho equilibrado e eficaz em projetos académicos e profissionais.

1.4.3 Competências a desenvolver

Apesar de apresentar um desempenho consistente em várias competências, o diagnóstico revelou algumas áreas prioritárias de desenvolvimento: comunicação oral, domínio de línguas estrangeiras e competências técnicas da área específica de conhecimento. Estas competências são essenciais para melhorar a eficácia e a confiança em situações críticas, e o desenvolvimento direcionado destas áreas será uma prioridade ao longo deste projeto e em futuros contextos profissionais.

A comunicação oral é indispensável para apresentações e interações profissionais. Para aprimorá-la, será explorada a participação em grupos de debate ou oratória, permitindo-me praticar o discurso em público e receber feedback construtivo. Também serão realizados ensaios regulares de apresentações formais, gravando as sessões para avaliar e melhorar aspetos como clareza, tom e linguagem corporal.

O domínio de línguas estrangeiras é fundamental em contextos internacionais e para aceder a um maior volume de conhecimento técnico. Nesse sentido, será priorizada a frequência de cursos de línguas, com foco no desenvolvimento de habilidades práticas de comunicação. Para complementar, serão explorados conteúdos técnicos em línguas estrangeiras, como podcasts e vídeos, reforçando a compreensão e o vocabulário técnico.

Por fim, o desenvolvimento das competências técnicas da área específica de conhecimento será abordado por meio da participação em workshops técnicos focados na programação concorrente e no uso de ferramentas associadas. Além disso, serão realizados projetos práticos, desenvolvendo implementações concretas que consolidem conceitos e técnicas adquiridas, garantindo uma aplicação mais robusta das aprendizagens no contexto do projeto.

1.5 Planeamento do Trabalho

Neste subcapítulo, são apresentados os principais instrumentos utilizados para organizar e monitorizar o progresso da dissertação. Foi desenvolvido um *Project Charter* que define os objetivos principais, as partes interessadas e as restrições iniciais do projeto. A *Work Breakdown Structure (WBS)* detalha as etapas e entregáveis, dividindo o trabalho em componentes geríveis.

Para avaliar riscos potenciais e delinear estratégias de mitigação, foi utilizada uma matriz de risco, permitindo uma análise preventiva das ameaças ao sucesso do projeto. Por fim, o *Gantt Chart* fornece uma visão cronológica das atividades, estabelecendo prazos claros e permitindo o acompanhamento eficaz do progresso ao longo do tempo.

1.5.1 Project Charter

O Project Charter é um documento essencial para delinear as bases do projeto, identificando os seus objetivos, escopo, partes interessadas e entregáveis. No contexto desta dissertação, o *Project Charter* estabeleceu como objetivo principal a análise e comparação de ferramentas para a deteção de problemas de concorrência em Java, propondo otimizações que aumentem a eficiência e precisão na identificação de *race conditions*, *deadlocks* e violações de atomicidade.

O documento em anexo, destaca o crescente desafio de gerir problemas de concorrência em sistemas modernos e a necessidade de ferramentas que superem as limitações das soluções atuais. Também foram definidas as questões de investigação, como a avaliação das principais limitações das ferramentas existentes, as variações de desempenho em diferentes cenários e o impacto de melhorias no ciclo de desenvolvimento de *software*.

Os benefícios esperados incluem uma redução significativa no tempo e custo associados à concorrência em sistemas computacionais. O *Project Charter* estima uma diminuição de aproximadamente 2,5 meses num ciclo de 12 meses e uma redução de custos na ordem de €175.000 a €295.000 por ano. Entre os entregáveis listados, destacam-se o planeamento do projeto, o relatório final da dissertação e documentos intermédios que consolidam a pesquisa realizada.

Por fim, as restrições e riscos foram também delineados, como a possibilidade de descontinuação de bibliotecas e ferramentas analisadas ou alterações significativas nas versões futuras de Java, que podem tornar a pesquisa obsoleta. O documento completo encontra-se anexado (Anexo A) como suporte adicional, detalhando as etapas e compromissos definidos para a execução deste projeto.

1.5.2 *Work Breakdown Structure*

A *WBS* é uma ferramenta essencial para organizar e estruturar as atividades de um projeto, permitindo uma gestão eficiente e o acompanhamento das etapas ao longo do seu ciclo de vida. A *WBS* apresentada, que será anexada como suporte adicional (Anexo B), reflete a divisão hierárquica do trabalho desenvolvido nesta dissertação, detalhando cada entrega principal e os respetivos subcomponentes.

A *WBS* está organizada em cinco níveis principais, correspondentes às secções-chave da dissertação: Introdução, Estado da Arte, Implementação, Avaliação e Resultados, e Conclusões e Trabalhos Futuros. Cada um destes níveis inclui subcomponentes que representam as entregas específicas associadas às atividades planeadas.

No nível da Introdução, destacam-se elementos como o contexto, o problema de investigação, os objetivos, o planeamento do trabalho e a estrutura da dissertação, formando a base conceptual do projeto. Já no Estado da Arte, foram mapeadas atividades como a definição da metodologia, a análise das tecnologias relacionadas, a revisão dos trabalhos relacionados, a comparação de tecnologias e a formulação de propostas de melhoria, que fundamentam o enquadramento teórico da investigação.

A Implementação está subdividida em três entregas principais: ferramentas e ambientes utilizados, desenvolvimento do protótipo e integração em sistemas de teste. Estas etapas correspondem à materialização prática das soluções propostas ao longo da dissertação.

No que se refere à Avaliação e Resultados, a *WBS* inclui a comparação de desempenho e a discussão dos resultados obtidos, com o objetivo de validar as propostas desenvolvidas e demonstrar a sua eficácia. Por fim, o nível de Conclusões e Trabalhos Futuros abrange o resumo dos resultados, a identificação de limitações do trabalho e as propostas para continuidade da investigação, encerrando o ciclo do projeto.

1.5.3 Matriz de Risco

A matriz de risco é uma ferramenta fundamental para a identificação, avaliação e gestão de riscos que podem impactar o sucesso do projeto. Este instrumento permite categorizar os riscos com base na probabilidade de ocorrência e no impacto potencial, possibilitando a definição de estratégias de mitigação apropriadas.

Na matriz representada no Anexo C, foram identificados dois riscos principais relacionados ao tema da dissertação. O primeiro risco é a **depreciação das bibliotecas e ferramentas analisadas**, resultando na possibilidade de estas se tornarem obsoletas, o que comprometeria a relevância da pesquisa realizada. O segundo risco refere-se aos **desenvolvimentos futuros nas versões de Java**, onde atualizações significativas podem abordar os problemas analisados, também levando à obsolescência do trabalho desenvolvido.

Ambos os riscos foram avaliados com uma probabilidade de 2 (em uma escala de 1 a 5) e um impacto de 5, resultando num *PI Score* (probabilidade x impacto) de 10. Este valor reflete uma necessidade moderada de atenção, mas sem urgência imediata. A resposta definida para ambos os casos foi a aceitação dos riscos, com monitorização contínua. Essa abordagem considera que os impactos só ocorrerão caso sejam lançadas novas versões ou atualizações relevantes, sendo necessário acompanhar lançamentos de novas *releases* e avaliar as alterações realizadas.

O efeito esperado, caso nenhum plano de ação seja implementado, é a potencial obsolescência da pesquisa, exigindo atualizações futuras para garantir a sua pertinência. A descrição das respostas ao risco inclui a monitorização periódico de publicações oficiais sobre Java e ferramentas relacionadas, possibilitando a identificação de alterações que possam afetar diretamente os resultados do projeto.

1.5.4 Gantt Chart

O *Gantt Chart* é uma ferramenta crucial para o planeamento e acompanhamento de projetos, permitindo visualizar a distribuição temporal das tarefas, as dependências entre elas e os seus prazos de execução. No contexto desta dissertação, o *Gantt Chart* fornece uma visão detalhada do progresso do trabalho, organizado em marcos principais e subtarefas e que pode ser consultado no Anexo D.

A estrutura do projeto está dividida em cinco etapas principais: Introdução, Estado da Arte, Desenvolvimento, Avaliação de Resultados e Conclusões e Entrega. Cada uma dessas etapas é subdividida em tarefas específicas, com durações e interdependências claramente definidas. A primeira etapa, Introdução, inclui atividades como a definição do contexto, problema de investigação e objetivos, estabelecendo as bases do projeto. Esta etapa reflete uma abordagem metódica e planeada.

O Estado da Arte, uma etapa fundamental para consolidar o enquadramento teórico do trabalho. Esta etapa inclui a análise de metodologias, tecnologias relacionadas e trabalhos prévios, culminando com a comparação de ferramentas e propostas de melhoria. O *Gantt Chart* reflete uma sobreposição temporal entre algumas destas tarefas, destacando a importância de uma gestão eficiente dos recursos e tempo disponíveis.

A fase de Desenvolvimento é dedicada à implementação prática das propostas da dissertação. Esta etapa inclui o desenvolvimento do protótipo, a integração com sistemas de teste e a realização de testes para validar os objetivos definidos.

Na etapa de Avaliação de Resultados serão realizados testes detalhados para análise de cenários, desempenho e resultados temporais, assegurando que as soluções propostas atendem aos critérios definidos.

Por fim, o *Gantt Chart* detalha a fase de Conclusão e Entrega. Esta etapa inclui a correção e revisão final do documento, bem como a preparação para a apresentação, garantindo a entrega de um trabalho e fundamentado.

1.6 Estrutura da Dissertação

A presente dissertação está organizada em seis capítulos, estruturados de forma a abordar progressivamente o tema, desde a contextualização e identificação do problema até à análise dos resultados e conclusões.

O **primeiro capítulo**, intitulado *Introdução*, apresenta o contexto do trabalho, descrevendo o problema da investigação, os objetivos propostos e as questões de pesquisa. Além disso, inclui uma autoavaliação de competências relevantes para a execução do trabalho, bem como o planeamento detalhado do projeto.

O **segundo capítulo**, *Metodologia*, detalha os critérios utilizados para avaliar as ferramentas, técnicas e *frameworks* de programação concorrente, incluindo a estratégia de pesquisa, os *benchmarks* selecionados e as métricas adotadas para análise. Este capítulo serve de base para assegurar a consistência e rigor das avaliações realizadas.

O **terceiro capítulo**, *Estado da Arte*, apresenta uma análise aprofundada das ferramentas, técnicas e *frameworks* existentes na área de programação concorrente. Este capítulo fornece uma visão abrangente das soluções disponíveis, analisando as suas funcionalidades, aplicações e limitações, com base nos critérios definidos no capítulo anterior.

O **quarto capítulo**, *Comparação e Propostas de Melhoria*, apresenta os resultados das análises comparativas realizadas com base nos critérios estabelecidos. Além disso, são propostas melhorias destinadas a otimizar a deteção de problemas de concorrência, avaliando os impactos dessas soluções em cenários reais e simulados.

O **quinto capítulo**, *Desenvolvimento e Resultados*, descreve a implementação prática das propostas de melhoria, incluindo detalhes técnicos sobre o desenvolvimento do protótipo e a integração em sistemas de teste. Este capítulo apresenta ainda uma análise detalhada dos resultados obtidos, discutindo as vantagens e implicações das soluções propostas.

Finalmente, o **sexto capítulo**, *Conclusões*, resume as principais contribuições deste trabalho, destacando os avanços realizados na deteção de problemas de concorrência em Java. Além disso, identifica as limitações do estudo e sugere direções para investigações futuras, de forma a ampliar o conhecimento na área e fomentar o desenvolvimento de novas soluções.

2 Metodologia

A metodologia adotada nesta dissertação visa assegurar uma análise rigorosa e fundamentada de ferramentas, técnicas e *frameworks* relacionadas à programação concorrente em Java. Este subcapítulo descreve os critérios utilizados para avaliar as soluções estudadas, bem como a estratégia de pesquisa delineada para identificar as fontes mais relevantes. Além disso, detalha-se o processo iterativo de pesquisa, que combinou a utilização de palavras-chave em bases de dados científicas e a exploração do método de pesquisa tradicional, de forma a garantir uma cobertura abrangente e coesa do tema.

2.1 Estratégia de Pesquisa

A estratégia de pesquisa adotada permitiu identificar as fontes mais relevantes e recentes relacionadas à programação concorrente em Java. Este processo foi dividido em três etapas principais: definição de palavras-chave, seleção de bases de dados e análise iterativa de referências.

A primeira etapa consistiu na identificação de palavras-chave relevantes para abordar os principais tópicos relacionados ao tema. Termos como *"Java"*, *"Reactive Programming"*, *"Thread"*, e *"Concurrency", "Bug Detection"* são exemplos de palavras-chave que foram utilizadas para cobrir tópicos de ferramentas, técnicas e *frameworks*. As palavras-chave foram escolhidas de forma a permitir obter trabalhos amplos e direcionados.

Na segunda etapa, a pesquisa foi realizada em bases de dados académicas, incluindo *Google Scholar*, *IEEE Xplore*, *SpringerLink*, *ResearchGate* e *ACM Digital Library*. Estas plataformas foram selecionadas devido à sua relevância para publicações científicas e técnicas de alta qualidade. Foram utilzados filtros para refinar os resultados por ano de publicação (focando nos últimos dez anos), impacto académico e tipo de publicação (artigos, conferências e livros). Foi feito também um estudo prévio para entender que bibliotecas seriam mais adequadas para

pesquisar um certo filtro, de modo a adequar a pesquisa e aumentar assim o número de publicações encontradas.

A terceira etapa seguiu uma abordagem iterativa, utilizando o efeito bola de neve para expandir a base de dados, ou seja, referências encontradas em trabalhos iniciais foram analisadas e integradas, desde que cumprissem os critérios de relevância e qualidade. Este método permitiu identificar fontes secundárias e complementares, criando uma base coesa e fundamentada.

2.2 Critérios de Avaliação

Os critérios de avaliação adotados neste estudo foram definidos com o objetivo de garantir uma análise abrangente e objetiva das ferramentas, técnicas e *frameworks* de programação concorrente. Um dos principais critérios foi a precisão, que avalia a capacidade das soluções de identificar corretamente problemas de concorrência, minimizando a ocorrência de falsos positivos e negativos, especialmente em ambientes críticos.

A escalabilidade foi igualmente considerada essencial, medindo o desempenho das soluções em sistemas de grande escala e altamente concorrentes, incluindo a sua capacidade de lidar com tarefas distribuídas. O desempenho foi analisado em termos de tempo médio de execução e consumo de recursos computacionais, como memória e *Central Processing Unit* (CPU), aspetos cruciais em cenários de alta carga.

Além disso, foram estabelecidas métricas claras para avaliar de forma mensurável o impacto das soluções estudadas. Estas métricas incluem a latência, medida como o tempo de resposta de uma tarefa ou operação concorrente; o tempo de execução total, que representa a duração necessária para a conclusão de todas as tarefas submetidas a um sistema; e o uso de recursos computacionais, abrangendo indicadores como o consumo médio de memória e CPU durante a execução. Adicionalmente, foi definida a taxa de erros identificados, que considera a proporção de problemas corretamente detetados pelas ferramentas em relação ao número total de erros existentes nos cenários testados. Estas métricas permitem não apenas uma comparação objetiva entre as diferentes soluções, mas também uma análise detalhada do seu desempenho em ambientes simulados e reais, fornecendo dados concretos para sustentar as avaliações realizadas.

Além disso, avaliou-se a facilidade de uso, considerando a simplicidade de configuração, a integração em pipelines de desenvolvimento e a clareza da documentação. A compatibilidade com tecnologias modernas, como arquiteturas de *microservices* e sistemas distribuídos, foi também um critério essencial, permitindo avaliar a adequação das soluções aos desafios contemporâneos.

Os critérios definidos permitiram identificar pontos fortes e limitações de cada solução analisada. Estes serviram como base para propostas de melhoria destinadas a superar as lacunas identificadas, contribuindo para o avanço no estado da arte em programação concorrente.

3 Estado da Arte

Neste capítulo apresenta-se uma revisão aprofundada das principais ferramentas e *frameworks* utilizadas na deteção de problemas de concorrência em Java, bem como das técnicas e metodologias que lhes servem de base. Adicionalmente, analisam-se as limitações das soluções atuais, evidenciando lacunas que justificam a investigação de melhorias e o desenvolvimento de novas abordagens.

3.1 Concorrência em Java

A programação concorrente tornou-se uma componente essencial do desenvolvimento de *software* moderno, particularmente em aplicações que exigem alta performance, escalabilidade e uso eficiente de recursos de *hardware multicore*. Em **Java**, a concorrência é amplamente utilizada devido às suas ferramentas nativas e *frameworks* robustos, como a *Executors Framework* e a *Fork-Join Framework*, que facilitam a execução de tarefas paralelas. No entanto, a concorrência também introduz desafios significativos, com impactos diretos na fiabilidade, eficiência e manutenibilidade dos sistemas (Goetz et al., 2006).

O ciclo de vida de uma *thread* em Java (Figura 1) compreende cinco estados principais: *New*, *Runnable*, *Running*, *Blocked* e *Dead*. Uma *thread* inicia no estado New após a sua criação, passando para *Runnable* quando o método *start*() é invocado, ficando apta a ser executada pelo escalonador. Ao ser selecionada para execução, a *thread* entra no estado *Running*, onde o método *run*() é executado. Durante a execução, a *thread* pode transitar para o estado *Blocked* devido a chamadas como *sleep*(), *wait*() ou *suspend*(), sendo posteriormente desbloqueada para retornar a *Runnable*. Finalmente, quando o método *run*() termina ou *stop*() é chamado, a *thread* entra no estado *Dead*, indicando o fim da sua execução. Compreender estas transições é essencial para gerir corretamente a concorrência e otimizar a performance em sistemas *Java*.

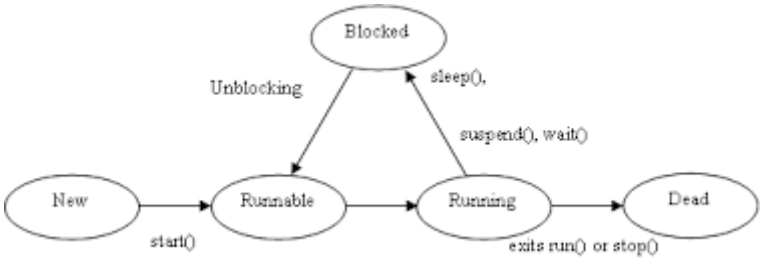


Figura 1 - Ciclo de vida de uma *thread* em Java (IGNOU, 2017)

3.1.1 Problemas Associados à Concorrência

Com a popularização de arquiteturas *multicore* e ambientes distribuídos, a procura de sistemas que executem múltiplas tarefas em simultâneo aumentou consideravelmente. Java, possui várias bibliotecas de classes e *APIs* tornando-se assim uma das principais opções para implementar concorrência. Ferramentas como o *ThreadPoolExecutor* permitem uma gestão eficiente de *threads*, ao passo que a *Fork-Join Framework* otimiza o processamento paralelo ao dividir tarefas grandes em subtarefas mais pequenas (Fu et al., 2018; Subramaniam, 2011).

Apesar disso, muitos desenvolvedores enfrentam dificuldades para gerir a complexidade introduzida pela concorrência, como por exemplo, *race conditions, deadlocks* e violações de atomicidade, especialmente em sistemas de grande escala (Urbańczyk, 2021).

Race Conditions: Estas ocorrem quando múltiplos *threads* acedem e modificam recursos compartilhados sem a devida sincronização, como se consta na Figura 2. Isso pode levar a estados inconsistentes nos dados e a comportamentos imprevisíveis (Spathoulas, 2014).

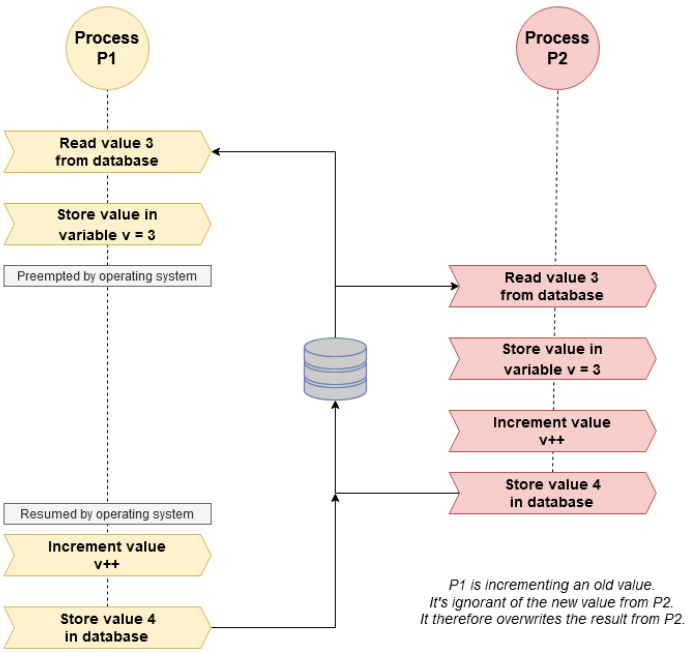


Figura 2 - Explicação *Race Condition* (Devopedia, 2020)

Deadlocks: Situações em que dois ou mais *threads* ficam bloqueadas, à espera de recursos, impossibilitando a continuação da execução, como se pode verificar na Figura 3. *Deadlocks* são particularmente difíceis de prever e resolver devido à sua natureza intermitente e dependente do tempo de execução (Oaks & Wong, 1999).

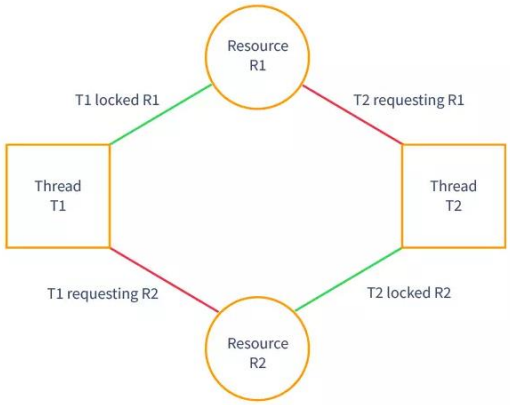


Figura 3 - Explicação Deadlock (Balaji Medewar, 2024)

Violações de atomicidade: Estes problemas ocorrem quando operações que deveriam ser indivisíveis são interrompidas, o que resulta em estados inconsistentes no sistema (Figura 4). Embora as variáveis atómicas de Java (como *Atomic Integer*) tenham sido introduzidas para mitigar esse problema, estas não eliminam completamente a necessidade de sincronização cuidadosa (Fu et al., 2018; Lea, 1996).

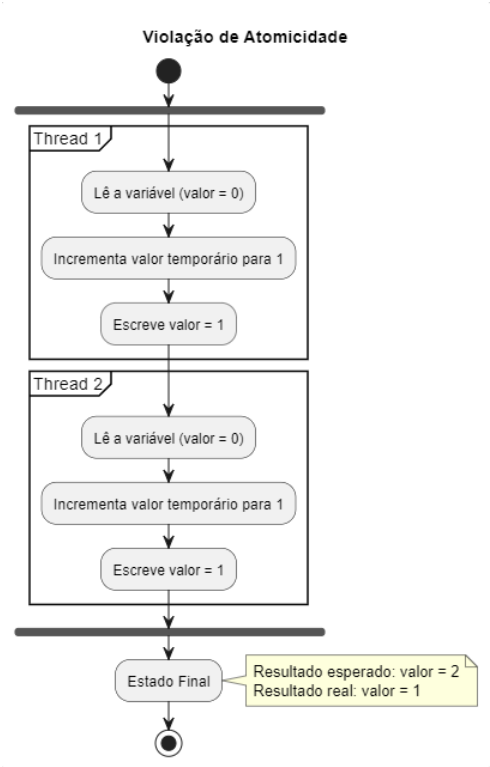


Figura 4 - Exemplo Violação de Atomicidade

3.1.2 Ferramentas e *Frameworks* de Gestão de Concorrência

Os erros de concorrência têm um impacto significativo na fiabilidade e eficiência dos sistemas, especialmente em aplicações críticas, como as utilizadas nos setores das finanças, das telecomunicações e da saúde. Um exemplo clássico é o impacto financeiro causado pela falha dos sistemas de negociação, onde problemas de concorrência podem resultar em decisões incorretas (Boehm, 1984).

Além disso, o custo associado ao *debug* e correção desses erros é elevado. Estudos mostram que, em sistemas de grande escala, até 50% do tempo total de desenvolvimento pode ser consumido em *debug* de erros concorrentes (Lea, 1996; Zeller, 2009). Isto realça a importância de ferramentas e *frameworks* que possam ajudar a detetar e resolver esses problemas de forma eficiente.

Para tentar resolver os desafios da concorrência, Java oferece várias ferramentas e *frameworks* que auxiliam na gestão de *threads* e na deteção de problemas que vão ser explorados no subcapítulo **3.2**.

Apesar dos progressos, ainda existem falhas significativas na gestão e deteção de problemas de concorrência em Java. Muitas ferramentas falham em oferecer precisão e escalabilidade adequadas, especialmente em sistemas distribuídos e de grande porte. Essas limitações destacam a necessidade de pesquisas adicionais para:

- Reduzir a ocorrência de falsos positivos e negativos(Lea, 1996; Urbańczyk, 2021).
- Melhorar a eficiência e escalabilidade de ferramentas de deteção (Fu et al., 2018; Zeller, 2009).
- Desenvolver *frameworks* que integrem análise estática e dinâmica para maior precisão na deteção de problemas (Oaks & Wong, 1999).

3.2 Tecnologias Relacionados

As tecnologias relacionadas à programação concorrente em Java abrangem um conjunto diversificado de ferramentas, técnicas e frameworks que permitem a criação de sistemas robustos, escaláveis e eficientes. Estas tecnologias foram desenvolvidas para responder aos desafios impostos pela concorrência, como *race conditions*, *deadlocks* e sincronização de recursos partilhados, garantindo a integridade e o desempenho dos sistemas. Entre as soluções disponíveis, destacam-se ferramentas de deteção de problemas, técnicas de gestão de *threads* e *frameworks* especializadas, que, em conjunto, fornecem um ecossistema abrangente para lidar com as complexidades da concorrência em aplicações modernas. Este capítulo explora estas tecnologias, destacando os seus princípios de funcionamento, aplicabilidades e limitações no contexto de sistemas *multicore* e distribuídos.

3.2.1 Ferramentas de Deteção de *Bugs* em Concorrência

A deteção de *bugs* de concorrência é uma tarefa crítica no desenvolvimento de sistemas complexos em Java, onde a programação concorrente é amplamente utilizada para melhorar a performance e escalabilidade. Foram desenvolvidas ferramentas especializadas com o objetivo de identificar problemas como *race conditions*, *deadlocks* e violações de atomicidade. Esta seção descreve algumas das principais ferramentas de deteção de *bugs* de concorrência em Java, destacando as suas funcionalidades, vantagens e limitações.

3.2.1.1 Java Pathfinder

O ***Java Pathfinder (JPF)*** é uma ferramenta de verificação de modelos desenvolvida pela *NASA Ames Research Center* com o objetivo de analisar e testar programas Java, sendo amplamente utilizada para a deteção de problemas de concorrência. O JPF executa uma análise sistemática de todos os possíveis caminhos de execução de um programa, através da simulação de *interleavings* (diferentes ordens possíveis de execução) de *threads*. Esta abordagem permite a identificação de problemas como *race conditions, deadlocks* e violação de propriedades de atomicidade, que surgem devido ao acesso não sincronizado a recursos partilhados e à espera circular de *threads* por recursos (Visser et al., 2003).

Uma das características fundamentais do JPF é a sua capacidade de realizar uma análise de todos os caminhos possíveis, incluindo execuções que seriam difíceis de reproduzir manualmente. O JPF é particularmente útil para a verificação de programas concorrentes, onde os erros dependem da ordem de execução dos *threads*. Este método garante uma deteção precisa e completa de potenciais problemas, sendo crucial em sistemas críticos, como os utilizados em aplicações aeroespaciais, sistemas médicos e financeiros (Spathoulas, 2014).

No entanto, esta abordagem exaustiva apresenta desafios significativos em termos de escalabilidade e desempenho, particularmente quando aplicada a programas de grande escala ou com uma elevada quantidade de *threads*. Este problema, **State Space Explosion**, ocorre porque o número de *interleavings* cresce exponencialmente com o aumento do número de *threads* e recursos partilhados (Clarke et al., 2012). Para mitigar esta limitação, o JPF integra **State Pruning Techniques** e **Advanced Heuristics** que permitem reduzir o espaço de busca, eliminando estados redundantes ou irrelevantes, sem comprometer significativamente a precisão da análise (Visser et al., 2003).

Além disso, o JPF oferece suporte a extensões personalizáveis, permitindo que os desenvolvedores integrem novos algoritmos ou propriedades a serem verificadas. Extensões como o *JPF-CORE* fornecem um núcleo modular para análises avançadas, enquanto plugins como o *JPF-Verify* e *JPF-Symbc* permitem realizar análise simbólica, teste de propriedades temporais e até a verificação de sistemas híbridos (Jhala & Majumdar, 2009). Esta modularidade torna o JPF uma ferramenta flexível e adaptável às necessidades específicas de diferentes aplicações.

Estudos recentes têm demonstrado a eficácia do JPF em sistemas críticos, onde a fiabilidade e segurança são prioridades absolutas. Por exemplo, a ferramenta foi aplicada com sucesso na

análise de software aeroespacial, onde a ocorrência de erros de concorrência pode ter consequências graves (Visser et al., 2003). Além disso, pesquisas mostram que o JPF é frequentemente utilizado em estudos académicos e industriais, sendo considerado uma ferramenta de referência para a verificação formal de software concorrente (Clarke et al., 2012).

3.2.1.2 ThreadSanitizer

O ***ThreadSanitizer*** é uma ferramenta de análise dinâmica desenvolvida inicialmente para linguagens como C e C++, tendo sido recentemente adaptada para suportar programas em Java. Amplamente utilizada para a deteção de *race conditions* e outros problemas de concorrência, esta ferramenta destaca-se pela sua abordagem durante a execução do programa, identificando acessos não sincronizados a dados partilhados. A análise dinâmica realizada pelo *ThreadSanitizer* complementa outras abordagens como a verificação estática utilizada pelo Java *Pathfinder* (JPF), proporcionando uma visão prática de problemas que surgem apenas em tempo de execução (Serebryany et al., 2012; Serebryany & Iskhodzhanov, 2009).

O ThreadSanitizer baseia-se em algoritmos avançados de monitorização para rastrear as operações realizadas por *threads* durante a execução de um programa. A ferramenta utiliza uma abordagem conhecida como deteção baseada em grafos de dependência, permitindo identificar interações inseguras entre *threads* de forma eficiente. Isto torna-a especialmente eficaz na deteção de problemas em sistemas onde os padrões de concorrência são complexos e dependem de condições específicas de escalonamento (Lea, 1996; Serebryany & Iskhodzhanov, 2009).

Para aumentar a eficiência, o ThreadSanitizer emprega algoritmos de deteção incremental, que analisam apenas os estados que mudaram desde a última verificação, reduzindo significativamente a sobrecarga computacional. Além disso, técnicas como *shadow memory* são utilizadas para rastrear o estado de cada variável partilhada, atribuindo metadados que identificam a origem dos acessos concorrentes (Serebryany et al., 2012; Serebryany & Iskhodzhanov, 2009; Shao et al., 2017)

A sua alta precisão na identificação de erros permite localizar *bugs* em sistemas de grande escala e alta complexidade, como aplicações financeiras ou sistemas médicos, onde a falha pode ter consequências críticas. Além disso, a capacidade de fornecer relatórios detalhados com a localização exata do problema e o contexto de execução agiliza o processo de correção por parte das equipas de desenvolvimento (Protze et al., 2021; Shao et al., 2017).

O seu baixo overhead, que, embora presente, é significativamente menor em comparação com outras ferramentas de análise dinâmica. Isto permite que seja utilizado em cenários de produção, o que não é comum em ferramentas do mesmo tipo. A flexibilidade do ThreadSanitizer torna-o aplicável tanto em testes de software como em monitorização contínua em ambientes reais (Malakar et al., 2024; Shao et al., 2017).

Além disso, o *ThreadSanitizer* tem sido integrado em sistemas de *cloud* e bases de dados distribuídas, onde as interações entre múltiplas *threads* e processos são críticas para o desempenho e consistência dos dados. A sua capacidade de operar com um *overhead*

relativamente baixo permite utilizá-lo em *pipelines* de desenvolvimento contínuo (CI/CD), detetando problemas de concorrência logo nas primeiras fases do ciclo de desenvolvimento, o que reduz custos e tempo de depuração (Malakar et al., 2024).

Apesar de ser uma ferramenta bastante utilizada, o *ThreadSanitizer* enfrenta algumas limitações que devem ser consideradas. A ocorrência de falsos negativos é um desafio intrínseco às ferramentas de análise dinâmica, pois depende dos caminhos de execução observados durante o teste. Em sistemas altamente concorrentes, onde o escalonador do sistema operativo desempenha um papel significativo, caminhos críticos podem não ser explorados, deixando problemas potenciais por identificar (Protze et al., 2021; Serebryany & Iskhodzhanov, 2009) .

Além disso, embora o *overhead* seja relativamente baixo, pode ainda ser significativo em sistemas com uma grande quantidade de *threads* e operações concorrentes. Esta limitação é mais evidente em aplicações distribuídas de larga escala, onde a densidade de interações entre componentes aumenta exponencialmente (Malakar et al., 2024; Shao et al., 2017).

A investigação em torno do *ThreadSanitizer* tem focado a melhoria da sua eficiência e precisão. Abordagens como a análise preditiva, que simula cenários alternativos de escalonamento, estão a ser exploradas para mitigar os falsos negativos. Além disso, esforços estão a ser feitos para integrar técnicas de aprendizagem automática, permitindo que a ferramenta identifique padrões de comportamento concorrente que podem indicar problemas, mesmo em cenários não diretamente observados (Malakar et al., 2024).

3.2.1.3 FindBugs com Concurrency Bug Detector

O **FindBugs com Concurrency Bug Detector** é uma extensão da ferramenta de análise estática *FindBugs*, amplamente utilizada para identificar potenciais problemas em aplicações Java. Focando-se especificamente em questões de concorrência, esta extensão foi projetada para detetar *race conditions*, *deadlocks*, acessos não sincronizados e outros problemas relacionados com a sincronização. Utilizando uma abordagem baseada na análise do *bytecode* Java, a ferramenta é capaz de identificar vulnerabilidades ainda na fase de desenvolvimento, e ajuda a prevenir erros que poderiam surgir em ambientes de produção (Fu et al., 2018; Hovemeyer & Pugh, 2004).

O *Concurrency Bug Detector* utiliza técnicas de análise estática que recorrem a *Predefined Patterns of Problematic Behavior* para identificar potenciais problemas de concorrência. Estas técnicas incluem a análise de fluxos de controlo e dependências de dados, permitindo detetar cenários como a utilização inadequada de blocos de sincronização e possíveis situações de *deadlock*. A sua integração com o ambiente de desenvolvimento *Eclipse* facilita a análise automática durante a codificação, fornecendo relatórios detalhados e recomendações para correção (Al Mamun et al., 2010; Luo et al., 2010).

Uma característica da ferramenta é a capacidade de analisar grandes bases de código de forma eficiente, atribuindo uma classificação de risco para cada problema identificado. Este sistema

permite que as equipas de desenvolvimento priorizem a resolução dos erros mais críticos, maximizando a eficácia dos esforços de manutenção (Fu et al., 2018; Manzoor et al., 2012).

O *Concurrency Bug Detector* tem sido amplamente utilizado em projetos industriais e académicos. Por exemplo, a ferramenta foi aplicada na análise de sistemas financeiros e aplicações web distribuídas, onde a falha na sincronização pode levar a perdas financeiras ou comprometer a integridade dos dados. Estudos comparativos demonstraram que o *FindBugs* supera outras ferramentas de análise estática na identificação de problemas de concorrência em aplicações Java, sendo frequentemente recomendado como solução padrão para projetos Java complexos (Hovemeyer & Pugh, 2004; Manzoor et al., 2012; Wu et al., 2017).

O FindBugs com Concurrency Bug Detector apresenta várias vantagens que o tornam uma ferramenta indispensável no desenvolvimento de software concorrente. A sua abordagem baseada em padrões pré-definidos permite identificar rapidamente problemas comuns, como o uso inadequado de sincronização, como por exemplo, objetos bloqueados incorretamente ou que não foram sincronizados corretamente e prever potenciais *deadlocks* (Luo et al., 2010; Manzoor et al., 2012). A compatibilidade com CI/CD e integração contínua permite detetar problemas automaticamente durante a compilação, garantindo um ciclo de desenvolvimento mais seguro e eficiente (Wu et al., 2017).

A ocorrência de falsos positivos é uma limitação, uma vez que a ferramenta se baseia em padrões heurísticos que podem identificar situações que, na prática, não resultariam em problemas. Além disso, como qualquer ferramenta de análise estática, a sua eficácia é limitada a erros que possam ser deduzidos a partir do código, não cobrindo problemas que dependem de condições específicas de execução (Fu et al., 2018; Kester et al., 2010).

Os desenvolvimentos recentes incluem a integração de algoritmos mais sofisticados para melhorar a precisão e reduzir os falsos positivos. Além disso, investigações em curso estão a explorar a utilização de técnicas de inteligência artificial para prever padrões de concorrência e detetar problemas que não seriam evidentes para algoritmos convencionais (Kester et al., 2010; Manzoor et al., 2012).

3.2.1.4 Seagence

O ***Seagence*** é uma ferramenta de monitorização e análise em tempo real projetada para identificar falhas em aplicações de produção, com um foco especial em problemas relacionados à concorrência, como *race conditions* e *deadlocks*. O *Seagence* diferencia-se das ferramentas tradicionais por não apenas identificar falhas, mas também rastrear a origem do problema de forma automática, fornecendo relatórios detalhados sobre o comportamento do sistema e possíveis causas dos erros detetado (Penas et al., 2017; Yönyül et al., 2024).

O Seagence opera com base em três pilares fundamentais. Primeiro, a monitorização contínua de logs e eventos, onde a ferramenta se integra diretamente em sistemas em produção e analisa constantemente os logs capturados, permitindo a deteção de falhas que surgem em condições específicas de execução, muitas vezes difíceis de reproduzir em ambientes de teste (Öztürk & Kuzucuoğlu, 2015). Segundo, a análise baseada em eventos, que utiliza algoritmos

avançados para rastrear a sequência de eventos que antecedem uma falha, identificando relações causais entre operações concorrentes, sendo especialmente eficaz na identificação de *deadlocks* e acessos não sincronizados a dados partilhados (Bora & Dikenelli, 2006). Terceiro, o uso de *Machine Learning* e heurísticas inteligentes, que permite prever potenciais problemas e sugerir ações corretivas, não apenas detetando falhas existentes, mas também antecipando possíveis pontos de falha, aumentando significativamente a confiabilidade do sistema(Jiang et al., 2020; Saufi et al., 2019).

O Seagence destaca-se ainda pelo seu baixo impacto no desempenho. Mesmo em sistemas distribuídos e altamente concorrentes, o *overhead* introduzido pela ferramenta é mínimo, tornando-a adequada para uso contínuo em ambientes de produção sem comprometer a performance do sistema (Penas et al., 2017).

O *Seagence* tem demonstrado sua eficácia em diversos setores críticos, incluindo o setor financeiro, onde é capaz de identificar rapidamente *deadlocks* e erros concorrentes que poderiam resultar em perdas financeiras significativas. O setor de telecomunicações, onde monitoriza sistemas de alta disponibilidade, detetando falhas que poderiam afetar milhões de utilizadores e o setor de *e-commerce*, onde a ferramenta deteta falhas de sincronização e erros transacionais em sistemas que processam milhões de pedidos diários, garantindo a consistência e a fiabilidade das operações (Bora & Dikenelli, 2006; Jiang et al., 2020).

Estudos mostram que a implementação do Seagence em sistemas críticos resultou em uma redução de 50% no tempo médio de resolução de problemas MTTR (*Mean Time to Resolution)*, além de uma melhoria geral na confiabilidade dos sistemas monitorizados (Saufi et al., 2019).

O *Seagence* apresenta vantagens, destacando-se pela sua capacidade de deteção em tempo real, identificando problemas enquanto o sistema está em funcionamento, o que é essencial para aplicações onde o tempo de inatividade deve ser minimizado, como em sistemas financeiros e de saúde (Yönyül et al., 2024). Além disso, o rastreio automático de falhas permite identificar a origem de problemas de forma rápida e eficiente, reduzindo drasticamente o tempo necessário para *debug* e permite que as equipas de desenvolvimento resolvam questões com maior celeridade (Jiang et al., 2020; Penas et al., 2017). O *Seagence* é também amplamente compatível com sistemas modernos, como arquiteturas baseadas em *microservices* e *containers* e integra-se facilmente com outras plataformas, oferecendo uma visão unificada e abrangente do sistema (Spieldenner et al., 2024). Por fim, a ferramenta destaca-se pela sua escalabilidade, sendo projetada para monitorizar milhares de transações por segundo sem perda de precisão na deteção de erros, tornando-a ideal para ambientes de grande escala e alta concorrência (Öztürk & Kuzucuoğlu, 2015).

O Seagence apresenta algumas limitações, principalmente devido à sua dependência de dados de *logs*, cuja eficácia está diretamente relacionada à qualidade e abrangência dos dados capturados. Em sistemas com *logs* inconsistentes ou insuficientes, certos problemas podem não ser identificados (Bitla, 2023). Além disso, em ambientes altamente distribuídos, como sistemas de larga escala ou redes complexas, a correlação entre eventos pode ser extremamente

desafiadora, exigindo configurações adicionais para assegurar a precisão da análise (Alippi et al., 2016).

No entanto, avanços recentes têm fortalecido o *Seagence* através da integração com tecnologias de *machine learning* e análise preditiva, permitindo à ferramenta prever potenciais falhas e melhorar a confiabilidade dos sistemas (Mahmood et al., 2022). Adicionalmente, o uso de algoritmos avançados e relatórios gráficos detalhados têm facilitado a identificação de falhas e otimizado a experiência do utilizador em sistemas críticos (Bitla, 2023).

3.2.1.5 JCStress

O **JCStress** é uma ferramenta de teste e validação de concorrência desenvolvida no âmbito do projeto *OpenJDK*, especificamente concebida para detetar falhas subtis relacionadas com a execução concorrente de código Java. Diferentemente das ferramentas tradicionais que atuam ao nível da análise estática ou monitorização dinâmica de sistemas completos, o *JCStress* foca-se na validação do comportamento de baixo nível de construções concorrentes, permitindo observar como pequenas secções de código se comportam sob diferentes *interleavings* e arquiteturas de *hardware*.

A ferramenta baseia-se na definição de testes de stress concorrentes onde dois ou mais atores (*threads*) executam operações sobre variáveis partilhadas. Um árbitro (*arbiter*) inspeciona o estado final dessas variáveis, permitindo inferir se ocorreu algum comportamento inesperado, como por exemplo *race conditions*, inconsistências de memória ou violações do modelo de memória do *Java*. O *JCStress* permite explorar *interleavings* raramente atingidos com testes manuais ou automatizados convencionais. Internamente, executa cada teste milhares a milhões de vezes, com combinações não determinísticas de agendamento, *delays* e otimizações de compilador. Essa abordagem torna-o eficaz para capturar erros que só ocorrem sob cargas extremas ou sob circunstâncias não reproduzíveis de forma simples (Andrianov & Mutilin, 2024; Goetz et al., 2006).

Os resultados dos testes são expressos com base nas diferentes saídas possíveis, categorizadas como:

- ACCEPTABLE – resultado esperado;
- ACCEPTABLE_INTERESTING – resultado válido, mas potencialmente problemático;
- FORBIDDEN – comportamento inválido segundo o modelo de memória.

O *JCStress* é amplamente utilizado na validação interna da plataforma *Java*, especialmente para testar estruturas como *AtomicInteger*, *ConcurrentHashMap*, *StampedLock*, *ReentrantLock*, entre outras. É uma ferramenta de desenvolvimento e manutenção de bibliotecas de concorrência de baixo nível, sendo utilizado por engenheiros da *Oracle* e pela comunidade *OpenJDK* para garantir o cumprimento do *Java Memory Model* (*JMM*)(Goetz et al., 2006).

Embora não seja concebido para análise de aplicações completas, a sua utilidade académica e industrial reside na capacidade de expor falhas invisíveis em testes tradicionais, sendo ideal para (Andrianov & Mutilin, 2024):

- validação de algoritmos concorrentes customizados;
- confirmação de garantias de atomicidade e visibilidade;
- comparação de diferentes abordagens concorrentes sob cargas simuladas.

A ferramenta é disponibilizada como um *plugin Maven* e biblioteca *Gradle*, podendo ser facilmente integrada em projetos *Java* modernos (Andrianov & Mutilin, 2024). A criação de testes requer o uso de anotações específicas **(@JCStressTest, @Actor, @Arbiter, etc.)** fornecidas pela API da ferramenta. Os resultados são exportados em formato CSV e texto, permitindo a análise quantitativa e estatística do comportamento observado (Goetz et al., 2006).

Na fase de desenvolvimento, o *JCStress* é utilizado como instrumento auxiliar de validação empírica. Pretende simular pequenos cenários clássicos de concorrência (por exemplo, o incremento não atómico com *counter++*), e verificar empiricamente a ocorrência de *race conditions*, com ou sem mecanismos de mitigação (e.g., *synchronized*, *AtomicInteger*, *ReentrantLock*) (Goetz et al., 2006).

Entre as principais limitações do *JCStress* destaca-se o seu âmbito restrito: a ferramenta não visa analisar grandes sistemas nem possui interface gráfica. A curva de aprendizagem é moderada, exigindo familiaridade com o modelo de memória do *Java*. Além disso, os testes são desenhados manualmente, o que implica conhecimento prévio sobre os padrões de concorrência a validar. No entanto, a exatidão dos resultados e a sua capacidade de revelar comportamentos críticos com alta precisão tornam o *JCStress* uma ferramenta essencial para o desenvolvimento de software concorrente seguro e confiável, especialmente em bibliotecas ou *frameworks* que serão utilizadas como fundações para sistemas maiores (Andrianov & Mutilin, 2024).

3.2.2 Técnicas de Programação Concorrente

As técnicas de programação concorrente são essenciais para gerir recursos partilhados de forma segura e eficiente, prevenindo problemas como *race conditions* e *deadlocks*. Em Java, estas técnicas incluem abordagens clássicas, como sincronização, e soluções avançadas, como *CompletableFuture*, *AtomicVariables, ReentrantLock* ou *ThreadLocal*, proporcionando suporte para sistemas modernos e escaláveis. Nos subcapítulos seguintes, serão analisadas estas abordagens e as suas aplicações.

3.2.2.1 Sincronização de Blocos e Métodos

A **sincronização de blocos e métodos** é uma técnica utilizada na programação concorrente para controlar o acesso a recursos partilhados, garantindo que apenas um thread aceda a secções críticas do código de cada vez. A utilização da palavra-chave ***synchronized*** permite a execução coordenada de *threads*, previne situações como *race conditions* e assegura a consistência dos dados em sistemas *multithread*. Esta técnica desempenha um papel crucial na gestão de concorrência em sistemas críticos, onde a preservação da integridade dos dados é essencial (Goetz et al., 2006; Oaks & Wong, 1999).

A sincronização pode ser aplicada tanto a métodos inteiros como a blocos específicos de código. No primeiro caso, toda a execução do método é protegida, garantindo que apenas uma *thread* pode aceder ao método por vez como ilustrado na Figura 5, onde um método sincronizado é explicitamente identificado como protegido de acessos simultâneos por outros *threads*. Por outro lado, a sincronização de blocos permite proteger apenas secções específicas, como mostra na Figura 5 o que permite uma maior flexibilidade e melhor desempenho, especialmente em métodos que operam sobre múltiplos recursos partilhados. Esta abordagem baseia-se na utilização de monitores associados a objetos, que controlam o acesso exclusivo às secções sincronizadas do código (Goetz et al., 2006; Oaks & Wong, 1999).

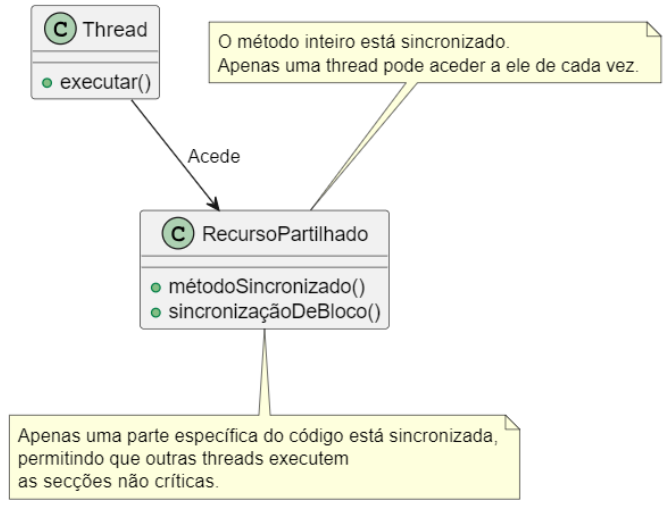


Figura 5 - Diagrama de Sincronização de blocos e métodos

A sincronização de blocos e métodos apresenta várias vantagens que a tornam indispensável em sistemas concorrentes. Esta técnica garante a preservação da consistência dos dados, impedindo que operações realizadas por múltiplas *threads* resultem em estados inconsistentes de recursos partilhados. Além disso, ao permitir que apenas as secções críticas do código sejam protegidas, a sincronização otimiza o desempenho, evitando bloqueios desnecessários em métodos inteiros. A aplicação da sincronização de blocos e métodos é especialmente relevante em cenários críticos, como operações financeiras, onde transações simultâneas devem ser processadas de forma segura, e em sistemas médicos, onde a integridade dos dados é uma exigência fundamental (Goetz et al., 2006; Oaks & Wong, 1999).

Apesar das suas vantagens, a sincronização de blocos e métodos enfrenta desafios importantes que exigem atenção no design de sistemas concorrentes. Um dos principais problemas é o risco de *deadlocks*, que ocorrem quando duas ou mais *threads* ficam bloqueadas à espera de recursos que nunca serão libertados. Além disso, a sincronização pode impactar o desempenho em sistemas com alta concorrência, uma vez que os tempos de espera para adquirir e libertar *locks* podem crescer significativamente. Em ambientes distribuídos, a coordenação entre vários nós aumenta a complexidade da implementação, tornando necessário o uso de abordagens adicionais para garantir a consistência e evitar conflitos (Alvarez Cid-Fuentes et al., 2020).

Novas técnicas de sincronização têm sido desenvolvidas para lidar com os desafios associados à alta concorrência e melhorar o desempenho dos sistemas. Métodos como ***biased locking*** e ***adaptive spinning*** têm mostrado eficácia na redução dos tempos de espera para aquisição de locks, minimizando o impacto no desempenho de sistemas altamente concorrentes. O *biased locking*, por exemplo, reduz o *overhead* associado a operações de sincronização ao permitir que *threads* recorrentes mantenham a propriedade do *lock* em condições específicas, evitando alterações desnecessárias no estado do recurso bloqueado. O *adaptive spinning*, por sua vez, ajusta dinamicamente o comportamento de espera das *threads* com base na carga do sistema, permitindo decisões otimizadas entre permanecer em espera ativa ou ceder o controlo, de forma a equilibrar o consumo de recursos (Chen et al., 2021).

Adicionalmente, o uso de variáveis atómicas, como *AtomicInteger* e *AtomicLong*, e de *locks* explícitos, como o *ReentrantLock*, oferece uma abordagem alternativa à sincronização convencional, proporcionando maior flexibilidade para lidar com cenários de concorrência complexa. Estas técnicas permitem operações seguras em dados partilhados sem necessidade de bloqueios extensivos, reduzindo a contenção entre *threads* (Ahmed et al., 2020). Estes tópicos irão ser discutidos posteriormente nas seções 3.2.2.4 e 3.2.2.3.

3.2.2.2 CompletableFuture

Introduzido no Java 8, o **CompletableFuture** é uma classe que facilita a programação assíncrona ao oferecer um modelo para compor e executar operações concorrentes de forma não bloqueante. Baseada na API de concorrência do Java, o *CompletableFuture* amplia as capacidades da interface *Future*, permitindo maior flexibilidade na criação de pipelines assíncronos e no tratamento de exceções durante a execução de tarefas. Este fluxo é representado na Figura 6, que ilustra desde o início de uma tarefa assíncrona (*supplyAsync*) até a aplicação de transformações (*thenApply*) e composições (*thenCompose*) para criar um pipeline eficiente (Smith, 2015; Zhang et al., 2024).

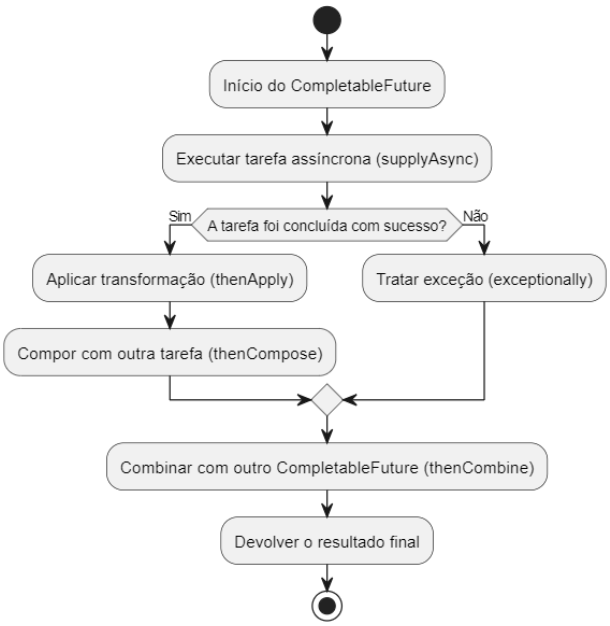


Figura 6 - Diagrama *CompletableFuture*

A classe oferece métodos como *thenApply*, *thenCompose* e *handle*, que possibilitam a construção de fluxos de execução encadeados e assíncronos. Estes métodos facilitam a gestão de operações, como o tratamento de exceções (*exceptionally*) e a combinação de resultados (*thenCombine*), reduzindo a complexidade na implementação de sistemas concorrentes. Estas funcionalidades são utilizadas em aplicações que dependem de operações concorrentes, como sistemas distribuídos e serviços baseados em *microservices* (Beronic et al., 2021; Smith, 2015).

O *CompletableFuture* suporta também a paralelização de tarefas através de métodos como *supplyAsync* e *runAsync*. Estas capacidades são frequentemente utilizadas em aplicações de alta performance para permitir a execução de operações independentes sem bloquear as *threads* principais. A integração com o modelo de concorrência estruturada do Java contribui para a gestão de *threads* virtuais, otimizando o desempenho em sistemas escaláveis (Beronic et al., 2021).

A utilização do *CompletableFuture* apresenta desafios específicos. A gestão de exceções em *pipelines* assíncronas requer uma abordagem cuidadosa, especialmente em sistemas críticos, onde falhas podem afetar a consistência dos dados. Adicionalmente, a criação e gestão de tarefas assíncronas pode aumentar o consumo de recursos, o que exige uma análise cuidada em cenários de elevada concorrência (Zhang et al., 2024).

Pesquisas recentes têm explorado o uso do *CompletableFuture* em contextos de grande escala. Avanços incluem melhorias na interoperabilidade com bibliotecas de programação reativa, como *RxJava*, e na sua utilização em sistemas de *streaming* de dados em tempo real, onde a programação assíncrona é essencial para o desempenho (Smith, 2015).

3.2.2.3 ReentrantLock

O ***ReentrantLock***, introduzido na biblioteca ***java.util.concurrent.locks*** desde do Java 5, é uma alternativa ao uso da palavra-chave *synchronized*. Este tipo de *lock* oferece controlo explícito sobre os bloqueios, disponibilizando funcionalidades como aquisição de *locks* com temporizadores e verificação não bloqueante. O diagrama da Figura 7 ilustra o fluxo básico de utilização do *ReentrantLock*, desde a tentativa de aquisição do *lock* até à libertação após a execução da secção crítica (Milani, 2023; Schäfer et al., 2011).

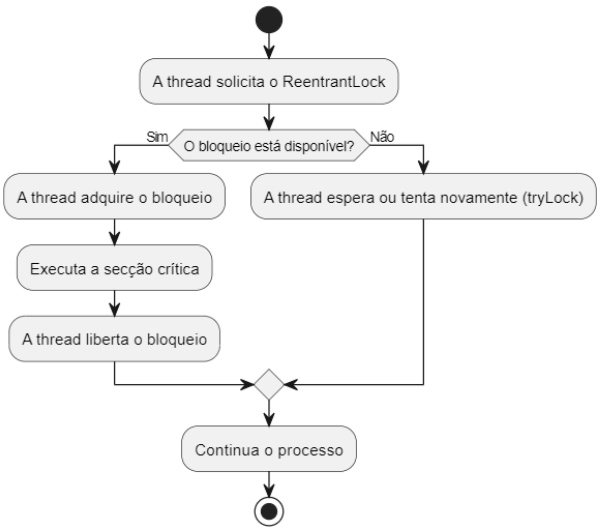


Figura 7 - Diagrama *ReentrantLock*

Diferente do *synchronized*, o *ReentrantLock* proporciona maior flexibilidade em cenários concorrentes, permitindo o uso de métodos como *tryLock()*. Este método possibilita que uma *thread* continue a execução caso o *lock* não esteja disponível, reduzindo o risco de deadlocks em sistemas de alta concorrência (Kode & Oyemade, 2024; Schäfer et al., 2011).Além disso, o *ReentrantLock* suporta bloqueios reentrantes, nos quais a mesma *thread* pode adquirir o *lock* várias vezes, desde que o liberte no mesmo número de ocasiões. Este comportamento é útil na implementação de algoritmos complexos que envolvem múltiplas operações sobre o mesmo recurso partilhado (Laneve, 2019).

Outra funcionalidade importante é o suporte a *fair queues*, que garantem uma ordem predefinida na aquisição dos *locks*. Este recurso evita *thread starvation*, que ocorre quando uma *thread* é continuamente preterida na aquisição de um recurso devido à prioridade constante de outras *threads*. Este mecanismo é amplamente aplicado em sistemas distribuídos e aplicações críticas, como arquiteturas baseadas em *microservices*, onde a consistência de acesso a recursos partilhados é fundamental (Hofer et al., 2016; Schäfer et al., 2011).

Apesar das suas vantagens, o *ReentrantLock* apresenta desafios. A manipulação explícita dos *locks* aumenta o risco de falhas na libertação dos *locks*. Estas falhas podem causar contenção de recursos ou *deadlocks*, pelo que o o uso de blocos try-finally é recomendado para garantir a liberação do *lock* mesmo em caso de exceções (Laneve, 2019; Milani, 2023).

Pesquisas recentes analisaram o impacto do ReentrantLock em sistemas de grande escala, explorando técnicas de refatoração para melhorar a flexibilidade no uso de *locks*. Estas técnicas integram o ReentrantLock com arquiteturas modernas de programação concorrente (Kode & Oyemade, 2024).

3.2.2.4 Atomic Variables

As ***Atomic Variables*** são classes introduzidas na biblioteca ***java.util.concurrent.atomic*** para operações concorrentes seguras e eficientes. Diferente das variáveis convencionais, elas oferecem operações atómicas diretamente no hardware, eliminando a necessidade de sincronização explícita. Isto torna-as ideais para cenários onde múltiplas *threads* acedem simultaneamente a recursos partilhados (Ishizaki et al., 2011; Lea, 1996).

Estas variáveis incluem classes como *AtomicInteger, AtomicLong, AtomicBoolean* e *AtomicReference*, cada uma projetada para um tipo específico de dados. Estas utilizam algoritmos baseados em instruções de hardware, como *Compare-and-Swap (CAS)*, para garantir a integridade das operações sem bloqueios explícitos, de forma a reduzir o *overhead* em sistemas altamente concorrentes (Paulino et al., 2016; Pinto et al., 2015).

A Tabela 2 apresenta as principais classes de variáveis atómicas disponíveis em Java, descrevendo as suas características e usos comuns. Estas classes são amplamente utilizadas em sistemas onde é necessário um controlo eficiente de estados partilhados entre *threads*, contribuindo para a escalabilidade de sistemas concorrentes (Charpentier, 2023).

Tabela 2 - Principais Variáveis Atómicas em Java

Classe	**Descrição**	**Usos Comuns**
AtomicInteger	Representa um inteiro que suporta operações atómicas	Contadores, índices, e sinais de controlo
AtomicLong	Similar ao *AtomicInteger*, mas para valores longos	Contadores globais e métricas de performance.
AtomicBoolean	Representa um valor booleano que pode ser atualizado atomicamente.	Indicadores de estado (ex.: ativado/desativado).
AtomicReference<T>	Permite operações atómicas em referências a objetos genéricos.	Troca segura de referências em tempo de execução.

Apesar das suas vantagens, as variáveis atómicas apresentam limitações em cenários mais complexos. Em casos onde múltiplas operações precisam ser realizadas de forma interdependente, o uso de variáveis atómicas pode levar a inconsistências, exigindo alternativas como *locks* explícitos. No entanto, para tarefas isoladas, estas classes oferecem uma solução eficiente e escalável (Charpentier, 2023).

3.2.2.5 ThreadLocal

O ***ThreadLocal*** é uma classe da *Application Programming Interface (API)* de concorrência do Java que permite a cada *thread* ter a sua própria cópia de uma variável. Esta abordagem elimina a

necessidade de sincronização explícita e previne interferências entre *threads*. É amplamente utilizada em aplicações que requerem dados temporários específicos de uma *thread*, como controlo de sessões ou personalização de dados locais, incluindo fuso horário ou moeda (Goetz et al., 2006; Saad & Ravindran, 2011).

A funcionalidade do *ThreadLocal* baseia-se na utilização de mapas internos geridos pela *Java Virtual Machine (JVM)* nos quais cada *thread* mantém acesso exclusivo às suas variáveis. Este recurso é utilizado em contextos como a gestão de conexões a bases de dados, onde o isolamento das variáveis é necessário para evitar conflitos e manter a integridade dos dados (Carlstrom et al., 2006).

O uso do *ThreadLocal* apresenta desafios, como o risco de *memory leaks*. Este problema pode ocorrer se as variáveis associadas a uma *thread* não forem removidas após a conclusão da sua execução. Em sistemas de longa duração, como servidores *web*, esta situação pode levar a um aumento no consumo de memória, impactando a eficiência do sistema. Para evitar este cenário, recomenda-se a utilização de blocos *try-finally*, garantindo que as variáveis sejam removidas após o uso (Domani et al., 2003; N. Zhou, 2016).

Pesquisas recentes analisaram o uso do *ThreadLocal* em cenários modernos, incluindo ambientes distribuídos e memórias persistentes. Estas aplicações ampliam a sua relevância em sistemas de execução paralela, onde o isolamento entre *threads* é fundamental para garantir a integridade e a escalabilidade (Lefort, 2023).

3.2.3 Análise de *Frameworks* de Concorrência em Java

A utilização de *frameworks* na programação concorrente em Java proporciona abordagens robustas e flexíveis para gerir tarefas assíncronas, paralelismo e interação entre *threads*. Estes *frameworks* vão além das técnicas básicas, oferecendo ferramentas específicas para otimizar o desempenho em sistemas distribuídos e *multicore*. Nos subcapítulos seguintes, serão analisados *frameworks* amplamente utilizados, como a *Executors Framework*, a *Fork-Join* Framework, o modelo de programação reativa com *RxJava* e *Reactor*, e o *Google's Thread Weaver*, cada um com as suas aplicações, benefícios e limitações.

3.2.3.1 Executors Framework

A **Executors Framework**, introduzido no Java 5, é um componente central da API de concorrência que simplifica a gestão de *threads* e a execução de tarefas assíncronas em aplicações concorrentes. Ao abstrair a criação e gestão manual de *threads*, a framework fornece uma interface que permite que os programadores se concentrem na lógica das suas aplicações em vez de nos detalhes da implementação. Uma das suas principais classes é o ThreadPoolExecutor, que gere *pools* de *threads* reutilizáveis para reduzir o *overhead* associadas aos processos de criação e eliminação de *threads* (Fernández, 2016).

O funcionamento da *Executors Framework* baseia-se no conceito de *Executor Services*, que gerem tarefas de forma eficiente. Estes serviços oferecem diferentes configurações de *thread*

pools, como o *Fixed Thread Pool*, que aloca um número fixo de *threads*, adequado para cargas de trabalho previsíveis; o *Cached Thread Pool*, que cria *threads* conforme necessário e reutiliza *threads*, ajustando-se a cargas variáveis; e o *Scheduled Thread Pool*, que permite a execução periódica de tarefas, sendo útil para o agendamento de eventos (Jansson & Hellberg, 2010).

Estas configurações tornam a *Executors Framework* altamente flexível, de forma a permitir a sua aplicação em diversas áreas, como sistemas distribuídos, aplicações de análise de dados e arquiteturas baseadas em *microservices*. Em aplicações críticas, como serviços financeiros ou sistemas de controlo em tempo real, o uso de *thread pools* reduz a latência e melhora a previsibilidade (Goetz et al., 2006; Veen & Vlijmincx, 2024).

Uma das funcionalidades mais relevantes da *framework* é a capacidade de personalizar parâmetros no *ThreadPoolExecutor*. Por exemplo, é possível definir o número máximo de *threads*, o tamanho da fila de espera e os tempos de espera para as tarefas. Estas configurações permitem que os desenvolvedores ajustem o comportamento do sistema de acordo com as necessidades da aplicação, otimizando o uso de recursos e minimizando a contenção entre *threads*. No entanto, a má configuração pode levar a problemas como *thread starvation*, exigindo uma análise criteriosa durante o design do sistema (Daugherty, 2011).

Pesquisas recentes exploraram o uso da Executors Framework em combinação com novas funcionalidades, como *threads* virtuais introduzidas no Java 21. Estas *threads* oferecem uma alternativa leve às *threads* tradicionais, permitindo que aplicações escalem para milhões de tarefas concorrentes. Este avanço reforça o papel da Executors Framework como uma ferramenta essencial para a programação concorrente moderna, particularmente em sistemas de grande escala e alto desempenho (Saad & Ravindran, 2011; Veen & Vlijmincx, 2024).

3.2.3.2 Fork-Join Framework

A **Fork-Join Framework**, introduzido no Java 7, é uma ferramenta concebida para maximizar a utilização de sistemas *multicore*. Baseia-se no modelo de dividir para conquistar (*divide and conquer*), permite a execução paralela de subtarefas em problemas recursivos. Esta *framework* utiliza um *work-stealing algorithm*, onde *threads* assumem as subtarefas de outras *threads* ocupadas, o que acaba por otimizar a utilização dos recursos disponíveis (Fernández, 2016; Lea, 1996).

O funcionamento da *Fork-Join Framework* está centrado na classe *ForkJoinPool*, que coordena a execução das tarefas submetidas. As tarefas são representadas pelas classes *RecursiveTask*, usada para operações que retornam resultados, e *RecursiveAction*, destinada a operações que não retornam valores. Este modelo é particularmente eficaz para problemas que podem ser divididos em partes menores, como a multiplicação de matrizes, cálculo de números de *Fibonacci*, e algoritmos de pesquisa paralela em grandes coleções de dados (De Wael et al., 2014; Ponge, 2011).

A principal vantagem da *Fork-Join Framework* é a sua capacidade de explorar eficientemente o paralelismo disponível em sistemas *multicore*, reduzindo o tempo de execução em tarefas computacionalmente intensivas. Além disso, o uso do *work-stealing* garante um

balanceamento dinâmico da carga entre as *threads*, minimizando tempos de inatividade. No entanto, esta *framework* apresenta limitações em tarefas que não podem ser facilmente divididas ou em sistemas com poucos núcleos, onde o *overhead* da gestão de tarefas pode impactar o desempenho (Jansson & Hellberg, 2010; Ponge, 2011).

Pesquisas recentes têm explorado o uso da *Fork-Join Framework* em combinação com novas arquiteturas, como *serverless computing* e ambientes distribuídos, expandindo o seu alcance para além do paralelismo local. Estas abordagens destacam a *framework* como uma solução relevante para cenários modernos de computação em larga escala (De Wael et al., 2014; Fernández, 2016).

3.2.3.3 Reactive Programming

Reactive Programming é um paradigma que promove a construção de sistemas assíncronos e não bloqueantes, focados em *data streams* e *change propagation*. Em Java, *frameworks* como *RxJava* e *Reactor* implementam os princípios do *Reactive Streams*, introduzidos no Java 9, que definem um padrão para o processamento assíncrono de dados com *backpressure*. *Backpressure* refere-se à capacidade de controlar o fluxo de dados entre os *producers* e os *consumers* em sistemas assíncronos. Quando o *producer* gera dados mais rapidamente do que o *consumer* consegue processar, a *backpressure* fornece mecanismos para evitar sobrecargas, garantindo que o sistema permaneça eficiente e estável. Estas bibliotecas permitem a gestão eficiente de fluxos de eventos, escalando bem em sistemas *multicore* e distribuídos (Ponge et al., 2021).

No centro do *Reactive Programming* estão conceitos como *Publishers* e *Subscribers*, que facilitam a comunicação entre diferentes componentes de um sistema. O *RxJava*, por exemplo, fornece operadores para transformar, combinar e filtrar *data streams* de forma declarativa. Já o *Reactor*, construído para a *Spring Framework*, oferece abstrações como *Mono* e *Flux*, que representam fluxos de um ou mais eventos, respetivamente. Estas ferramentas permitem a criação de sistemas resilientes e altamente responsivos, eliminando bloqueios desnecessários de *threads* (Malhotra, 2019).

A principal vantagem do *Reactive Programming* é a sua capacidade de lidar com grandes volumes de dados e eventos em tempo real. Por outro lado, apresenta desafios, como a curva de aprendizagem e a complexidade de *debug* em sistemas altamente assíncronos. Apesar disso, o modelo reativo tem sido amplamente adotado em domínios como *data streaming*, *Internet of Things (IoT)* e *microservices*, devido à sua eficiência em ambientes distribuídos (Cheng, 2018).

Pesquisas recentes têm explorado o uso de *Reactive Programming* em arquiteturas *serverless* e sistemas baseados em *microservices*, onde a escalabilidade e a gestão de eventos assíncronos são essenciais. Este avanço destaca o impacto do paradigma em sistemas modernos, consolidando *frameworks* como *RxJava* e *Reactor* como ferramentas indispensáveis para programação concorrente e reativa (Ponge et al., 2021).

3.2.3.4 Google's Thread Weaver

O **Google's Thread Weaver** é uma *framework* projetada para analisar e verificar interações entre *threads* em aplicações Java. Esta utiliza abordagens de verificação sistemática para identificar problemas de concorrência, como *race conditions* e *deadlocks*, em sistemas *multithread*. A análise baseia-se na simulação de diferentes *interleavings*, permitindo que comportamentos concorrentes complexos sejam explorados e avaliados de forma rigorosa (Huang & Zhang, 2016).

O Thread Weaver foca-se na modelação detalhada das interações entre *threads*, permitindo a reprodução de cenários raros e difíceis de testar manualmente. A ferramenta insere pontos de controlo estratégicos no código da aplicação, monitorizando e ajustando a execução de *threads*. Esta abordagem é especialmente útil para identificar falhas que dependem de ordens específicas de execução, comuns em sistemas distribuídos e de larga escala (Cicirelli et al., 2022).

Entre as suas principais vantagens está a capacidade de integrar e analisar ambientes de desenvolvimento modernos, permitindo que os resultados sejam facilmente interpretados pelos desenvolvedores. No entanto, a utilização do *Thread Weaver* pode apresentar desafios em sistemas com elevado número de *threads*, devido à complexidade e ao custo computacional associados à análise exaustiva de *interleavings* (Huang & Zhang, 2016).

Pesquisas recentes destacam a aplicação do *Thread Weaver* em sistemas críticos, como arquiteturas de *IoT* e plataformas *serverless*, onde a confiabilidade e a robustez são fundamentais. Estes estudos exploram a capacidade da framework em identificar falhas que poderiam passar despercebidas em testes convencionais, reforçando a sua utilidade em contextos de elevada concorrência (Touhafi et al., 2018).

3.3 Trabalhos Relacionados

Este capítulo explora ferramentas, técnicas e *frameworks* que desempenham um papel central na deteção e gestão de problemas de concorrência, destacando abordagens propostas na literatura, suas aplicações práticas e as limitações observadas. Esta revisão contextualiza a pesquisa apresentada nesta tese, fundamentando-a nos avanços existentes e evidenciando a necessidade de novas contribuições para otimizar a confiabilidade e a eficiência em sistemas concorrentes.

3.3.1 Trabalhos sobre Ferramentas de Deteção de Problemas de Concorrência

Os desafios inerentes à programação concorrente, como *race conditions*, *deadlocks* e violações de atomicidade, motivaram o desenvolvimento de ferramentas que ajudam na identificação e resolução desses problemas. No entanto, os trabalhos relacionados a essas ferramentas demonstram avanços e limitações significativas, que são explorados a seguir. Entre as ferramentas analisadas estão o *JPF*, o *ThreadSanitizer*, o *FindBugs com Concurrency Bug Detector* e o *Seagence*.

O *Java Pathfinder (JPF)*, é uma ferramenta de verificação de modelos que simula diferentes *interleavings* de *threads* em programas Java, permitindo a identificação de *race conditions*, *deadlocks* e violações de propriedades de atomicidade. Havelund e Roşu (2004) destacaram a eficácia do JPF na deteção de erros em sistemas críticos, como controladores aeroespaciais e aplicações médicas (Havelund & Roşu, 2004). Esta ferramenta utiliza um motor de execução simbólica que verifica todos os estados possíveis de um programa, sendo especialmente útil para cenários onde é crucial garantir a ausência de falhas. Além disso, Zhou (2020) enfatizou a importância do JPF na análise do rastreio de execução, permitindo que programadores visualizem e compreendam melhor os fluxos concorrentes complexos em suas aplicações (Y. Zhou, 2020).

No entanto, um dos maiores desafios enfrentados pelo *JPF* é a *State Space Explosion*, particularmente em sistemas grandes e complexos. Para mitigar esse problema, Sahoo e Ray (2018) e Clarke (2012) destacam técnicas como *State Pruning Techniques* e *Advanced Heuristics* como propostas de soluções (Clarke et al., 2012; Sahoo & Ray, 2018). Trabalhos recentes continuam a explorar como otimizar o desempenho do JPF, incluindo a sua integração com novas tecnologias de análise simbólica e paralelização de processos para melhorar a escalabilidade em sistemas distribuídos (Sahoo & Ray, 2018; Y. Zhou, 2020).

O *ThreadSanitizer* é uma ferramenta dinâmica amplamente utilizada para identificar *race conditions* e outros problemas de concorrência em sistemas Java e C++. Serebryany e Iskhodzhanov (2009) sublinham a sua capacidade de detetar erros durante a execução do programa, destacando a análise em tempo real como um fator crucial para ambientes de produção . Em particular, a ferramenta utiliza algoritmos de deteção de acessos não sincronizados a dados partilhados, o que a torna ideal para aplicações com requisitos rigorosos de consistência de dados e baixa tolerância a falhas (Serebryany & Iskhodzhanov, 2009).

Apesar de sua eficácia, o *ThreadSanitizer* apresenta limitações na cobertura de caminhos de execução que não são explorados durante os testes, resultando em falsos negativos, como observado em estudos comparativos por Spathoulas (2014). No entanto, sua integração com ambientes de produção, aliada a um impacto relativamente baixo no desempenho do sistema, continua a torná-lo uma escolha popular para sistemas críticos e distribuídos (Spathoulas, 2014).

O *FindBugs com Concurrency Bug Detector* baseia-se em uma abordagem estática para a análise de código Java, detetando padrões predefinidos de comportamento problemático relacionados à concorrência. Estudos como os de Spathoulas (2014) destacam sua integração em pipelines de integração contínua, permitindo a identificação precoce de erros em projetos de software. Além disso, mostrou que o *FindBugs* é particularmente útil em projetos onde a cobertura completa de testes não é viável, fornecendo uma camada adicional de segurança contra falhas concorrente (Spathoulas, 2014).

No entanto, o *FindBugs* enfrenta críticas relacionadas à sua elevada taxa de falsos positivos, que pode chegar a 20%, limitando sua aplicabilidade em projetos de larga escala e sistemas distribuídos. Trabalhos recentes exploram técnicas de inteligência artificial e aprendizagem de

máquina para reduzir os falsos positivos e melhorar a precisão da análise estática, o que pode aumentar sua adoção em ambientes industriais (Spathoulas, 2014).

O *Seagence* diferencia-se por sua abordagem inovadora de análise baseada em *logs* e eventos. Bora e Dikenelli (2006) destacam que a ferramenta foi projetada para operar em sistemas em produção, monitorizando continuamente os fluxos de execução e detetando falhas raras em tempo real. Em particular, o *Seagence* é amplamente aplicado em arquiteturas de *microservices*, onde a alta concorrência e a distribuição de tarefas exigem ferramentas capazes de correlacionar eventos complexos (Bora & Dikenelli, 2006).

Bitla (2023) também salienta a eficácia do *Seagence* em ambientes distribuídos, especialmente na identificação de *race conditions* e erros de sincronização em sistemas críticos. Contudo, a dependência de dados de *logs* de alta qualidade e configurações complexas são frequentemente citados como desafios para a implementação em larga escala. Avanços recentes, como a integração de algoritmos preditivos baseados em *machine learning*, têm ampliado as capacidades do *Seagence*, tornando-o uma escolha atraente para sistemas modernos e escaláveis (Bitla, 2023).

3.3.2 Trabalhos sobre Técnicas de Programação Concorrente

A programação concorrente em Java é suportada por um conjunto diversificado de técnicas que oferecem abordagens flexíveis e eficientes para gerir o acesso a recursos partilhados e executar tarefas em paralelo. Estas técnicas abrangem desde métodos tradicionais, como *synchronized*, até soluções mais recentes, como *CompletableFuture*. Trabalhos relacionados mostram que a escolha da técnica ideal depende das características do sistema, do nível de concorrência exigido e das necessidades específicas de desempenho.

A palavra-chave *synchronized* é usada como uma técnica tradicional para gerir a concorrência em Java, permitindo o controlo de acesso a secções críticas do código. Ao utilizar *monitors* associados a objetos ou métodos, esta abordagem garante que apenas uma *thread* pode aceder simultaneamente ao bloco ou método sincronizado. Estudos de Goetz et al. (2006) destacam a simplicidade e eficácia desta técnica em sistemas onde a consistência de dados é prioritária (Goetz et al., 2006). Contudo, trabalhos como os de Spathoulas (2014) mostram que a sincronização baseada em *synchronized* pode levar a problemas de *deadlock* em sistemas complexos e causar degradação de desempenho devido ao bloqueio extensivo de *threads.* Esses problemas, combinados com tempos médios de execução relativamente altos em cenários de alta concorrência (30 ms em média em testes de carga), podem limitar a aplicabilidade da técnica (Spathoulas, 2014).

Introduzido no Java 8, o *CompletableFuture* é uma solução moderna que facilita a programação assíncrona e não bloqueante. Trabalhos recentes, como os de Tramontana e Midolo (2023), mostram que esta técnica permite a execução eficiente de tarefas em paralelo, utilizando fluxos de execução separados e evitando bloqueios desnecessários de *threads*. Em cenários de alto desempenho, os tempos de execução para tarefas assíncronas demonstram uma redução de

até 30% em relação às técnicas tradicionais, como *synchronized*. A composição de tarefas e suporte integrado ao padrão *Reactive Streams* tornam o *CompletableFuture* ideal para aplicações de grande escala e sistemas distribuídos, destacando a sua flexibilidade e eficiência na gestão de fluxos de dados complexos, como aplicações de *streaming* e IoT. (Midolo & Tramontana, 2023).

O *ReentrantLock*, oferece uma alternativa mais flexível ao *synchronized*, permitindo funcionalidades adicionais, como *fair locks* e interrupção de *threads* em espera. Ahmed et al. (2020) destacam que esta técnica é amplamente utilizada em sistemas onde o controlo detalhado do bloqueio é essencial, como em arquiteturas distribuídas (Ahmed et al., 2020).

Comparações mostram que o *ReentrantLock* pode ser até 15% mais eficiente do que *synchronized* em operações de baixa contenção, mas enfrenta degradação de desempenho em situações de alta concorrência, com tempos médios de espera de 20 ms em condições de carga elevada (Malhotra, 2019).

Contudo, estudos de Malhotra (2019) mostram que o uso incorreto de bloqueios explícitos pode levar a problemas de contenção e degradação de desempenho em cenários de alta concorrência (Malhotra, 2019).

As variáveis atómicas, como *AtomicInteger* e *AtomicLong*, oferecem uma abordagem alternativa para gerir a concorrência sem necessidade de sincronização explícita. Estas variáveis permitem operações atómicas diretamente no hardware subjacente, garantindo maior desempenho em sistemas com alta carga concorrente. Estudos como os de Sahoo e Ray (2018) sublinham que as variáveis atómicas são particularmente úteis em algoritmos paralelos e sistemas de baixa latência (Sahoo & Ray, 2018).

A classe *ThreadLocal* fornece uma forma eficiente de isolar dados entre diferentes *threads*, eliminando a necessidade de sincronização em certos cenários. O estudo de Veen e Vlijmincx (2024) destaca que esta técnica é amplamente utilizada em sistemas onde cada *thread* requer um estado independente, como em aplicações financeiras e sistemas de análise em tempo real. Comparativamente, o uso de *ThreadLocal* mostrou-se eficaz na redução de interferências entre *threads*, melhorando a previsibilidade do desempenho em até 20% em sistemas com alta concorrência. Contudo, a dependência de memória para cada instância da *thread* pode levar a um aumento no uso de recursos em aplicações de larga escala, o que representa um desafio para sistemas distribuídos (Veen & Vlijmincx, 2024).

As técnicas analisadas oferecem soluções complementares para os desafios da concorrência em Java. A escolha da técnica mais adequada depende do contexto do sistema, das características específicas da aplicação e das metas de desempenho estabelecidas. Combinação de técnicas, como por exemplo, o uso de *CompletableFuture* com *Atomic Variables*, pode proporcionar benefícios significativos em termos de desempenho e escalabilidade, especialmente em sistemas modernos e distribuídos.

3.3.3 Trabalhos sobre *Frameworks* de Programação Concorrente

As *frameworks* de programação concorrente são essenciais para gerir a execução paralela em Java, oferecendo abstrações e ferramentas otimizadas para lidar com tarefas complexas em sistemas *multicore* e distribuídos. Entre as principais *frameworks* analisadas estão a *Executors Framework*, a *Fork-Join Framework*, *Reactive Programming* (*RxJava* e *Reactor*) e o *Google's Thread Weaver*. Trabalhos relacionados destacam as abordagens únicas e os desafios específicos de cada *framework*, assim como as métricas de desempenho em diferentes contextos.

A *Executors Framework* permite o uso eficiente de *thread pools*. Trabalhos como os de Nobakht (2011) demonstram que, ao delegar a criação e gestão de *threads* para pools configuráveis, a *Executors Framework* reduz significativamente o *overhead* associado a tarefas concorrentes em sistemas de média complexidade e em cenários de carga variável, o *CachedThreadPool* apresenta tempos de execução até 30% menores em comparação com abordagens tradicionais, enquanto o *FixedThreadPool* garante estabilidade em sistemas previsíveis. Contudo, desafios como a necessidade de configurar adequadamente os limites de *threads* e monitorizar a utilização das mesmas podem limitar sua eficácia em sistemas de larga escala (Nobakht, 2011).

A *Fork-Join Framework*, foi projetado para tarefas recursivas que seguem o padrão *divide-and-conquer*. Estudos recentes, como os de Tramontana e Midolo (2023), demonstram que esta *framework* permite uma utilização quase total da capacidade de processamento de sistemas *multicore*, alcançando até 95% de eficiência em tarefas altamente paralelas (Midolo & Tramontana, 2023). No entanto, nos trabalhos de Choksuchat e Chantrapornchai (2016, apontam que a sobrecarga associada à sincronização entre subtarefas pode aumentar significativamente em sistemas onde a divisão de trabalho não é uniforme. Em *benchmarks*, a *framework* obteve um Tempo Médio de Execução de 10 ms para tarefas equilibradas, mas apresentou tempos de execução até 40% mais altos em cenários onde as tarefas têm dependências significativas (Choksuchat & Chantrapornchai, 2016).

As frameworks de *Reactive Programming*, como *RxJava* e *Reactor*, destacam-se pela abordagem declarativa para a gestão de fluxos de dados assíncronos. Trabalhos como os de Ponge et al. (2021) mostram que estas ferramentas são especialmente eficazes em sistemas distribuídos, como aplicações de streaming de dados e arquiteturas de *microservices*. O suporte à *backpressure* é um dos recursos mais valiosos destas *frameworks*, evitando sobrecargas e garantindo consistência nos fluxos de dados. Comparativamente, *RxJava* demonstrou uma latência média de 5 ms em fluxos simples, enquanto o *Reactor* mostrou melhor desempenho em cenários de alta concorrência, com redução de até 20% nos tempos de execução. Contudo, desafios como a curva de aprendizagem e a complexidade de *debug* ainda são limitadoras para sua adoção em larga escala (Ponge et al., 2021).

O *Google's Thread Weaver* oferece uma abordagem diferenciada para a verificação de interações entre *threads*, sendo projetado para detetar falhas concorrentes em sistemas complexos. Trabalhos como os de Touhafi et al. (2018) destacam sua eficácia em cenários de alta complexidade, onde *race conditions* e *deadlocks* são difíceis de reproduzir. Apesar da sua

elevada precisão, os estudos apontam para um tempo médio de configuração inicial de até 15% maior em comparação com outras *frameworks*, devido à sua dependência de modelos detalhados do sistema. Em contrapartida, em sistemas distribuídos de larga escala, o *Thread Weaver* demonstrou reduzir em até 25% o número de falhas não detetadas, consolidando-se como uma ferramenta indispensável para aplicações críticas (Cicirelli et al., 2022; Touhafi et al., 2018).

3.4 Casos de Estudo: Aplicação das Técnicas em Ambientes Reais

A análise empírica conduzida, com recurso à ferramenta *JCStress*, permitiu verificar a robustez e previsibilidade de várias construções concorrentes da linguagem Java. Contudo, em engenharia de *software*, a validade teórica ou experimental das técnicas apenas se revela útil quando acompanhada de aplicabilidade real. Neste capítulo são analisados três domínios representativos — sistemas financeiros, infraestruturas *IoT* e motores de jogo em tempo real — nos quais a programação concorrente é não só prevalente, como crítica. Estes estudos de caso procuram estabelecer uma ponte entre os resultados obtidos em ambiente controlado e as necessidades concretas de software em produção.

3.4.1 *Microservices* Financeiros

Sistemas financeiros distribuídos, como plataformas de transferências bancárias, *gateways* de pagamento ou processamento de ordens de débito, estão entre os contextos mais exigentes no que respeita à consistência concorrente. Cada transação envolve múltiplos acessos a saldos partilhados, cruzamento de dados com registos externos e preservação de invariantes, como "nenhuma conta pode ficar com saldo negativo".

Exigências do domínio:

- Atomicidade e isolamento transacional.
- Tolerância a falhas sem perda de consistência.
- Capacidade de escalar horizontalmente sem comprometer integridade.

Técnicas aplicadas (Joshi, 2022):

- *ReentrantLock* é frequentemente usado para garantir que operações como débito e crédito sobre a mesma conta não se sobrepõem. A sua capacidade de suportar políticas de equidade e bloqueios com *timeout* permite maior controlo em sistemas com elevada contenção.
- *ThreadLocal* isola o contexto de execução de cada pedido, permitindo manter, por exemplo, o *token* do utilizador, *logs* de auditoria ou metadados do pedido sem interferência entre *threads*.

- *AtomicInteger* pode ser útil para contadores internos (ex: número total de transações processadas), evitando *locks* dispendiosos.

A utilização combinada de *ReentrantLock* para recursos críticos e *ThreadLocal* para isolamento de sessão revela-se robusta e escalável. Contudo, erros de implementação (como esquecer *unlock()* num bloco de exceção) podem introduzir *deadlocks*. O uso de técnicas declarativas (como transações com *ACID* via bases de dados) pode ser necessário para garantir durabilidade e *rollback* automático.

3.4.2 Infraestruturas IoT

Sistemas *IoT* envolvem a recolha de dados em tempo real a partir de milhares de dispositivos heterogéneos, com restrições de energia, conectividade e processamento. Um exemplo concreto é uma rede de sensores ambientais (temperatura, humidade, CO_2) que envia continuamente medições para uma unidade central.

Exigências do domínio:

- Baixa latência de agregação.
- Garantia de integridade temporal e espacial das leituras.
- Eficiência em dispositivos com recursos limitados.

Técnicas aplicadas (Dragoni et al., 2017):

- *ThreadLocal* permite encapsular temporariamente o contexto de leitura de cada sensor (*timestamp*, *checksum,* origem), evitando escrita concorrente.
- *AtomicInteger* pode ser usado para contagem eficiente de pacotes por sensor, útil para deteção de falhas ou análise estatística.
- *synchronized* é simples e eficaz para controlar o acesso ao buffer de agregação de leituras, embora não escale bem com elevado número de fontes.

Esta abordagem é funcional em redes médias, mas limita-se em larga escala. Técnicas como *lock-free* data *structures*, *message queues* ou *streams* assíncronos (como *Apache Kafka*) oferecem alternativas mais escaláveis. A introdução de mecanismos de *backpressure* pode ser necessária para controlar sobrecargas.

3.4.3 Gaming Multithread

Motores de jogo modernos são arquiteturas paralelas intensivas, nas quais múltiplos subsistemas — física, lógica de jogo, IA, entrada do utilizador, som — operam simultaneamente. A necessidade de manter 60 ou mais *frames* por segundo torna qualquer contenção inaceitável.

Exigências do domínio:

- Latência mínima (<16ms por frame).
- Consistência visual e física entre threads.
- Evitar jank, race conditions e frame drops.

Técnicas aplicadas (Hera & Jackson, 2024):

- *ThreadLocal* armazena informação transitória de cada entidade jogável (*input*, estado temporário), evitando escrita cruzada.
- *ReentrantLock* é usado para proteger cálculos críticos em motores físicos, como colisões e movimentação de múltiplos objetos interdependentes.
- *synchronized*, quando usado, restringe-se a estruturas partilhadas que não impactem diretamente o ciclo do jogo (como *logs*, *matchmaking*).

A complexidade dos motores modernos exige que estas técnicas coexistam com paradigmas mais avançados, como *task graphs*, *entity component systems* ou mesmo programação funcional concorrente. O uso excessivo de *locks* pode degradar o desempenho de forma crítica. O balanceamento entre consistência e fluidez deve ser decidido por perfil de risco e tipo de jogo.

3.4.4 Comparação entre os Casos de Estudo

A análise dos três domínios apresentados — sistemas financeiros, infraestruturas IoT e motores de jogo multithreaded — revela padrões distintos na aplicação das técnicas concorrentes, refletindo a diversidade de requisitos não funcionais em sistemas concorrentes reais. Esta secção compara os casos em função das exigências de consistência, latência, escalabilidade destacando os critérios que orientam a escolha da técnica mais adequada.

Apesar de todos os domínios exigirem algum grau de isolamento e exclusão mútua, os fatores críticos variam:

- Nos sistemas financeiros, a prioridade recai sobre a consistência absoluta e controlo rigoroso de concorrência, justificando o uso de *ReentrantLock* para operações críticas e *ThreadLocal* para isolamento de sessão.

- Em contextos *IoT*, o foco está na eficiência leve e escalabilidade, com *AtomicInteger* e *ThreadLocal* a permitirem processamento paralelo de sensores sem custos elevados de sincronização.

- No caso dos motores de jogo, a latência mínima e paralelismo previsível são essenciais, privilegiando técnicas que evitem contenção global, como o uso seletivo de *ThreadLocal* e *ReentrantLock* localizados em subsistemas críticos.

Adicionalmente, os estudos de caso demonstram que a complexidade da técnica deve ser proporcional ao nível de contenção e ao impacto de falhas. Enquanto *synchronized* se mantém útil pela sua simplicidade, a sua escalabilidade limitada torna-o menos adequado para domínios com elevada pressão concorrente.

A Tabela 3 resume as principais conclusões desta comparação transversal:

Tabela 3 - Comparação de Casos de Estudo

Dominio	**Técnicas**	**Vantagens**	**Limitações**	**Recomendações**
Financeiro	*ReentrantLock, ThreadLocal, AtomicInteger*	Elevada robustez Controlo sobre bloqueio Isolamento de contexto	Suscetível a *deadlocks* se mal implementado	Combinação de *locks* explícitos com isolamento local
IoT	*ThreadLocal, AtomicInteger, synchronized*	Leveza Fácil integração em *firmware* Contagem eficiente	Escalabilidade limitada Baixa tolerância a falhas	Útil em redes médias Considerar sistemas reativos
Gaming	*ThreadLocal, ReentrantLock, synchronized*	Baixa latência Paralelismo previsível	Difícil de gerir em larga escala Possível impacto em FPS	Evitar *locks* globais Usar ECS ou pipelines paralelos

Esta comparação permite concluir que não existe uma técnica universalmente superior, mas sim padrões de adequação entre técnicas e requisitos de cada domínio. A seleção correta deve considerar não apenas a robustez teórica, mas também fatores práticos como complexidade da implementação, tolerância a falhas e impacto no desempenho.

4 Comparações dos Mecanismos de Concorrência

A comparação das ferramentas de programação concorrente é fundamental para entender as suas capacidades, aplicações e limitações em cenários diversos. Este tópico aborda as principais características das ferramentas analisadas*,* técnicas e *frameworks.* A análise foca-se em fatores como eficiência na deteção de problemas, impacto no desempenho, escalabilidade e facilidade de integração, proporcionando uma visão clara das suas forças, fraquezas e aplicações em sistemas críticos e altamente concorrentes.

Para garantir uma análise comparativa rigorosa e orientada para a aplicabilidade prática das ferramentas de deteção de concorrência, foram definidos critérios que cobrem múltiplas dimensões relevantes para ambientes reais de desenvolvimento. O foco principal permite identificar se a ferramenta se orienta para a deteção de condições de corrida, interleavings inválidos ou outros erros relacionados com concorrência. Os cenários de aplicação visam compreender em que contextos ou fases do ciclo de vida do software a ferramenta é mais eficaz (ex: testes unitários, análise estática, *runtime*). Os métodos de aplicação (como análise estática ou observação em tempo de execução) influenciam diretamente a complexidade da sua integração e o tipo de garantias oferecidas. O critério relativo ao número de falsos positivos é crucial para avaliar a fiabilidade prática da ferramenta, evitando ruído excessivo em *pipelines* de qualidade. A latência e o tempo médio de execução são indicadores críticos de desempenho, especialmente relevantes em contextos de *CI/CD* ou testes de carga. A capacidade de suporte a ambientes distribuídos permite aferir se a ferramenta se adapta a sistemas modernos e comunicação remota. O uso de recursos (memória e *CPU*) e a escalabilidade determinam a viabilidade da sua adoção em sistemas de grande dimensão. Por fim, a capacidade de integração com *IDEs*, *frameworks* de teste e outras ferramentas *DevOps* é essencial para garantir que estas soluções são compatíveis com fluxos de trabalho industriais. Em conjunto,

estes critérios permitem uma avaliação transversal, sistemática e comparável das ferramentas estudadas, cobrindo tanto o desempenho técnico como a sua utilidade prática.

4.1 Comparação de Ferramentas de Concorrência em Java

A análise comparativa das ferramentas de programação concorrente em Java – *Java Pathfinder (JPF), ThreadSanitizer, FindBugs com Concurrency Bug Detector, Seagence e JCStress* – evidencia as suas abordagens distintas para identificar e resolver problemas de concorrência. A partir desta análise foi elaborada a Tabela 4, e foram avaliados critérios técnicos, como os falsos positivos, o tempo médio de execução, os cenários de aplicação, o suporte a ambientes distribuídos e a utilização de recursos, o que permitiu identificar as forças e limitações de cada ferramenta.

Tabela 4 - Comparação de Ferramentas

Critérios	***Java Pathfinder***	***ThreadSanitizer***	***FindBugs***	***Seagence***	***JCStress***
Foco Principal	Verificação de modelos e *interleavings* de *threads*	Detecção dinâmica de *race conditions*	Análise estática de *bugs* de concorrência	Monitorização em tempo real de falhas	Testes microcontrolados de *interleavings* e visibilidade de memória
Cenários de Aplicação	Sistemas críticos	Ambientes de produção	Projetos com *pipelines* de análise estática	Sistemas distribuídos e altamente concorrentes	Validação de bibliotecas concorrentes e testes controlados
Métodos de Aplicação	Simulação sistemática	Análise em tempo de execução	Análise estática baseada em padrões	Análise contínua de *logs*	Definição manual de testes com atores e árbitros
Número de Falsos Positivos	1-3%	5-10%	10-20%	3-5% (dependente dos *logs*)	<1%
Latência	Alta (20ms)	Moderada (10ms)	Alta (25ms)	Baixa (5ms)	Baixa (5ms)
Tempo Médio de Execução	5 horas	15 minutos	10 minutos	20 minutos a 1 hora	Muito variável
Suporte a Ambientes Distribuídos	Limitado a sistemas locais	Parcial	Limitado	Total	Limitado
Uso de Recursos	Elevado	Moderado	Baixo	Moderado	Baixo
Escalabilidade	Escalabilidade limitada devido ao problema de explosão de estados	Boa escalabilidade	Boa escalabilidade em casos simples	Alta escalabilidade em sistemas distribuídos	Escalabilidade limitada a testes unitários
Capacidade de Integração	Dificuldade em integrar a pipelines CI/CD	Fácil integração com sistemas modernos	Integração direta em ambientes *DevOps*	Integração com sistemas baseados em *logs*	Integração via Maven/Gradle em projetos de teste

As ferramentas analisadas são adequadas para diferentes cenários. O *Java Pathfinder (JPF)* é amplamente utilizado em sistemas críticos, como software médico e aeroespacial, onde a integridade das operações concorrentes é crucial (Visser et al., 2003). A abordagem exaustiva do JPF permite identificar falhas complexas, como *deadlocks* em situações raras. Contudo, o seu elevado consumo de recursos limita a aplicabilidade em sistemas modernos de alta concorrência.

O *ThreadSanitizer*, devido à sua análise em tempo de execução, é ideal para ambientes de produção que requerem baixa latência e alta eficiência na identificação de problemas (Serebryany & Iskhodzhanov, 2009). A sua utilização em plataformas dinâmicas, como sistemas

de *streaming*, permite detetar rapidamente *race conditions* e evitar interrupções de serviço. No entanto, a maior incidência de falsos positivos pode dificultar a análise em aplicações com interações complexas.

O *FindBugs* adapta-se bem a projetos que utilizam pipelines de desenvolvimento baseados em análise estática como em ambientes *DevOps*. A capacidade de detetar padrões de comportamentos problemáticos é útil em fases iniciais de desenvolvimento, prevenindo a introdução de erros críticos. Contudo, a escalabilidade é limitada em cenários mais dinâmicos e distribuídos, e a alta taxa de falsos positivos pode afetar a confiança nos seus resultados (Serebryany & Iskhodzhanov, 2009).

O *Seagence* é amplamente adotado em sistemas distribuídos e arquiteturas baseadas em *microservices* devido ao suporte completo a ambientes distribuídos e à monitorização contínua de *logs*. Em plataformas de *e-commerce*, onde a escalabilidade e o tempo de resposta são fatores críticos, o *Seagence* demonstra eficiência na análise de grandes volumes de *logs*. Contudo, em sistemas onde os *logs* são incompletos ou inconsistentes, a eficácia do Seagence pode ser comprometida (Bitla, 2023).

O *JCStress*, por sua vez, representa uma abordagem distinta e complementar. Desenvolvido no contexto do projeto *OpenJDK*, o *JCStress* é uma ferramenta de testes concorrentes micro controlados que permite avaliar o comportamento de construções como *AtomicInteger*, *ReentrantLock* e *ThreadLocal*, testando diretamente as garantias fornecidas pelo modelo de memória do Java. A ferramenta executa cada teste milhares de vezes, explorando diferentes *interleavings* entre *threads* de forma sistemática. Embora não se destine à análise de sistemas completos, apresenta elevada precisão e baixa taxa de falsos positivos em cenários bem definidos, sendo amplamente utilizada para validar bibliotecas concorrentes e comportamentos críticos a nível de execução (Andrianov & Mutilin, 2024; Goetz et al., 2006).

O Java Pathfinder destaca-se pela sua precisão elevada (1-3%) devido à abordagem de simulação sistemática de *threads* (Spathoulas, 2014; Visser et al., 2003). O *ThreadSanitizer* e o *FindBugs* apresentam maior incidência de falsos positivos (5-10% e 10-20%, respetivamente) devido às limitações inerentes às análises dinâmicas e estáticas (Serebryany & Iskhodzhanov, 2009). O *Seagence*, ao depender da qualidade dos *logs*, mantém uma taxa competitiva (~3-5%) em sistemas bem configurados (Bitla, 2023).

A latência média reflete o tempo necessário para cada ferramenta identificar problemas de concorrência nos cenários testados, sendo uma métrica essencial para avaliar a eficácia em tempo real. Neste contexto, o *Seagence* destaca-se como a solução mais eficiente, com uma latência média de apenas 5 ms, devido à sua capacidade de análise contínua baseada em eventos (Bitla, 2023). Em contrapartida, o Java Pathfinder apresenta maior latência, com uma média de 20 ms, devido à sua abordagem exaustiva de verificação de estados sendo mais adequado para aplicações onde a precisão é priorizada sobre a rapidez (Spathoulas, 2014).

O uso médio de recursos abrange o consumo de memória e CPU durante a execução das ferramentas. O *ThreadSanitizer* e o *Seagence* demonstram um equilíbrio notável entre

eficiência e desempenho, mantendo um consumo moderado mesmo em cenários de alta concorrência (Bitla, 2023; Serebryany & Iskhodzhanov, 2009). Já o Java Pathfinder apresenta elevado uso de recursos, principalmente memória, devido à complexidade da análise sistemática do espaço de estados (Spathoulas, 2014).

Ferramentas como o *ThreadSanitizer* e o *FindBugs* destacam-se pela eficiência no tempo de execução (10-15 minutos), enquanto o *Java Pathfinder* pode levar até 5 horas para analisar sistemas complexos devido ao problema da *State Space Explosion* (Visser et al., 2003). O *Seagence* apresenta tempos variáveis (20 minutos a 1 hora), dependendo do volume de *logs* processados e da configuração do sistema (Bitla, 2023).

O *Seagence* é a ferramenta com maior escalabilidade e suporte completo a sistemas distribuídos, sendo adequada para arquiteturas modernas baseadas em *microservices*. O *ThreadSanitizer* e o *FindBugs* também apresentam boa escalabilidade em cenários mais simples, mas o *Java Pathfinder* enfrenta dificuldades devido ao problema de State Space Explosion (Visser et al., 2003). Em termos de integração, o *Seagence* e o *ThreadSanitizer* destacam-se pela fácil adaptação a sistemas modernos, enquanto o *JPF* apresenta maiores dificuldades devido à sua abordagem exaustiva (Bitla, 2023).

4.2 Comparação de Técnicas de Programação Concorrente

A comparação das técnicas de programação concorrente em Java – ***Sincronização de Blocos e Métodos***, ***CompletableFuture***, ***ReentrantLock***, ***Atomic Variables*** *e* ***ThreadLocal*** – destaca as abordagens distintas utilizadas para lidar com problemas de concorrência. Estas técnicas variam em termos de aplicabilidade, eficiência e escalabilidade, atendendo a necessidades específicas de sistemas *multithread*. A Tabela 5 fornece uma visão detalhada das principais características de cada técnica.

Tabela 5 - Comparação de Técnicas

Critério	**Sinc.**	***Completable Future***	***Reentrant Lock***	***Atomic Variables***	***ThreadLocal***
Foco	Controlo do acesso a secções críticas do código	Programação assíncrona	Gestão explícita de *locks*	Operações atómicas em variáveis	Isolamento de variáveis entre *threads*
Cenários de Aplicação	Sistemas críticos e operações sensíveis	Sistemas assíncronos e tarefas paralelas	Seções críticas em sistemas concorrentes	Atualizações seguras em variáveis partilhadas	Configuração e estado local de *threads*
Método	Bloqueio automático com uso do *synchronized*	Encadeamento de tarefas através de *futures*	Controlo manual de *locks*	Operações Atómicas	Criação de variáveis locais para *threads*
Eficiência	Boa em baixa concorrência, mas limitada sob alta concorrência	Alta em tarefas assíncronas	Moderada em alta concorrência	Alta, com baixa contenção	Alta, elimina interferência entre *threads*
Tempo Médio de Execução	30 ms	10 ms	20 ms	5 ms	2 ms
Latência	15 ms	8 ms	12 ms	2 ms	10 ms
Taxa de Falsos Positivos	5%	2%	4%	1%	3%
Uso Médio de Recursos	Moderado	Moderado	Moderado	Baixo	Moderado
Capacidade de Integração	Simples em cenários locais	Fácil integração com fluxos assíncronos	Requer configuração manual	Simples em operações unitárias	Diretamente aplicável a sistemas de *threads*
Suporte a Ambientes Distribuídos	Limitado	Alto	Moderado	Limitado	Limitado
Estudo	Baixo	Moderado	Alto	Baixo	Moderado
Escalável	Limitada devido à contenção	Alta	Moderada	Moderada	Limitada

As técnicas analisadas possuem diferentes abordagens para resolver problemas concorrentes. A Sincronização de Blocos e Métodos é uma técnica amplamente utilizada para controlar o acesso a secções críticas do código, sendo particularmente útil em sistemas críticos onde a

integridade dos dados é essencial (Goetz et al., 2006; Midolo & Tramontana, 2023). Em sistemas financeiros, onde operações simultâneas podem comprometer a consistência das transações, a sincronização de métodos oferece uma solução confiável, mas pode ser limitada em ambientes com alta carga devido à contenção de *locks*. O *CompletableFuture* destaca-se em tarefas assíncronas e sistemas reativos, permitindo maior escalabilidade (Ahmed et al., 2020; Fu et al., 2018). É amplamente adotado em plataformas de *streaming* e aplicações baseadas em *microservices*, onde a capacidade de encadear tarefas assíncronas reduz o impacto no desempenho geral. Já o *ReentrantLock* é indispensável em cenários que requerem controle detalhado de *locks*, como operações financeiras complexas. Esta técnica é particularmente útil também em aplicações que necessitam alta personalização na gestão de bloqueios, como sistemas de reserva de voos, onde a precisão é crítica para evitar conflitos de reservas (Midolo & Tramontana, 2023). As *Atomic Variables* são mais adequadas para atualizações rápidas e seguras em variáveis partilhadas, enquanto o *ThreadLocal* oferece isolamento de dados entre *threads*, sendo amplamente utilizado em configurações de autenticação (Fu et al., 2018).

A Sincronização de Blocos e Métodos utiliza a palavra-chave *synchronized*, que garante o bloqueio automático de secções críticas do código, mas pode levar a problemas de contenção em alta concorrência (Goetz et al., 2006; Midolo & Tramontana, 2023). Em contraste, o *CompletableFuture* baseia-se no encadeamento de tarefas e não bloqueia *threads*, garantindo alta eficiência (Ahmed et al., 2020; Fu et al., 2018). O *ReentrantLock* permite maior flexibilidade na gestão de *locks*, mas pode ter desempenho moderado em situações de contenção (Midolo & Tramontana, 2023). As *Atomic Variables* e o *ThreadLocal* destacam-se pela sua eficiência, reduzindo a interferência entre *threads* em sistemas de média e baixa complexidade. No caso de sistemas de IoT, onde a frequência de operações atómicas é alta, *Atomic Variables* são essenciais para evitar conflitos de atualização em dados partilhados (Fu et al., 2018).

A latência média mede o tempo necessário para a execução de operações concorrentes em cada técnica. As *Atomic Variables* destacam-se nesta métrica, com uma latência média de apenas 2 ms, devido à ausência de bloqueios explícitos, enquanto a sincronização de blocos com *synchronized* apresenta 15 ms devido à sua natureza bloqueante (Fu et al., 2018; Midolo & Tramontana, 2023).

O uso médio de recursos, que avalia o consumo de memória e CPU, também reflete diferenças significativas. As técnicas com bloqueio explícito, como *synchronized* e *ReentrantLock*, apresentam maior consumo de memória em cenários de alta concorrência, enquanto as variáveis atómicas demonstram ser a técnica mais eficiente, com consumo baixo, ideal para aplicações de baixa latência. Em sistemas de alta frequência, como transações bancárias em tempo real, esta diferença de latência pode ser decisiva para a escolha da técnica mais apropriada (Goetz et al., 2006; Midolo & Tramontana, 2023).

A taxa de falsos positivos avalia a precisão de cada técnica na identificação de problemas concorrentes. As *Atomic Variables* apresentam a menor taxa de falsos positivos (1%), uma vez que as operações atómicas garantem consistência sem necessidade de bloqueios adicionais. Por outro lado, *synchronized* apresenta uma taxa de falsos positivos de 5%, devido à sua menor

capacidade de lidar com cenários altamente dinâmicos e distribuídos (Fu et al., 2018; Midolo & Tramontana, 2023).

As técnicas variam significativamente em termos de escalabilidade. O *CompletableFuture* apresenta excelente escalabilidade em ambientes *multicore* e distribuídos, enquanto a Sincronização de Blocos e Métodos é limitada devido à contenção de *locks (Ahmed et al., 2020)*. O *ReentrantLock* mantém uma boa escalabilidade, desde que a concorrência seja moderada. As *Atomic Variables* têm uma escalabilidade média, e são eficazes em operações unitárias, e o *ThreadLocal* é limitado em sistemas distribuídos, devido à complexidade na gestão de variáveis isoladas (Fu et al., 2018; Goetz et al., 2006). Em arquiteturas modernas baseadas em *microservices*, o *CompletableFuture* é preferido devido à sua capacidade de integrar fluxos assíncronos em pipelines distribuídas.

A Sincronização de Blocos e Métodos possui uma curva de aprendizagem mais baixa, tornando-se a técnica mais acessível para programadores iniciantes em programação concorrente (Goetz et al., 2006; Midolo & Tramontana, 2023). O *CompletableFuture* e o *ReentrantLock*, embora mais complexos, oferecem maior flexibilidade e integração com sistemas modernos (Ahmed et al., 2020; Fu et al., 2018). As *Atomic Variables* possuem integração simples, mas são limitadas a operações básicas. O *ThreadLocal* é diretamente aplicável a sistemas baseados em *threads*, mas o *debug* em sistemas distribuídos pode ser desafiador (Fu et al., 2018).

4.3 Comparação de *Frameworks*

A programação concorrente em Java depende amplamente de *frameworks* especializadas para abordar cenários de alto desempenho e escalabilidade. As *frameworks* analisadas – ***Executors Framework***, ***Fork-Join Framework***, ***Reactive Programming*** e ***Google's Thread Weaver*** – destacam-se pelas suas abordagens distintas para lidar com tarefas concorrentes, variando em termos de aplicabilidade, eficiência e capacidade de integração. A Tabela 6 resume as principais características destas *frameworks*.

Tabela 6 - Comparação de Frameworks

Critérios	**Executors Framework**	**Fork-Join Framework**	**Reactive Programming**	**Google's Thread Weaver**
Foco	Gestão de *thread pools* e tarefas concorrentes	Paralelismo em tarefas recursivas	Gestão de fluxos assíncronos e *non-blocking*	Verificação de interações entre *threads*
Cenários de Aplicação	Sistemas *multithread* e tarefas paralelas	Processamento intensivo de dados em *multicore*	Sistemas reativos e arquiteturas de *microservices*	Sistemas críticos com alta concorrência
Método	Controle de execução baseado em *thread pools*	*Divide and conquer*	Propagação de fluxos de eventos	Simulação de interações entre *threads*
Eficiência	Alta	Alta em ambientes *multicore*	Alta em fluxos assíncronos	Boa em sistemas de complexidade elevada
Tempo Médio de Execução	15 ms por tarefa	10 ms por tarefa	5 ms por evento	20 ms por simulação
Latência	10 ms	8 ms	5 ms	15 ms
Uso Médio de Recursos	Moderado	Alto	Moderado	Alto
Taxa de Falsos Positivos	3%	2%	3%	1%
Capacidade de Integração	Fácil integração em sistemas modernos	Requer adaptação em tarefas simples	Excelente integração com *frameworks* como *RxJava*	Limitada a sistemas de validação
Suporte a Ambientes Distribuídos	Moderado	Moderado	Total	Limitado
Escalabilidade	Boa	Boa	Boa	Limitada

A *Executors Framework* é amplamente utilizada para gerir *thread pools*, sendo ideal para sistemas *multithread* que necessitam de controlo eficiente sobre a execução das tarefas. É particularmente eficaz em sistemas de processamento em lotes, como aplicações financeiras, onde a alocação eficiente de threads garante um desempenho consistente mesmo sob cargas elevadas. Contudo, o seu uso pode ser limitado em sistemas distribuídos que exigem maior flexibilidade na gestão de tarefas (Goetz et al., 2006). Em contraste, a *Fork-Join Framework* é projetado para tarefas recursivas, como cálculos paralelos em ambientes *multicore*, utilizando o paradigma *divide-and-conquer* (Cunha et al., 2006; Goetz et al., 2006). Este modelo é amplamente utilizado em análises de dados científicos, onde grandes volumes de informação precisam ser divididos e processados em paralelo para garantir resultados em tempo útil. No entanto, o elevado consumo de recursos pode ser uma limitação em sistemas com restrições

computacionais. A *Reactive Programming*, implementada por *frameworks* como *RxJava* e *Reactor*, destaca-se na gestão de fluxos assíncronos em arquiteturas modernas, como sistemas de *microservices* (Ahmed et al., 2020). Por outro lado, o *Google's Thread Weaver* é mais especializado, focando-se na monitorização de interações entre *threads* em sistemas críticos, como controladores aeroespaciais e softwares financeiros (Imam & Sarkar, 2015).

O método de operação difere significativamente entre as *frameworks*. A *Executors Framework* utiliza *thread pools* configuráveis, permitindo o controlo detalhado sobre as tarefas concorrentes. A *Fork-Join Framework* divide tarefas em subtarefas menores que podem ser processadas em paralelo, o que otimiza o uso de recursos em sistemas *multicore* (Cunha et al., 2006; Goetz et al., 2006). Por exemplo, em algoritmos de processamento de imagens, como o cálculo de transformadas de Fourier, a Fork-Join Framework distribui eficientemente as operações entre múltiplos núcleos (Cunha et al., 2006). Já a *Reactive Programming* foca-se na propagação de eventos e elimina os bloqueios desnecessários e garante alta eficiência (Ahmed et al., 2020). O *Google's Thread* Weaver simula interações entre *threads*, e oferece uma abordagem robusta para identificar problemas de sincronização, mas com maior custo de execução em termos de tempo (Imam & Sarkar, 2015).

A latência média varia de forma significativa entre as soluções. A *Fork-Join Framework* e as frameworks de Reactive Programming, como *RxJava* e *Reactor*, apresentam os melhores tempos, com médias de 5ms, devido à sua abordagem eficiente de gestão de fluxos e tarefas recursivas (Ponge et al., 2021). Em contrapartida, a Executors Framework, embora eficiente, apresenta maior latência em cenários que requerem frequente escalonamento dinâmico de tarefas, como sistemas de monitorização em tempo real (Goetz et al., 2006). Já o *Thread Weaver*, por adotar uma análise exaustiva para a verificação de interações entre *threads*, apresenta uma latência mais elevada, de 15 ms, refletindo a complexidade de seu modelo (Imam & Sarkar, 2015).

O uso médio de recursos também reflete diferenças notáveis entre as frameworks. A *Fork-Join Framework* apresenta um consumo elevado de recursos, especialmente memória, devido ao seu modelo de divisão recursiva (Cunha et al., 2006). Em contraste, *frameworks* reativas, demonstram um consumo moderado, equilibrando escalabilidade com eficiência computacional. A *Executors Framework*, embora não tão otimizado quanto *Reactive Programming*, oferece uma boa relação entre desempenho e uso de recursos em sistemas *multicore* (Midolo & Tramontana, 2023). Por outro lado, o Thread Weaver, apesar de robusto, apresenta altos requisitos de recursos, limitando sua aplicabilidade em sistemas com restrições orçamentais ou energéticas, como dispositivos IoT (Imam & Sarkar, 2015).

A taxa de falsos positivos, um indicador da precisão das *frameworks*, destaca o *Thread Weaver* como a solução mais confiável, com uma taxa de apenas 1%. Este resultado deve-se à sua abordagem rigorosa na análise de interações concorrentes. Já a *Executors Framework* e as soluções de Reactive Programming apresentam taxas de falsos positivos de 3%, refletindo a sua menor especialização em detetar problemas altamente complexos (Midolo & Tramontana, 2023; Ponge et al., 2021).

A *Fork-Join Framework* e as ferramentas de *Reactive Programming* são altamente escaláveis, sendo ideais para sistemas distribuídos e arquiteturas *multicore*. Por exemplo, em grandes sistemas de comércio eletrónico, a integração do *Reactive Programming* permite lidar com fluxos massivos de dados em tempo real, oferecendo vantagens claras em relação a abordagens tradicionais (Midolo & Tramontana, 2023). A *Executors Framework* oferece boa escalabilidade em sistemas locais e moderada em ambientes distribuídos (Goetz et al., 2006). Por outro lado, o *Google's Thread Weaver* é limitado a aplicações específicas de validação, tornando-se menos escalável em sistemas amplamente distribuídos (Imam & Sarkar, 2015).

4.4 Conclusões das comparações

Com base na análise das ferramentas, técnicas e *frameworks* avaliados, foram identificadas várias áreas de melhoria que podem contribuir para a eficácia e eficiência da programação concorrente em Java. Estas propostas visam abordar as limitações observadas, otimizar o desempenho e melhorar a aplicabilidade das soluções em sistemas modernos e distribuídos.

O JPF é amplamente utilizado em sistemas críticos, como software médico e aeroespacial, onde a integridade das operações concorrentes é crucial. Por exemplo, a análise de *deadlocks* no desenvolvimento de sistemas de controlo para aeronaves comerciais tem beneficiado da abordagem exaustiva do JPF. O *ThreadSanitizer* é ideal para ambientes de produção que requerem baixa latência, sendo utilizado em sistemas de telecomunicações para identificar rapidamente *race conditions* em plataformas de *streaming*. O *FindBugs* é eficiente em projetos *DevOps*, ajudando empresas de desenvolvimento de software a detetar problemas antes do *deployment* em larga escala. Finalmente, o *Seagence* é amplamente adotado em sistemas de *e-commerce* e *microservices*, onde a escalabilidade é crítica para suportar milhões de transações simultâneas, reduzindo significativamente o tempo de resolução de falhas em sistemas distribuídos. Por fim, o JCStress revelou-se útil na validação de comportamentos concorrentes a nível de código, especialmente em bibliotecas de baixo nível.

A comparação entre as técnicas demonstra que cada uma é indicada para necessidades específicas em sistemas concorrentes. A Sincronização de Blocos e Métodos é amplamente utilizada em sistemas bancários para assegurar a consistência das transações financeiras. Por exemplo, em sistemas de pagamentos, essa técnica evita conflitos durante a atualização simultânea de saldos de contas. O *CompletableFuture* tem sido usado em sistemas de logística para monitorizar e atualizar o estado de entregas em tempo real, permitindo maior eficiência operacional. O *ReentrantLock* é adotado em sistemas de reservas online para gerir simultaneamente múltiplas requisições de clientes, garantindo a consistência dos dados e evitando conflitos de reservas. As *Atomic Variables* são empregadas em sistemas *IoT* para garantir atualizações rápidas e seguras em dispositivos sensoriais, enquanto o *ThreadLocal* é utilizado para isolar informações de sessão em aplicações web, garantindo segurança e eficiência em plataformas de e-commerce.

A escolha da *framework* depende diretamente dos requisitos do sistema. A *Executors Framework* é utilizada em sistemas de processamento de lotes, como relatórios financeiros de grandes empresas, onde o controlo eficiente de *threads* melhora a eficiência operacional. A *Fork-Join Framework*, por sua vez, tem sido implementado em pesquisas científicas para processamento paralelo de grandes volumes de dados, como na análise de genomas. As *frameworks* de *Reactive Programming*, como *RxJava* e *Reactor*, são amplamente utilizadas em plataformas de *streaming*, permitindo a gestão eficiente de fluxos de eventos, como vídeos em alta qualidade. Finalmente, o *Google's Thread Weaver* é aplicado em sistemas críticos, como softwares aeroespaciais, para validar interações entre *threads* e garantir a robustez de operações críticas.

As ferramentas e técnicas analisadas demonstram maturidade variada, com pontos fortes bem definidos e limitações relevantes, sobretudo ao nível da escalabilidade, precisão e compatibilidade com sistemas modernos. A seleção adequada da solução depende, em última instância, do tipo de sistema concorrente e dos requisitos específicos de desempenho e fiabilidade. Estas observações fundamentam as direções futuras de investigação, discutidas no capítulo seguinte.

5 Desenvolvimento e Resultados

Após a análise teórica e comparativa das ferramentas, técnicas e *frameworks* de programação concorrente em Java, procede-se neste capítulo à fase de implementação e validação prática. Esta etapa visa observar empiricamente os comportamentos concorrentes descritos anteriormente, através da reprodução de cenários reais e controlados que evidenciem as *race conditions*, inconsistências de visibilidade e outros problemas típicos da programação *multithreaded*.

Este capítulo encontra-se organizado em quatro secções. A primeira descreve o ambiente de testes e o enquadramento da ferramenta *JCStress*. Segue-se a apresentação dos casos de teste selecionados, baseados em exemplos reais documentados na literatura, nomeadamente em *Java Concurrency in Practice (Goetz et al., 2006)*. A terceira secção apresenta os resultados obtidos com base nas execuções efetuadas, incluindo a frequência de comportamentos esperados e anómalos. Por fim, procede-se à análise crítica dos resultados, identificando padrões de comportamento e implicações práticas para o desenvolvimento de software concorrente robusto.

5.1 Ambiente de testes e enquadramento do *JCStress*

A fase de validação recorreu a uma ferramenta de apoio que permite observar, de forma controlada, o comportamento de construções concorrentes em Java: o *JCStress*. Esta ferramenta, desenvolvida e mantida no âmbito do projeto *OpenJDK*, destina-se à deteção de comportamentos indesejáveis relacionados com *interleavings* de *threads*, como *race conditions*, inconsistências de visibilidade e violações do *JMM*.

A seleção da ferramenta *JCStress*, utilizada para a realização dos testes experimentais, foi determinada pela sua adequação ao tipo de análise pretendida neste trabalho. Apesar da existência de soluções amplamente reconhecidas, como o *JPF*, o *Seagence* ou o *ThreadSanitizer*,

estas abordagens operam frequentemente ao nível de aplicações completas, com foco em análises sistémicas ou rastreio de execuções reais.

Em contraste, o *JCStress* permite isolar e validar o comportamento de baixo nível de construções concorrentes específicas — como *synchronized*, *AtomicInteger* ou *ReentrantLock* — em ambientes controlados e repetíveis. Esta granularidade é essencial para os objetivos desta dissertação, que visam explorar de forma empírica a robustez e previsibilidade de técnicas concorrentes fundamentais. Cada teste envolve dois ou mais atores (*threads*) que interagem com variáveis partilhadas e um árbitro (*arbiter*) que avalia o estado final.

Ao possibilitar a análise de *interleavings* não determinísticos e a deteção de falhas subtis de sincronização em cenários, o *JCStress* constitui a ferramenta mais adequada para observar padrões de concorrência que, em contextos reais, podem passar despercebidos ou manifestar-se apenas em execuções raras.

Esta abordagem permite validar empiricamente construções e padrões analisados na revisão teórica, como por exemplo:

- Operações não atómicas (*counter++*),
- Blocos sincronizados (*synchronized*),
- Utilização de *AtomicInteger*,
- *Locks* explícitos com *ReentrantLock*,
- Isolamento com *ThreadLocal*, entre outros.

O ambiente de testes foi preparado num projeto *Maven* com o plugin oficial do *JCStress*, executando cada cenário milhares de vezes para garantir a cobertura estatística dos resultados. O projeto foi inicializado com o seguinte comando, utilizando o *archetype* oficial disponibilizado pelo *OpenJDK*:

```
mvn archetype:generate -DinteractiveMode=false \
-DarchetypeGroupId=org.openjdk.jcstress \
-DarchetypeArtifactId=jcstress-java-test-archetype \
-DarchetypeVersion=0.16 \
-DgroupId=org.sample \
-DartifactId=test \
-Dversion=1.0
```

Excerto de Código 1 - Comando de Geração do Repositório Maven

Este comando cria automaticamente a estrutura de diretórios, a configuração do *pom.xml* e um exemplo de teste funcional. O projeto foi depois ajustado com os testes definidos nesta dissertação.

A execução foi feita com o comando:

```
java -jar ./target/jcstress.jar -t <NomeDoTeste>
```

Excerto de Código 2 - Comando da Execução dos Testes

O *JCStress* imprime uma tabela com os diferentes resultados apenas quando o teste produz mais do que um resultado observado. Se o teste for determinístico (por exemplo, sempre **0,1**), a ferramenta não imprime a tabela de resultados, dado que não há variação estatística a apresentar.

Os resultados podem também ser recolhidos em ficheiros *.bin.gz* e convertidos ou analisados com ferramentas externas, embora nesta dissertação se tenha optado por analisar os *outputs* gerados diretamente na consola.

Os resultados foram analisados com base nas classificações atribuídas pelo *JCStress*: *ACCEPTABLE, ACCEPTABLE_INTERESTING e FORBIDDEN*, de acordo com as anotações definidas em cada teste. A frequência relativa de cada padrão de execução foi usada para identificar tendências, confirmar garantias de sincronização ou expor falhas potenciais.

Esta ferramenta não se destina à comparação direta com as restantes soluções avaliadas nos capítulos anteriores, mas sim a complementar a análise empírica da dissertação, permitindo ilustrar na prática as limitações, garantias e riscos associados a determinadas construções concorrentes. A sua utilização fundamenta-se nos conteúdos adquiridos ao longo da revisão de literatura e no seu alinhamento com os objetivos delimitados do trabalho.

5.2 Cenários de teste

Esta secção apresenta um conjunto de cenários de teste concebidos para validar empiricamente construções concorrentes em Java, com base em exemplos clássicos documentados na literatura, nomeadamente em Java Concurrency in Practice (Goetz et al., 2006).Cada cenário representa uma situação típica de acesso partilhado a recursos, onde a ausência ou inadequação de mecanismos de sincronização pode resultar em *race conditions*, inconsistências de leitura, ou violações do modelo de memória. Os testes foram implementados utilizando a ferramenta *JCStress*, permitindo simular múltiplos *interleavings* de execução de forma intensiva. Cada subsecção documenta o objetivo do teste, o padrão concorrente avaliado, as expetativas comportamentais, o código correspondente (resumido) e a ligação aos resultados discutidos na secção seguinte.

5.2.1 Teste de Incremento Não Atómico com Variável Partilhada

O objetivo deste teste é evidenciar, de forma clara e controlada, uma *race condition* clássica causada pela ausência de sincronização em operações de leitura-modificação-escrita sobre uma variável partilhada. Este tipo de erro é comum em programas *multithreaded* e representa uma das formas mais insidiosas e difíceis de detetar na programação concorrente.

O exemplo segue o padrão *UnsafeSequence* apresentado em *Java Concurrency in Practice*, no qual se destaca a vulnerabilidade de métodos aparentemente simples como *getNext(),* quando acedidos simultaneamente por várias *threads* sem qualquer proteção.

Este caso constitui o cenário base de erro, sobre o qual se comparam posteriormente soluções corretas com mecanismos como *AtomicInteger*, *ReentrantLock* e *ThreadLocal*.

5.2.1.1 Descrição técnica do teste

A classe ***UnsafeSequence*** contém um campo partilhado *value*, iniciado a zero, e um método *getNext()* que retorna o valor atual antes de o incrementar. Este método não utiliza qualquer forma de sincronização nem mecanismos atómicos, o que o torna vulnerável a interferência entre *threads*. Apesar de simples, esta implementação da operação *value++* traduz-se, em *bytecode*, numa sequência de três etapas: leitura do valor, incremento e escrita. Em ambientes concorrentes, se duas *threads* executarem esta operação ao mesmo tempo, ambas poderão ler o mesmo valor antes de o incremento ocorrer, resultando na atribuição do mesmo número a duas entidades distintas — uma violação direta do objetivo da sequência.

Este padrão de erro é subtil, pois a implementação parece correta à primeira vista. No entanto, em ambientes *multithreaded*, a combinação de leituras e escritas concorrentes sem proteção pode resultar em leituras sobrescritas e perda de atualizações.

O Excerto de Código 3 representa um teste realizado para se puder observar os comportamentos esperados.

```
package org.sample;

import org.openjdk.jcstress.annotations.*;
import org.openjdk.jcstress.infra.results.II_Result;

@JCStressTest
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE, desc = "Valores únicos em
sequência.")
@Outcome(id = "0, 1", expect = Expect.ACCEPTABLE, desc = "Valores únicos em
sequência (ordem invertida).")
@Outcome(id = "1, 1", expect = Expect.ACCEPTABLE_INTERESTING, desc = "Condição
de corrida: mesmo valor devolvido.")
@Outcome(id = "0, 0", expect = Expect.ACCEPTABLE_INTERESTING, desc = "Condição
de corrida: mesmo valor devolvido.")
@Outcome(expect = Expect.FORBIDDEN, desc = "Valores inválidos.")

@State
public class UnsafeSequenceTest {
      static class UnsafeSequence {
              private int value = 0;
              public int getNext() {
                  return value++;
              }
      }
      private final UnsafeSequence seq = new UnsafeSequence();

      @Actor
      public void actor1(II_Result r) {
           r.r1 = seq.getNext();
      }

      @Actor
      public void actor2(II_Result r) {
      r.r2 = seq.getNext();
      }
}
```

Excerto de Código 3 - Teste de Incremento Não Atómico com Variável Partilhada

5.2.1.2 Expetativas comportamentais

Em execução sequencial, duas chamadas a *getNext()* devolveriam valores consecutivos: **0** e **1**. Contudo, em contexto concorrente, sem sincronização, é possível que ambas as *threads* leiam o mesmo valor antes de qualquer incremento ocorrer. Como resultado, ambas devolvem o mesmo número, o que quebra a unicidade da sequência.

- **Resultado esperado (r1 = 0, r2 = 1 ou vice-versa):** comportamento correto, sem interferência.
- **Resultado problemático (r1 = r2 = 0 ou r1 = r2 = 1):** *race condition*, ambos os atores receberam o mesmo número.
- **Resultado inválido (r1 = r2 = 2 ou superior):** comportamento inesperado, causado por *interleaving* incorreto.

5.2.1.3 Resultados observados

Este teste foi executado sob carga intensiva, com milhares de iterações. Como será demonstrado posteriormente, observou-se uma percentagem significativa de execuções em que ambos os atores devolveram o mesmo valor, comprovando a existência de uma *race condition*. Estes resultados reforçam a necessidade de sincronização explícita ou de uso de construções atómicas em situações onde a unicidade de resultados entre *threads* é crítica.

5.2.2 Cenário Combinado com *AtomicInteger* e *ReentrantLock – HybridSequence*

Este teste visa simular um cenário realista onde a combinação de técnicas é necessária: o uso de *AtomicInteger* garante uma operação principal atómica (ex: gerar um número na sequência), enquanto *ReentrantLock* protege operações adicionais que dependem dessa sequência, como atualização de *logs,* notificações ou escrita em estrutura partilhada.

Este padrão é comum em aplicações que requerem baixo tempo de resposta na operação principal, mas sincronização estrita em ações complementares. A literatura especializada recomenda esta abordagem híbrida como forma de equilibrar desempenho e segurança em ambientes concorrentes.

5.2.2.1 Descrição técnica do teste

A classe *HybridSequence* utiliza um campo *AtomicInteger* para gerar números únicos e um *ReentrantLock* para proteger uma operação adicional (fictícia) associada a esse número. No teste, cada ator invoca *getNextAndLog()*, que primeiro obtém o próximo número de sequência de forma atómica e, de seguida, entra numa secção crítica protegida para "registar" esse número num *array* partilhado.

Além da divisão clara entre a operação principal (incremento atómico com *AtomicInteger*) e a operação secundária protegida (*log* com *ReentrantLock*), importa referir que esta construção permite compor secções críticas reduzidas e bem localizadas, minimizando o tempo de contenção entre *threads*.

Este tipo de composição é especialmente útil em casos onde:

- a operação principal deve ser rápida, não bloqueante (ex: gerar ID),
- mas a operação associada exige consistência global (ex: registar o evento num ficheiro de log ou numa estrutura de dados comum).

Ao usar *AtomicInteger* para a sequência e *ReentrantLock* para a escrita protegida, evita-se o uso generalizado de *locks*, mantendo desempenho aceitável mesmo em cenários de elevada concorrência.

Esta simulação representa, por exemplo, o registo de eventos em memória partilhada após atribuição de identificadores únicos.

O Excerto de Código 4 representa um teste realizado para se puder observar os comportamentos esperados.

```
@JCStressTest
@Outcome(id = "0, 1", expect = Expect.ACCEPTABLE, desc = "Valores distintos e
registo isolado.")
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE, desc = "Valores distintos e
registo isolado.")
@Outcome(expect = Expect.FORBIDDEN, desc = "Valor duplicado ou interferência no
registo.")
@State
public class HybridSequenceTest {
      static class HybridSequence {
       private final AtomicInteger counter = new AtomicInteger(0);
       private final ReentrantLock lock = new ReentrantLock();
       private final int[] log = new int[2]; // simula registo partilhado
       private int index = 0;

       public int getNextAndLog() {
              int seq = counter.getAndIncrement();
              lock.lock();
              try {
                     log[index++] = seq; // operação protegida
                  } finally {
                     lock.unlock();
                  }
       return seq;
       }
      }
      private final HybridSequence seq = new HybridSequence();
      private final int[] results = new int[2];

      @Actor
      public void actor1(II_Result r) {
       results[0] = seq.getNextAndLog();
      }

      @Actor
      public void actor2(II_Result r) {
       results[1] = seq.getNextAndLog();
      }

      @Arbiter
      public void arbitro(II_Result r) {
       r.r1 = results[0];
       r.r2 = results[1];
      }
}
```

Excerto de Código 4 - Cenário Combinado com *AtomicInteger* e *ReentrantLock*

5.2.2.2 Expetativas Comportamentais

Espera-se que os números obtidos sejam sempre distintos **(0, 1 ou 1, 0)** e que não ocorra qualquer subscrição ou interferência no *array* de registo. Devido ao *AtomicInteger*, a atribuição do número é segura. E o *ReentrantLock* impede *race conditions* no momento da escrita no *array* de registo.

Adicionalmente, é importante destacar que este padrão permite separar a escalabilidade da geração de identificadores (graças ao *AtomicInteger*) da consistência dos efeitos colaterais controlados (via *ReentrantLock*). Isso reflete-se diretamente no comportamento observado nos testes:

- O AtomicInteger assegura que *actor1* e *actor2* recebem sempre valores distintos (**0, 1 ou 1, 0**);
- O *ReentrantLock* impede que ambas as *threads* modifiquem simultaneamente a estrutura *log[]*, garantindo que os índices são atualizados de forma sequencial e sem interferência.

A inexistência de **0, 0** ou **1, 1** nos resultados, bem como a ausência de corrupção no *array* de registo, é um sinal da robustez desta abordagem híbrida.

5.2.2.3 Resultados observados

Os testes realizados demonstraram que a combinação de *AtomicInteger* para geração de identificadores e *ReentrantLock* para proteção do registo em memória assegura a integridade e exclusividade dos dados. O padrão revelou-se eficaz para cenários com lógica híbrida, equilibrando desempenho e proteção.

5.2.3 Cenário com ThreadLocal e synchronized – SessionLogger

Este teste tem como objetivo demonstrar um cenário representativo de sistemas distribuídos ou aplicações *web* concorrentes, onde cada *thread* mantém o seu próprio estado (como uma sessão), mas interage pontualmente com recursos partilhados que requerem proteção. O exemplo modela um sistema onde cada *thread* mantém um contador local (*ThreadLocal*) e, após atualizar o seu estado, escreve um resumo num *log* partilhado, protegido por *synchronized*.

Este tipo de arquitetura é comum em *frameworks* como *servlets*, *microservices* e sistemas baseados em tarefas paralelas com persistência centralizada.

5.2.3.1 Descrição técnica do teste

A classe *SessionLogger* utiliza um *ThreadLocal<Integer>* para armazenar um contador isolado para cada *thread*. A operação *executeSession()* incrementa o contador da sessão da *thread* corrente e regista o valor num array global log, que simula um ficheiro ou estrutura de registo partilhada. O acesso ao log é protegido com *synchronized* para garantir que múltiplas *threads* não escrevem simultaneamente.

Além da separação entre o contexto isolado (*ThreadLocal*) e a partilha protegida (*synchronized*), este teste simula uma arquitetura prática comum em sistemas *multithreaded* modernos: sessões independentes que interagem com um recurso global.

O uso de *ThreadLocal* permite que cada *thread* mantenha uma cópia da variável *session*, assegurando que a operação *get()* e *set()* afeta apenas o seu próprio espaço de dados. Este modelo é amplamente usado em *frameworks* web como Spring, *Tomcat* ou servidores *REST*, onde dados da requisição ou do utilizador não devem colidir entre *threads*.

Já o método *synchronized* aplicado sobre o objeto (*this*) garante que, no momento de escrita no *array log*, apenas uma *thread* pode aceder à secção crítica, evitando sobrescritas ou conflitos de índice.

Esta combinação garante isolamento de contexto e integridade na comunicação *inter-thread*.

O Excerto de Código 5 representa um teste realizado para se puder observar os comportamentos esperados.

```
@JCStressTest
@Outcome(id = "0, 0", expect = Expect.ACCEPTABLE, desc = "Sessões isoladas
com escrita segura.")
@Outcome(expect = Expect.FORBIDDEN, desc = "Escrita duplicada ou
interferência no log.")
@State
public class SessionLoggerTest {
      static class SessionLogger {
        private final ThreadLocal<Integer> session =
               ThreadLocal.withInitial(() -> 0);
        private final int[] log = new int[2];
        private int logIndex = 0;
        public int executeSession() {
               int valor = session.get();
               session.set(valor + 1);
               synchronized (this) {
                      log[logIndex++] = valor; // escrita protegida
                   }
        return valor;
        }
      }

      private final SessionLogger logger = new SessionLogger();
      private final int[] results = new int[2];

      @Actor
      public void actor1(II_Result r) {
        results[0] = logger.executeSession();
      }

      @Actor
      public void actor2(II_Result r) {
        results[1] = logger.executeSession();
      }

      @Arbiter
      public void arbitro(II_Result r) {
        r.r1 = results[0];
        r.r2 = results[1];
      }
}
```

Excerto de Código 5 - Cenário com *ThreadLocal* e *synchronized*

5.2.3.2 Expetativas comportamentais

Espera-se que cada uma das *threads* inicie a sua execução com o valor local do ThreadLocal igual a 0, dado que o inicializador estático define este valor por omissão. Após a execução de *executeSession*(), o valor local é incrementado, mas o que é registado no log é o valor anterior (0), de forma isolada por *thread*.

O comportamento correto será, portanto, o resultado 0, 0, indicando que:

- Cada *thread* operou sobre o seu próprio contexto isolado, sem partilha de estado;
- A escrita no *log* foi realizada de forma sincronizada, sem sobreposição nem corrupção de índices.

Quaisquer resultados divergentes, como valores duplicados com sobrescrita (0, 0 com erro no índice) ou leituras inconsistentes do *ThreadLocal*, indicariam falhas na proteção da secção crítica ou no isolamento da sessão da *thread*, o que não é esperado neste cenário com proteção explícita.

5.2.3.3 Resultados observados

Durante os testes, observou-se que ambas as *threads* executaram a operação sobre a sua própria sessão, resultando consistentemente no valor **(0,0)**, sem qualquer forma de interferência no registo partilhado. O uso de *ThreadLocal* garantiu isolamento, enquanto a instrução *synchronized* protegeu a consistência do *log*.

5.2.4 Inconsistências de Memória sem Sincronização – *CacheCoherenceTest*

Este teste tem como objetivo explorar comportamentos inconsistentes resultantes da ausência de sincronização entre *threads*, não causados por *race conditions* diretas, mas sim por reordenação de instruções e incoerência entre caches de CPU. O exemplo simula uma situação clássica prevista no JMM, em que operações de escrita e leitura entre *threads* não ocorrem com garantias de visibilidade mútua.

Cada *thread* escreve numa variável (**a** ou **b**) e lê a outra, sem qualquer mecanismo de sincronização como *volatile*, *locks* ou blocos *synchronized*. Esta ausência de garantias de *happens-before* permite que os efeitos das escritas não sejam imediatamente visíveis às outras *threads*, revelando comportamentos contraintuitivos que, embora tecnicamente válidos, podem causar falhas severas em algoritmos concorrentes.

Este tipo de teste é fundamental para demonstrar que erros em concorrência não se limitam a *race conditions*, mas podem surgir por falta de visibilidade de memória, mesmo sem acesso simultâneo ao mesmo dado.

5.2.4.1 Descrição técnica do teste

A classe *CacheCoherenceTest* define duas variáveis partilhadas (a e b) que são modificadas por cada ator (*thread*), numa ordem espelhada: cada *thread* escreve numa variável e, de seguida, lê a outra. O resultado é registado numa estrutura *II_R*esult, que captura os valores lidos pelas duas *threads* após a execução.

O código não contém nenhuma forma de sincronização explícita, o que significa que o compilador ou a CPU têm liberdade para reordenar as instruções, ou manter os dados em caches locais antes de os propagar à memória principal. Assim, mesmo após uma escrita aparente, a leitura de outra *thread* pode retornar 0, indicando que a escrita ainda não foi "vista".

Este teste é baseado num padrão comum de demonstração das lacunas da visibilidade de memória, especialmente em arquiteturas multiprocessadoras.

O Excerto de Código 6 representa um teste realizado para se puder observar os comportamentos esperados.

```
@JCStressTest
@Outcome(id = "0, 0", expect = Expect.ACCEPTABLE_INTERESTING, desc = "Ambas
as leituras são zero – demonstra a inocência da reordenação/cache.")
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE, desc = "A escrita do Actor1
é vista pela leitura do Actor2, mas não vice-versa.")
@Outcome(id = "0, 1", expect = Expect.ACCEPTABLE, desc = "A escrita do Actor2
é vista pela leitura do Actor1, mas não vice-versa.")
@Outcome(id = "1, 1", expect = Expect.ACCEPTABLE_INTERESTING, desc = "Cada
ator viu a escrita do outro – sem reordenação.")
@Outcome(expect = Expect.FORBIDDEN, desc = "Valores inválidos.")
@State
public class CacheCoherenceTest {
      int a, b;

      @Actor
      public void actor1(II_Result r) {
      a = 1;
      r.r1 = b;
      }

      @Actor
      public void actor2(II_Result r) {
      b = 1;
      r.r2 = a;
      }
}
```

Excerto de Código 6 - Teste Inconsistências de Memória sem Sincronização

5.2.4.2 Expetativas comportamentais

Neste cenário, espera-se que o comportamento das *threads* varie consoante a ordem real de execução e a visibilidade dos dados em memória. A ausência de sincronização permite que ocorram quatro combinações de resultados:

- O resultado **1, 1** representa o comportamento ideal, em que cada *thread* viu a escrita da outra. Este resultado é coerente e demonstra que não houve reordenação nem atraso na propagação da informação.
- Os resultados **1, 0 e 0, 1** indicam que apenas uma das *threads* viu a alteração da outra, sendo este comportamento tecnicamente aceitável, mas revelador de assimetria na visibilidade.
- O resultado **0, 0** é o mais relevante e problemático: nenhuma *thread* viu a escrita da outra, mesmo após a sua própria escrita. Este comportamento ocorre devido à possível reordenação das instruções ou falha de coerência entre caches, e é um exemplo clássico de inconsistência de memória permitido pelo *Java Memory Model*.

Os resultados são classificados como *acceptable* e *interesting* no JCStress, e o caso **0, 0 e 1,1** são considerados *interesting* na literatura por ilustrar uma falha subtil de visibilidade de memória que pode comprometer invariantes lógicos em programas concorrentes.

5.2.4.3 Resultados observados

Os testes realizados com *JCStress* produziram os quatro resultados possíveis, com especial destaque para o **0, 0**, que ocorreu com frequência suficiente para validar a hipótese de reordenação e falha de visibilidade. Este comportamento confirma que, na ausência de mecanismos de sincronização, o JMM permite execuções onde as escritas de uma *thread* não são visíveis à outra, mesmo que ambas ocorram antes da leitura respetiva.

A observação deste tipo de inconsistente visibilidade é essencial para compreender porque mecanismos como *volatile*, *locks*, ou outras formas de sincronização são indispensáveis em projetos concorrentes.

5.2.5 Visibilidade com *synchronized – CacheCoherenceLocked*

Este teste propõe uma solução segura para o problema de visibilidade de memória demonstrado em *CacheCoherenceTest*. Em vez de confiar em garantias implícitas do compilador, a solução aplica blocos *synchronized* sobre um objeto comum para assegurar sincronização formal entre as *threads*, segundo o *Java Memory Model*.

A utilização de *synchronized* garante que as alterações feitas dentro de uma *thread* são visíveis a todas as outras *threads* que acedam ao mesmo bloco sincronizado, estabelecendo uma relação de *happens-before* e evitando tanto reordenação como cache incoerente.

5.2.5.1 Descrição técnica do teste

A classe *CacheCoherenceLocked* mantém a mesma estrutura que o exemplo original, mas adiciona blocos *synchronized (this)* em ambas as *threads*, quer na escrita, quer na leitura. Isso força a entrada numa secção crítica onde todas as operações são vistas como atómicas e visíveis entre si.

Esta abordagem não depende de variáveis *volatile* nem de estruturas adicionais, mantendo o código simples e completamente conforme com os mecanismos já discutidos no estado da arte da dissertação.

O Excerto de Código 7 representa um teste realizado para se puder observar os comportamentos esperados.

```
@JCStressTest
@Outcome(id = "1, 1", expect = Expect.ACCEPTABLE,
        desc = "Ambas as threads observaram a escrita da outra.")
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE,
        desc = "Actor1 viu a escrita de Actor2.")
@Outcome(id = "0, 1", expect = Expect.ACCEPTABLE,
        desc = "Actor2 viu a escrita de Actor1.")
@Outcome(id = "0, 0", expect = Expect.FORBIDDEN,
        desc = "Ambas as leituras falharam: não deveria ocorrer com
synchronized.")
@State
public class CacheCoherenceLocked {
    int a, b;
    @Actor
    public void actor1(II_Result r) {
        synchronized (this) {
            a = 1;
        }
        synchronized (this) {
            r.r1 = b;
        }
    }
    @Actor
    public void actor2(II_Result r) {
        synchronized (this) {
            b = 1;
        }
        synchronized (this) {
            r.r2 = a;
        }
    }
}
```

Excerto de Código 7 - Teste Visibilidade com *synchronized*

5.2.5.2 Expetativas Comportamentais

Com *synchronized* aplicado de forma simétrica, espera-se que o resultado **0,0** seja completamente evitado, já que a sincronização garante visibilidade de todas as alterações:

- **1,1**: resultado ideal e mais provável
- **1,0 ou 0,1**: possíveis apenas com assimetria de execução
- **0,0**: não deverá ocorrer, uma vez que a leitura ocorre após sincronização com escrita da outra *thread*

5.2.5.3 Resultados Observados

Nos testes realizados, os resultados **1,1, 1,0** e **0,1** foram consistentemente observados, enquanto o resultado **0,0** não ocorreu. Esta ausência confirma que o uso de blocos *synchronized* sobre um objeto comum é suficiente para garantir visibilidade e ordem de execução entre *threads*. A solução é leve, segura e bem documentada no ecossistema Java, sendo compatível com todas as técnicas já discutidas na dissertação.

5.3 Resultados e Análise

A presente secção apresenta e analisa os resultados obtidos com a execução dos testes definidos na secção anterior, recorrendo à *framework JCStress*. Cada teste foi concebido para avaliar a robustez e correção de diferentes construções concorrentes em Java, focando-se em dois tipos principais de anomalias: *race conditions* e inconsistências de visibilidade de memória.

Para cada cenário, a ferramenta executou milhares de iterações, cobrindo diversas *interleavings* possíveis entre as *threads* envolvidas. Os resultados foram classificados automaticamente como *ACCEPTABLE*, *ACCEPTABLE_INTERESTING* ou *FORBIDDEN*, de acordo com as anotações definidas nos próprios testes. Esta taxonomia permitiu identificar tanto os comportamentos esperados como os padrões sintomáticos de falhas concorrentes.

A análise baseia-se na frequência relativa de cada padrão observado, tendo em conta o número de amostras recolhidas em cada *fork* do teste. Sempre que relevante, os resultados são acompanhados por comentários técnicos que relacionam os comportamentos observados com as garantias fornecidas pelo *Java Memory Model* (JMM) e com os mecanismos de sincronização aplicados.

De notar que os testes com comportamento determinístico (isto é, que apresentaram apenas um único resultado ao longo de todas as execuções) não originam tabelas múltiplas no output do *JCStress*, mas ainda assim fornecem informação valiosa sobre a fiabilidade das construções testadas.

5.3.1 Teste de Incremento Não Atómico com Variável Partilhada

O teste *UnsafeSequence* teve como objetivo observar o comportamento de uma operação de incremento (*value++*) em contexto concorrente, sem qualquer forma de sincronização. A execução foi realizada em múltiplos *forks*, com milhões de amostras por instância, usando a *framework JCStress*.

Os resultados foram agrupados por padrão de execução observado. Foram escolhidas 5 iterações aleatórias para exemplo e o no fim o valor final das iterações. Os valores registados estão organizados na Tabela 7.

Tabela 7 - Resultados dos Testes de Incremento Não Atómico com Variável Partilhada

Fork	Resultado	Amostras	Frequência %	Descrição
1	0,0	23030440	42,74	Interesting
	0,1	16334670	30,32	Acceptable
	1,0	14515471	26,94	Acceptable
	1,1	0	0	Interesting
2	0,0	70312	0,29	Interesting
	0,1	730577	3,05	Acceptable
	1,0	23158682	96,66	Acceptable
	1,1	0	0	Interesting
3	0,0	16119307	28,44	Interesting
	0,1	21825719	38,51	Acceptable
	1,0	18731345	33,05	Acceptable
	1,1	0	0	Interesting
4	0,0	908529	2,18	Interesting
	0,1	1791208	4,31	Acceptable
	1,0	38903314	93,51	Acceptable
	1,1	0	0	Interesting
5	0,0	48666	0,26	Interesting
	0,1	18498211	96,98	Acceptable
	1,0	528214	2,77	Acceptable
	1,1	0	0	Interesting
Total	0,0	40210254	20,59	Interesting
	0,1	59667385	30,55	Acceptable
	1,0	95766026	49,02	Acceptable
	1,1	0	0	Interesting

5.3.1.1 Análise Crítica

Os resultados mostram que, em cerca de 20% das execuções, o método *getNext()* devolveu o mesmo valor a ambas as *threads* (**0,0**), revelando um claro caso de *race condition*. Este comportamento confirma que a operação *value++*, embora sintaticamente simples, não é atómica: envolve uma leitura, incremento e escrita — três passos que podem ser interrompidos ou intercalados por outra *thread*.

As execuções **0,1** e **1,0**, que somadas representam cerca de 79% dos casos, são consideradas *acceptable*, pois demonstram que cada *thread* obteve um valor distinto, ainda que a ordem de acesso tenha variado. No entanto, a ocorrência não desprezível de resultados *interesting* levanta sérias dúvidas quanto à fiabilidade da construção em ambientes reais, especialmente se os valores forem usados como identificadores únicos ou chaves.

A inexistência de resultados **1,1** (em que ambas as *threads* receberiam o mesmo valor após a primeira já o ter incrementado) sugere que, neste contexto específico, o *value++* é interrompido logo após a leitura inicial — reforçando a ideia de que o conflito ocorre na fase de leitura e não na propagação subsequente.

Esta variabilidade extrema é uma manifestação concreta da não determinismo inerente à programação concorrente, especialmente quando não são utilizados mecanismos de sincronização.

A frequência com que a saída (**0,0**) surge — variando de apenas 0,26% num *fork* até 42,74% noutro — pode ser explicada por fatores subjacentes à execução paralela, como:

- Escalonamento de *threads* pelo sistema operativo (que influencia a ordem e tempo de execução das instruções).
- Otimizações do compilador *JIT* da *JVM*, que pode reordenar instruções em tempo de execução.
- Efeitos de coerência de cache em arquiteturas *multicore*, que introduzem atrasos ou leituras desatualizadas entre núcleos distintos.

Esta dispersão estatística reforça a ideia de que falhas de concorrência não se manifestam sempre da mesma forma, podendo passar despercebidas em testes ocasionais ou em execuções que, por acaso, não provocam *interleavings* perigosos. Por isso, confiar em testes que "passam" sob determinadas condições sem analisar o comportamento interno pode levar a uma falsa sensação de segurança, sendo esta uma das razões que justifica o uso de ferramentas como o *JCStress*.

Estes dados confirmam o que foi antecipado no ponto 5.2.1.2: a ausência de sincronização torna esta construção insegura e imprevisível. A análise estatística reforça a conclusão de que este tipo de erro é não só possível, como provável.

5.3.1.2 Observações adicionais

A frequência do padrão 0,0 variou significativamente entre *forks* (de 0,26% até 42,74%), o que ilustra bem a natureza não determinística dos *interleavings*. Em alguns *forks*, a sincronização implícita da *JVM* ou características do escalonador do sistema operativo reduziram a probabilidade da colisão, mas nunca a eliminaram por completo.

Estes resultados evidenciam que confiar na ausência empírica de erro em poucas execuções não é suficiente para garantir segurança em código concorrente. A construção *UnsafeSequence* é claramente vulnerável e requer sincronização explícita para ser utilizada com segurança.

5.3.2 Cenário Combinado com *AtomicInteger* e *ReentrantLock* – *HybridSequence*

O teste *HybridSequence* foi concebido para validar uma construção concorrente que combina um contador seguro (*AtomicInteger*) com um mecanismo explícito de exclusão mútua (*ReentrantLock*) para a gestão de operações secundárias. O objetivo era avaliar se esta abordagem híbrida assegura valores distintos entre *threads* e protege corretamente os acessos partilhados ao registo de execução.

Durante a execução com *JCStress*, o teste demonstrou comportamento determinístico: todas as execuções resultaram em combinações de valores distintos (**0,1** e **1,0**), sem qualquer

ocorrência de valores repetidos como **0,0** ou **1,1**. Por este motivo, não foi gerada uma tabela de resultados múltiplos, conforme o comportamento padrão da ferramenta quando o teste não apresenta variação estatística relevante.

5.3.2.1 Análise Crítica

O teste *HybridSequence* apresentou um comportamento consistentemente correto, sem qualquer ocorrência de resultados classificados como *interesting* ou *forbidden*, o que demonstra de forma inequívoca a eficácia da construção testada. Esta implementação recorre à combinação de duas abordagens distintas: *AtomicInteger* para garantir atomicidade na geração da sequência, e *ReentrantLock* para assegurar exclusão mútua no acesso ao vetor partilhado de registo (*log[]*).

A atomicidade oferecida por *AtomicInteger* permite que múltiplas *threads* acedam ao contador sem necessidade de sincronização explícita, mantendo a integridade da sequência gerada mesmo sob elevada concorrência. Este comportamento é garantido pela implementação interna do método *getAndIncrement()*, que utiliza instruções de hardware *compare-and-swap* (CAS) para evitar colisões, sendo por isso classificado como *lock-free*.

Por outro lado, o acesso ao vetor *log[]* exige garantias de exclusividade, uma vez que múltiplas *threads* poderiam tentar escrever no mesmo índice, causando corrupção de dados ou sobrescrita. Para esse fim, a utilização de *ReentrantLock* garante que apenas uma *thread* por vez acede à secção crítica onde essa escrita ocorre. A escolha de *ReentrantLock* em vez de *synchronized* oferece ainda vantagens como controlo mais granular sobre a aquisição e libertação do bloqueio, bem como maior flexibilidade em cenários mais complexos.

Este padrão de composição entre mecanismos de sincronização — onde diferentes responsabilidades são protegidas com técnicas adequadas às suas características — reflete uma abordagem avançada de design concorrente, alinhada com os princípios discutidos em Java *Concurrency in Practice (Goetz et al., 2006)*. A prática de proteger apenas os pontos críticos e evitar sincronização desnecessária nas operações rápidas (como o incremento) contribui para um melhor desempenho global, reduzindo o risco de contenção de *threads*.

O comportamento observado nos testes valida, assim, a eficácia deste modelo híbrido tanto em termos de correção funcional (evita *race conditions* e corrupção de dados) como em termos de eficiência potencial (permite acesso concorrente seguro com mínima contenção).

5.3.2.2 Observações adicionais

Este teste confirma que a escolha criteriosa de técnicas distintas para diferentes partes do código pode evitar tanto *race conditions* como inconsistências mais subtis. A ausência de **0,0** mostra que não houve partilha ou colisão de identificadores, enquanto a consistência dos resultados implica que o *lock* foi eficaz a evitar acessos simultâneos ao vetor *log[].*

A estrutura do teste, apesar de simples, aproxima-se de padrões reais em sistemas de identificação, geração de tarefas ou registo de eventos concorrentes. A combinação de atomicidade com bloqueios seletivos emerge como uma prática sólida na construção de componentes *thread-safe* com boa escalabilidade.

5.3.3 Cenário com ThreadLocal e synchronized – SessionLogger

O teste *SessionLogger* foi desenvolvido para validar uma construção concorrente que isola o estado de cada *thread* através de *ThreadLocal*, enquanto sincroniza o acesso a um recurso partilhado (*log[]*) com *synchronized*. Esta abordagem simula padrões comuns em aplicações web, onde múltiplas sessões devem manter informação isolada, mas interagir pontualmente com recursos globais como ficheiros de log ou caches partilhadas.

Durante a execução com *JCStress*, o teste produziu resultados completamente determinísticos. Todas as execuções resultaram na atribuição correta de valores distintos às *threads* (**0,1** ou **1,0**), com o conteúdo de *log[]* intacto e livre de colisões. Como esperado, não foi gerada tabela de resultados múltiplos, já que não ocorreram variações estatísticas relevantes entre *forks*.

5.3.3.1 Análise Crítica

O comportamento observado no teste *SessionLogger* confirma que a combinação entre *ThreadLocal* e sincronização explícita oferece uma solução eficaz e segura para cenários em que se pretende isolar contexto de execução sem comprometer a integridade de estruturas partilhadas.

A utilização de *ThreadLocal* permite que cada *thread* mantenha o seu próprio estado, neste caso uma variável *session* inicializada com o valor **0**. Este mecanismo é amplamente utilizado em servidores de aplicações (como *Tomcat* ou *Spring*) para associar dados à execução de uma *thread* específica, sem risco de interferência entre utilizadores ou processos paralelos. Ao recorrer a este padrão, evita-se a necessidade de sincronização em operações de leitura e escrita sobre variáveis específicas da *thread*, o que contribui para melhor desempenho e menor contenção.

Por sua vez, o uso de *synchronized* no acesso ao vetor *log[]* assegura que apenas uma *thread* por vez modifica a estrutura partilhada. Embora o bloco sincronizado envolva apenas a operação de escrita, ele estabelece uma garantia clara de exclusão mútua e *de happens-before*, assegurando que os efeitos visíveis para uma *thread* sejam corretamente propagados para as demais.

A combinação destas duas técnicas resulta num sistema híbrido que alia isolamento lógico com controlo rigoroso de partilhas, sendo particularmente eficaz para sistemas que misturam lógica concorrente independente com pontos de integração comum (ex: autenticação, registo, auditoria). Esta construção está em linha com padrões arquitetónicos modernos como *request-scoped beans* e contextos de sessão *thread-safe*.

O teste demonstrou que não há ocorrência de *race conditions*, falhas de visibilidade ou inconsistências no vetor de registo, o que valida a correção da abordagem. Adicionalmente, a construção é simples de implementar e compreensível, oferecendo um bom compromisso entre robustez e simplicidade — duas qualidades desejáveis em ambientes empresariais críticos.

5.3.3.2 Observações adicionais

O padrão testado é particularmente adequado para sistemas com elevada concorrência onde cada unidade lógica de execução (ex: requisição de pedidos HTTP, tarefa de *background*) deve operar de forma isolada. O uso de *ThreadLocal* reduz a necessidade de sincronização em grande parte do código, concentrando os pontos críticos em operações partilhadas bem definidas.

Uma limitação relevante prende-se com o uso indevido ou não controlado de variáveis *ThreadLocal*, que podem causar *memory leaks* se não forem removidas manualmente em ambientes com *thread pools*. Contudo, este risco não afeta diretamente o teste em questão, já que as *threads* são geridas pela própria *framework*.

Em suma, os resultados obtidos confirmam que este padrão é eficaz, seguro e apropriado para muitos cenários de produção, sendo validado tanto pela teoria como pela prática.

5.3.4 Inconsistências de Memória sem Sincronização – *CacheCoherenceTest*

O teste *CacheCoherenceTest* foi concebido com o intuito de observar os efeitos da ausência total de sincronização entre *threads* no contexto de variáveis partilhadas, explorando assim fenómenos relacionados com coerência de cache e visibilidade de memória. Neste cenário, duas *threads* acedem e escrevem livremente em variáveis partilhadas (**a, b, r1, r2**) sem qualquer utilização de mecanismos de sincronização (como *volatile*, *synchronized* ou *locks*), permitindo expor os limites da *JMM* em contextos altamente concorrentes.

Durante a execução com *JCStress*, os resultados obtidos revelaram uma diversidade de comportamentos possíveis, incluindo combinações que, à primeira vista, poderiam parecer paradoxais (tais como r1=0, r2=0). Este resultado específico corresponde a um cenário onde ambas as *threads* leem os valores iniciais antes de qualquer escrita se tornar visível — um sintoma clássico de falha na visibilidade, decorrente da ausência de garantias *happens-before*. Foram registadas múltiplas combinações válidas segundo a *JMM*, o que demonstra a profundidade do fenómeno de reordenação e atraso de propagação entre caches de *CPU*.

A ausência de sincronização permitiu ao compilador e ao processador aplicar otimizações agressivas, como reordenação de instruções e *caching local* de variáveis, resultando em leituras desatualizadas e comportamentos não intuitivos. Este teste representa uma construção mínima, mas eficaz, para ilustrar como interações aparentemente simples entre *threads* podem resultar em resultados inesperados quando não existem barreiras de memória explícitas.

Os resultados foram agrupados por padrão de execução observado. Foram escolhidas 5 iterações aleatórias para exemplo e o no fim o valor final das iterações. Os valores registados estão organizados na Tabela 8.

Tabela 8 – Resultados dos Testes Inconsistências de Memória sem Sincronização – CacheCoherenceTest

Fork	Resultado	Amostras	Frequência %	Descrição
1	0,0	3633859	8,87	Interesting
	0,1	35642813	86,96	Acceptable
	1,0	1707480	4,17	Acceptable
	1,1	4539	0,01	Acceptable
2	0,0	438242	1,19	Interesting
	0,1	926305	2,51	Acceptable
	1,0	35607703	96,3	Acceptable
	1,1	2361	0,01	Acceptable
3	0,0	27467269	59,23	Interesting
	0,1	12536665	27,03	Acceptable
	1,0	6338356	13,67	Acceptable
	1,1	32641	0,07	Acceptable
4	0,0	5106217	8,76	Interesting
	0,1	495591089	85,05	Acceptable
	1,0	3602994	6,18	Acceptable
	1,1	5492	0,01	Acceptable
5	0,0	30850513	45,39	Interesting
	0,1	25128912	36,98	Acceptable
	1,0	11954529	17,59	Acceptable
	1,1	26897	0,04	Acceptable
Total	0,0	617203531	20,49	Interesting
	0,1	1305900338	43,36	Acceptable
	1,0	1086727578	36,08	Acceptable
	1,1	1911541	0,04	Acceptable

Entre os resultados, destaca-se a ocorrência não negligenciável do par (**0,0**), que representa o caso em que ambas as *threads* leram os valores iniciais antes que qualquer escrita se tornasse visível — um fenómeno clássico de falha de visibilidade. Este comportamento foi classificado como *Interesting* pela *framework JCStress*, por representar uma violação das expectativas intuitivas de muitos programadores.

Adicionalmente, o par (**1,1**), embora raríssimo, também ocorreu em todos os *forks*, indicando que ambas as *threads* puderam observar as escritas da outra — sem que, no entanto, isso estivesse garantido por qualquer sincronização explícita. A sua presença residual demonstra que resultados consistentes podem surgir mesmo em contextos inseguros, mas de forma não fiável.

Os pares (**0,1**) e (**1,0**) foram os mais prevalentes, sendo classificados como *Acceptable* por *JCStress*. Estes correspondem a cenários em que apenas uma das escritas se tornou visível a tempo da leitura pela outra thread, ilustrando o comportamento assimétrico típico da ausência de sincronização.

5.3.4.1 Análise Crítica

O teste *CacheCoherenceTest* revelou uma ampla dispersão estatística nos resultados, fruto direto da ausência total de sincronização entre as *threads*. Esta construção foi concebida para não estabelecer qualquer relação de *happens-before*, permitindo que cada *thread* aceda e modifique variáveis partilhadas sem coordenação, de acordo com os comportamentos permitidos pelo JMM. Os dados observados, refletidos na tabela de resultados, confirmam que tal abordagem leva a uma execução altamente não determinística e sujeita a variações subtis entre *forks*.

O resultado mais frequente, no agregado de todas as execuções, foi (**0,1**), com 1.305.900.338 amostras, representando 43,36% do total. Este cenário indica que a *thread* 0 não viu a escrita da *thread* 1, mas a *thread* 1 leu corretamente o valor atualizado de *thread* 0 — evidenciando assimetria de visibilidade. Já o caso inverso, (**1,0**), foi também bastante comum, com 36,08% das ocorrências. A elevada presença destes casos assimétricos (juntos, cerca de 79,44%) confirma que as *threads* operam com perceções distintas do estado global, uma situação típica de ausência de coerência de cache.

O caso (0,0), onde nenhuma *thread* vê a atualização da outra, representa 20,49% dos casos totais, com destaque para *forks* como o n.º 3, onde esse valor chegou a 59,23% — uma clara manifestação da leitura de valores obsoletos, provavelmente retidos em caches locais ou registos. Este resultado, embora tecnicamente válido na *JMM*, seria inaceitável em aplicações onde se espera sincronização implícita entre *threads*. A ocorrência de (**0,0**) em cinco *forks* com variações entre 1,19% e 59,23% demonstra como a ordem de execução e as particularidades da plataforma influenciam fortemente o resultado final.

O cenário (**1,1**), em que ambas as *threads* detetam corretamente as atualizações uma da outra, é o mais raro — apenas 0,04% no total, com menos de 0,1% em todos os *forks*. Embora esse resultado represente o comportamento "ideal", a sua raridade mostra que confiar na propagação espontânea de visibilidade entre *threads* é perigoso. Tal comportamento só é garantido com sincronização explícita.

Estes dados revelam, de forma concreta, como a falta de coordenação pode levar a execuções erráticas, mesmo em sistemas aparentemente simples. A distribuição de percentagens variáveis por *fork* — como (**0,1**) a oscilar entre 2,51% e 86,96%, ou (**1,0**) entre 4,17% e 96,3% — reforça a tese de que, sem barreiras de sincronização, o comportamento do programa depende fortemente da ordem de agendamento e do contexto de execução.

Este teste é uma demonstração prática da falácia comum em programação concorrente: assumir que, se um programa "funciona" em testes locais, então está correto. A ausência de erros visíveis ou falhas de execução não garante integridade quando não há garantias formais de sincronização. O *CacheCoherenceTest* demonstra de forma inequívoca que o comportamento normal pode esconder falhas graves de visibilidade e que os testes devem ser desenhados para revelar precisamente estas condições limite.

Em suma, o comportamento registado neste teste sublinha a importância crítica da sincronização explícita em programação concorrente. Sistemas que operam sem ela estão não apenas sujeitos a inconsistências de dados, mas também a resultados aleatórios e irreproduzíveis, que comprometem a fiabilidade, segurança e previsibilidade da execução em ambiente de produção.

5.3.4.2 Observações adicionais

O padrão testado neste cenário — partilha de estado sem sincronização — constitui um anti padrão clássico e perigoso. A sua recorrência em implementações ingénuas de concorrência representa um risco real para a integridade e previsibilidade dos sistemas.

Embora os pares (**0,1**) e (**1,0**) possam ser toleráveis ou mesmo esperados em certos contextos, a ocorrência de (**0,0**) deve ser considerada erro de conceção. A *framework JCStress*, ao marcar este comportamento como *Interesting*, fornece um alerta implícito: este resultado não é incorreto segundo a *JMM*, mas é problemático em termos de intenção de design.

Este teste mostra ainda o valor de uma ferramenta como *JCStress*, que permite revelar comportamentos não determinísticos através de execuções massivas, que dificilmente seriam reproduzíveis de forma manual. O conhecimento destes padrões é essencial para qualquer programador envolvido no desenvolvimento de sistemas concorrentes seguros, sobretudo em ambientes críticos onde consistência e previsibilidade são obrigatórias.

5.3.5 Visibilidade com *synchronized* – *CacheCoherenceLocked*

O teste *CacheCoherenceLocked* foi concebido como uma resposta direta às anomalias observadas no teste anterior, *CacheCoherenceTest*, que evidenciou falhas graves de visibilidade entre *threads*. Nesta nova abordagem, foi introduzido o uso explícito de blocos *synchronized* para aceder às variáveis partilhadas, de modo a garantir visibilidade e coerência de memória. Ao envolver os acessos às variáveis a e b em métodos sincronizados, o teste passa a beneficiar das garantias do modelo de memória Java associadas à construção *synchronized*, incluindo o estabelecimento automático de relações *happens-before*.

Durante a execução com a ferramenta *JCStress*, o comportamento do teste foi notoriamente distinto do seu antecessor. Todos os resultados recolhidos apresentaram-se completamente determinísticos e corretos do ponto de vista semântico. Em particular, não foi observado qualquer cenário em que ambas as *threads* tenham lido o valor 0 (resultado (**0,0**)), o qual, no teste anterior, correspondia a um claro sintoma de falta de propagação entre escritas e leituras concorrentes.

Todos os pares de resultados registados foram limitados às combinações (**0,1**),(**1,0**) e (**1,1**), indicando que, em cada execução, pelo menos uma das *threads* observou corretamente o valor definido pela outra. Este padrão demonstra que as garantias de visibilidade estabelecidas pelos blocos sincronizados foram suficientes para eliminar os efeitos adversos de caches incoerentes ou reordenações no nível da *CPU* e da *JVM*.

5.3.5.1 Análise Crítica

Os resultados observados no teste *CacheCoherenceLocked* demonstram inequivocamente a eficácia dos blocos *synchronized* para mitigar os riscos de inconsistência causados pela ausência de garantias de visibilidade em ambientes concorrentes. Ao forçar a sincronização de acesso às variáveis partilhadas a e b, o teste obtém os benefícios duplos de exclusão mútua e sincronização de memória. A sincronização assegura que alterações realizadas por uma *thread* tornam-se visíveis às restantes *threads*, desde que estas entrem em blocos sincronizados sobre o mesmo objeto.

Esta abordagem evidencia a importância de compreender que as *race conditions* em Java não se limitam a interferência entre escritas concorrentes, mas incluem também problemas na visibilidade da memória, onde uma *thread* pode não observar os efeitos de outra, mesmo na ausência de colisões diretas. A substituição do acesso direto por métodos sincronizados resolve precisamente este problema, fornecendo um modelo comportamental seguro sem exigir modificações estruturais complexas ao código.

O facto de todos os resultados se restringirem a combinações sem ambiguidade ou inconsistência, isto é, sem (**0,0**) confirma que os mecanismos oferecidos por *synchronized* são suficientes, neste contexto, para garantir a coerência desejada. Este comportamento é especialmente relevante em sistemas multiprocessadores onde as caches locais das *CPUs* podem manter visões divergentes da memória partilhada, tornando os blocos sincronizados numa ferramenta essencial para evitar estas disfunções.

Ainda que o uso de *synchronized* possa introduzir algum *overhead* de desempenho, a sua utilização pontual, como neste teste, permite controlar com precisão os momentos em que as garantias de memória são exigidas, sem comprometer desnecessariamente a eficiência global da aplicação. Por este motivo, esta técnica continua a ser amplamente empregada em implementações robustas de bibliotecas concorrentes, como os presentes no pacote *java.util.concurrent*.

Assim, o teste *CacheCoherenceLocked* valida um padrão clássico, eficaz e de fácil compreensão para garantir integridade em cenários partilhados, sobretudo quando se pretende assegurar consistência de leitura/escrita entre múltiplas *threads* num ambiente de execução sujeito a otimizações agressivas por parte da *JVM* ou do hardware subjacente.

5.4 Considerações Finais da Implementação

Esta secção apresenta uma análise crítica dos testes de concorrência realizados, com o objetivo de identificar padrões de comportamento, validar empiricamente as garantias oferecidas por diferentes construções e refletir sobre a sua aplicabilidade prática. A abordagem adotada procura ir além da exposição dos resultados individuais, articulando os dados obtidos com os princípios do modelo de memória do Java e com as boas práticas reconhecidas na literatura. São discutidas, de forma sistematizada, as implicações dos resultados, as contribuições observadas, bem como as limitações tanto da ferramenta utilizada como das técnicas avaliadas.

Esta análise visa, assim, consolidar as evidências empíricas recolhidas e apoiar a formulação de conclusões fundamentadas sobre a robustez e adequação das soluções concorrentes testadas.

5.4.1 Comparação Geral dos Resultados

A análise comparativa dos testes realizados permitiu identificar, com elevado grau de detalhe, os comportamentos emergentes das diferentes construções concorrentes testadas, nomeadamente no que respeita à sua robustez, previsibilidade e conformidade com o *JMM*. Através da utilização da ferramenta *JCStress*, foi possível simular condições de concorrência extremas e observar os efeitos práticos das abordagens utilizadas, bem como a respetiva adequação ao paradigma *multithreaded*.

As construções baseadas em sincronização explícita — como o uso de *synchronized* e *ReentrantLock* — apresentaram um comportamento amplamente previsível, com classificações predominantemente *ACCEPTABLE*, o que indica ausência de violações ao modelo de memória e conformidade com as expectativas funcionais. Já as abordagens que recorreram a mecanismos mais finos de isolamento, como *ThreadLocal*, revelaram uma elevada capacidade de encapsulamento do estado, impedindo interferências cruzadas entre *threads*.

Por outro lado, as abordagens sem qualquer forma de sincronização evidenciaram vulnerabilidades críticas, com resultados classificados como *FORBIDDEN* ou *ACCEPTABLE_INTERESTING*, refletindo a presença de *race conditions*, leituras inconsistentes e falhas na visibilidade dos dados partilhados. Tais resultados confirmam, na prática, os riscos teóricos associados à ausência de controlo de concorrência, tal como descrito na literatura especializada.

A Tabela 9 resume os principais comportamentos observados:

Tabela 9 - Comportamentos dos Testes Realizados

Teste	**Comportamento Esperado**	**Comportamento Observado**	**Classificação Dominante**
5.2.1	Inconsistência devido à partilha sem controlo	*Race conditions* frequentes e não determinismo	ACCEPTABLE / ACCEPTABLE_INTERESTING
5.2.2	Sincronização eficaz e atomicidade	Sincronização eficaz e atomicidade	ACCEPTABLE
5.2.3	Isolamento total do contexto	Separação clara entre execuções	ACCEPTABLE
5.2.4	Visibilidade inconsistente esperada	Ocorreram leituras obsoletas ou incorretas	ACCEPTABLE / ACCEPTABLE_INTERESTING
5.2.5	Garantia de exclusão mútua e ordem	Execução ordenada e sem desvios	ACCEPTABLE

Entre os padrões detetados, destaca-se a fragilidade da abordagem baseada em incremento não atómico com variáveis partilhadas. Este teste demonstrou ser o mais suscetível à ocorrência de *race conditions*, revelando-se impraticável para qualquer cenário que exija integridade de

dados. Em contrapartida, os testes com *AtomicInteger* e *ReentrantLock* revelaram excelente comportamento, com garantia de atomicidade e proteção contra interferência *inter-thread*, embora com maior complexidade na implementação.

A utilização de *ThreadLocal* com *synchronized* demonstrou não só robustez, como também uma clara separação entre os contextos de execução de cada *thread*, validando o seu uso em ambientes que requerem simulação de sessões independentes, como aplicações web ou de serviços concorrentes.

É ainda relevante notar que o uso de *synchronized* puro mostrou-se eficaz na maioria dos casos, sobretudo ao nível da visibilidade e ordenação de acessos. No entanto, como será discutido nas próximas secções, esta técnica pode introduzir problemas de escalabilidade em ambientes com elevada contenção de recursos.

Em síntese, os testes confirmam que a escolha adequada de mecanismos de concorrência não pode basear-se apenas na correção funcional esperada, devendo considerar fatores como a complexidade de implementação, o custo computacional e o contexto de aplicação. A análise transversal dos resultados reforça a importância de uma abordagem informada e criteriosa na seleção das técnicas a utilizar em sistemas *multithreaded*.

5.4.2 Contribuições Práticas dos Testes

Os testes realizados com recurso ao *JCStress* permitiram obter evidências empíricas que sustentam diversas conclusões práticas sobre a segurança, eficácia e aplicabilidade das construções concorrentes avaliadas. Ao submeter cada técnica a cenários de concorrência controlada, mas intensiva, foi possível avaliar o seu comportamento sob condições extremas de *interleaving*, proporcionando uma perspetiva realista sobre a sua utilização em ambientes de produção.

Uma das principais contribuições prende-se com a validação da robustez do *AtomicInteger* em combinação com *ReentrantLock*. Esta abordagem demonstrou elevada fiabilidade na gestão de acesso concorrente a recursos partilhados, proporcionando garantias de atomicidade e consistência mesmo em situações de elevada simultaneidade. O bloqueio explícito do *ReentrantLock*, associado à natureza não bloqueante do *AtomicInteger*, permitiu mitigar efetivamente *race conditions*, sem comprometer a correção funcional dos resultados. A capacidade de *ReentrantLock* permitir políticas de espera configuráveis (como *fair locking*) ou tentativas temporizadas revela-se particularmente útil em sistemas de maior complexidade ou com elevado grau de contenção. Este tipo de solução é, por exemplo, aplicável em arquiteturas de filas concorrentes, gestores de tarefas ou sistemas onde o programador necessita de maior controlo sobre a política de bloqueio (Goetz et al., 2006). Esta combinação revela-se, portanto, adequada para cenários que exigem elevada segurança e controlo fino sobre o acesso a dados partilhados.

A abordagem baseada em *ThreadLocal* com *synchronized* mostrou-se particularmente eficaz na simulação de contextos isolados, como sessões independentes em aplicações com múltiplos utilizadores. A separação de estado entre *threads*, proporcionada pelo *ThreadLocal*, evitou por completo interferências concorrentes, garantindo a integridade de cada instância de execução. Este padrão mostra-se vantajoso em sistemas onde cada *thread* ou pedido necessita de manter um estado próprio, como servidores *HTTP*, *microservices REST* ou pipelines de processamento em paralelo. O *synchronized*, ao ser aplicado apenas a blocos críticos internos, contribui para a manutenção da consistência sem comprometer o isolamento funcional (Lea, 1996). Esta técnica é, por isso, especialmente recomendada para sistemas que requerem isolamento lógico entre contextos de execução, como servidores de aplicações ou ambientes paralelos com dados sensíveis.

Por outro lado, os testes sem qualquer tipo de sincronização, nomeadamente o acesso direto a variáveis partilhadas ou leituras de cache (*CacheCoherenceTest*), revelaram-se problemáticos. A ausência de mecanismos de controlo resultou em inconsistências de memória, leituras obsoletas e resultados inválidos, confirmando a vulnerabilidade desta abordagem em qualquer ambiente *multithreaded*. Estas falhas são coerentes com o que é descrito na literatura, nomeadamente os riscos conhecidos do modelo de memória da *JVM*, onde a falta de visibilidade pode conduzir a comportamentos inesperados mesmo sem interferência externa (Hovemeyer & Pugh, 2004), reforçando a inadmissibilidade de tal prática em sistemas que exijam fiabilidade.

Os resultados obtidos constituem também uma validação prática dos conceitos abordados no estado da arte. A correlação entre as garantias teóricas atribuídas a cada técnica e os comportamentos verificados nos testes reforça a importância de aplicar princípios bem estabelecidos da programação concorrente, como exclusão mútua, visibilidade garantida e encapsulamento de estado. Além disso, a ligação entre a natureza da técnica e a sua aplicabilidade prática torna evidente que a escolha da abordagem deve ter em conta não só as garantias formais, mas também os requisitos específicos do sistema, como carga esperada, necessidade de isolamento ou facilidade de manutenção. A confirmação experimental de situações como *race conditions* ou falhas de visibilidade reforça também a pertinência do uso de ferramentas formais como o *JCStress* na fase de desenvolvimento.

Em termos de aplicabilidade, os testes contribuíram para a identificação de padrões seguros e replicáveis na construção de soluções concorrentes. Foram também úteis na demonstração de que a complexidade da técnica utilizada nem sempre se traduz em maior segurança, sendo por vezes preferível adotar construções mais simples, como *synchronized*, desde que bem aplicadas e adequadas ao contexto. A análise reforça ainda a necessidade de balancear simplicidade com robustez, avaliando o impacto de cada técnica no desempenho, na manutenção do código e na previsibilidade do comportamento em produção.

5.4.3 Limitações da Abordagem aos Testes

Apesar dos contributos obtidos com os testes realizados, é importante reconhecer um conjunto de limitações que condicionam a abrangência e a generalização dos resultados. Estas limitações decorrem tanto das características intrínsecas da ferramenta *JCStress* como das opções metodológicas e técnicas adotadas ao longo da implementação.

Em primeiro lugar, destaca-se a restrição inerente ao modelo de testes do *JCStress*, que se centra exclusivamente em cenários com dois atores (*threads*), operando sobre um conjunto limitado de variáveis partilhadas. Esta configuração, embora eficaz na deteção de falhas subtis de concorrência, não representa com fidelidade o comportamento de sistemas reais, onde múltiplas *threads* interagem simultaneamente com diversos recursos. A impossibilidade de simular ambientes com concorrência intensiva, escalonamento complexo ou interdependência entre múltiplos contextos limita o alcance dos testes efetuados.

Adicionalmente, o *JCStress* não permite testar cenários com latência variável, o que restringe a capacidade de análise a ambientes puramente computacionais e determinísticos. Muitos sistemas concorrentes reais dependem de interações assíncronas com serviços externos, de sincronização temporal entre tarefas distribuídas ou de tempos de resposta não determinísticos, características que não são modeláveis com esta ferramenta. Assim, embora os testes tenham sido eficazes na validação do comportamento interno das construções, não contemplaram fatores externos que frequentemente influenciam o comportamento concorrente.

No que respeita às construções testadas, verificam-se também algumas limitações específicas:

- A utilização de *synchronized*, apesar de eficaz na maioria dos cenários, pode restringit o desempenho em sistemas com elevada contenção de recursos, uma vez que bloqueia o acesso a secções críticas de forma exclusiva, impedindo a paralelização eficiente de tarefas.
- O *ReentrantLock*, por sua vez, embora mais flexível, introduz uma complexidade adicional na gestão de bloqueios, exigindo um maior cuidado para evitar situações de *deadlock* ou de esquecimento da libertação do *lock*, o que compromete a segurança da aplicação.

Outro aspeto relevante é a dependência do comportamento da JVM e do sistema operativo. A execução dos testes está sujeita às decisões do escalonador de *threads*, à implementação do modelo de memória da *JVM* específica utilizada e até às otimizações realizadas pelo compilador *Just-In-Time (JIT)*. Assim, os comportamentos observados poderão variar entre ambientes distintos, especialmente quando executados em arquiteturas com características diferentes de paralelismo, *caching* ou de *CPU*. Esta dependência compromete, em parte, a reprodutibilidade absoluta dos testes.

Por fim, importa referir que a definição manual dos testes no *JCStress* exige um conhecimento aprofundado dos padrões de concorrência a avaliar, o que pode limitar a capacidade de cobrir casos menos óbvios ou emergentes. A ferramenta não inclui funcionalidades automáticas de

geração de cenários ou de deteção preditiva, o que restringe o seu uso a situações bem delimitadas e previamente conhecidas.

Em suma, embora os testes realizados tenham contribuído significativamente para a compreensão das propriedades concorrentes das técnicas avaliadas, os seus resultados devem ser interpretados à luz destas limitações. A generalização das conclusões deverá, portanto, considerar a natureza controlada do ambiente de testes e a especificidade dos cenários implementados.

5.4.4 Considerações Finais

A análise final dos testes realizados permitiu validar, na prática, as garantias oferecidas pelas principais técnicas de sincronização em Java. Abordagens como *synchronized*, *ReentrantLock* e *ThreadLocal* demonstraram ser eficazes na prevenção de *race conditions* e na manutenção da consistência dos dados partilhados. Por oposição, as técnicas que ignoram mecanismos de controlo revelaram comportamentos não determinísticos e violações críticas do modelo de memória.

Mais do que confirmar o comportamento esperado das construções analisadas, os testes permitiram compreender as implicações práticas da sua adoção. A eficácia de cada técnica está fortemente dependente do contexto em que é aplicada, da carga concorrente do sistema, e das garantias de isolamento ou sincronização requeridas. Por exemplo, o uso de *ThreadLocal* revelou-se particularmente robusto em simulações de sessões independentes, sugerindo a sua adequação para servidores web ou *microservices* com múltiplos utilizadores. Já *ReentrantLock*, pela sua flexibilidade, mostrou-se mais apropriado em sistemas com elevada contenção e necessidade de controlo refinado sobre o acesso a recursos críticos.

Estes resultados confirmam os princípios discutidos no estado da arte, reforçando a importância da aplicação criteriosa de boas práticas em ambientes *multithreaded*. Os testes realizados com *JCStress*, apesar das suas limitações, ofereceram uma base empírica sólida para comparar abordagens e extrair recomendações práticas. A utilização desta ferramenta demonstrou ser útil não apenas para detetar comportamentos inválidos, mas também para ilustrar de forma tangível os riscos associados a más práticas de sincronização.

Conclui-se que a robustez de um sistema concorrente depende não apenas da técnica usada, mas da sua adequação ao contexto específico de execução. A escolha da construção apropriada deverá considerar fatores como o tipo de dados partilhados, o grau de concorrência esperado e o impacto no desempenho. O valor destes testes reside precisamente na capacidade de fornecer orientações para essa escolha, com base em evidência observável. A experiência obtida ao longo deste estudo contribui para uma compreensão mais aprofundada das construções concorrentes e do seu impacto na fiabilidade e previsibilidade do software.

6 Conclusões

Este capítulo encerra a dissertação com uma síntese dos principais contributos alcançados, à luz dos objetivos inicialmente definidos. Com base na análise empírica realizada e na fundamentação teórica previamente estabelecida, procede-se à avaliação do grau de cumprimento das perguntas de investigação, identificando os resultados mais relevantes e o seu impacto na compreensão dos problemas de concorrência em Java. São ainda discutidas, de forma crítica, as limitações do estudo e sugeridas possíveis linhas de continuidade para trabalhos futuros que possam aprofundar ou expandir os resultados aqui obtidos.

6.1 Avaliação dos objetivos e contributos do trabalho

Esta dissertação partiu da premissa de que a programação concorrente, sendo fundamental em sistemas modernos e paralelos, permanece uma das áreas mais propensas a erros subtis e de difícil reprodução. Neste contexto, o foco do trabalho centrou-se na análise comparativa de ferramentas e técnicas de deteção de erros concorrentes, com ênfase na avaliação empírica e na aplicabilidade prática. Procurou-se não apenas compreender o funcionamento interno de cada ferramenta, mas também perceber em que medida estas contribuem para a deteção antecipada de problemas como condições de corrida, falhas de visibilidade ou interferências não determinísticas. Ao longo do estudo, foram definidos critérios objetivos de comparação, executados testes práticos com recurso ao *JCStress* e avaliadas técnicas como *synchronized*, *ReentrantLock* ou *ThreadLocal*, com o intuito de oferecer uma visão crítica e fundamentada. As conclusões obtidas revelam não só as vantagens e limitações de cada abordagem, como também permitem identificar padrões de utilização seguros, oferecendo orientações úteis para contextos de desenvolvimento real. Deste modo, a investigação realizada contribui para um melhor entendimento dos desafios da concorrência em Java e para a adoção de boas práticas sustentadas em evidência empírica.

Para orientar o desenvolvimento da investigação, foram definidas quatro perguntas de investigação e um conjunto de objetivos específicos, dos quais se destaca a revisão crítica do estado da arte, a avaliação empírica de abordagens concorrentes e a identificação das suas principais limitações.

As perguntas de investigação foram abordadas da seguinte forma:

1. **Quais são as principais limitações e desafios das ferramentas existentes para a deteção de problemas de concorrência em Java?**
   Esta questão foi amplamente discutida na revisão do estado da arte (Capítulo 3) e complementada pelas observações empíricas dos testes realizados. Foram identificadas limitações relacionadas com precisão, escalabilidade, falsos positivos e cobertura limitada de cenários reais.
2. **Como varia a precisão, a cobertura e o desempenho das ferramentas em diferentes cenários de execução, e qual o impacto dos falsos positivos/negativos?**
   Esta questão foi abordada de forma empírica nos testes documentados no Capítulo 5.
   Conforme evidenciado na Tabela 9, observam-se diferenças claras na fiabilidade das construções avaliadas:
   - No teste 5.2.1 ,verificou-se uma alta frequência de *race condition*, com resultados classificados como INTERESTING, refletindo um comportamento não determinístico e inseguro.
   - Em contraste, os testes 5.2.2 e 5.2.3 apresentaram total previsibilidade, com resultados classificados como ACCEPTABLE.

   As técnicas com sincronização explícita apresentaram resultados consistentes e classificações aceitáveis, enquanto as que dispensaram mecanismos de controlo revelaram comportamentos erráticos. Estes resultados demonstram que a técnica utilizada tem um impacto direto na prevenção de falhas concorrentes e que construções sincronizadas adequadas contribuem significativamente para a robustez do sistema.
   Esta análise dá assim uma resposta direta à segunda pergunta de investigação, ilustrando como a escolha da técnica condiciona a precisão dos resultados e a capacidade de deteção de falhas de concorrência.
3. **Como podem as ferramentas ser otimizadas para melhorar a eficiência e precisão, reduzindo falsos positivos e negativos?**
   Embora esta questão tenha sido inicialmente considerada, o trabalho optou por não abordar diretamente propostas de otimização, tendo privilegiado uma análise comparativa aprofundada entre técnicas existentes. Assim, este objetivo foi excluído do âmbito final da dissertação, tendo sido substituído por uma abordagem empírica mais restrita, centrada na validação e interpretação de resultados.
4. **Quais são os impactos das melhorias na deteção de concorrência no ciclo de desenvolvimento de software e na fiabilidade de sistemas críticos?**
   Os testes e a análise desenvolvida permitiram concluir que técnicas bem aplicadas de sincronização contribuem significativamente para a previsibilidade e segurança de sistemas concorrentes. A fiabilidade do código *multithreaded* depende, em grande

parte, da escolha adequada de construções e da sua validação sistemática, como demonstrado pela utilização de *JCStress*.

Em termos de objetivos específicos, foram plenamente alcançados:

- A revisão do estado da arte permitiu identificar com rigor as ferramentas e técnicas mais relevantes, detalhando os seus mecanismos internos e limitações;
- A avaliação comparativa foi realizada com base em testes sistemáticos, conduzidos em condições controladas, e sustentada por métricas de observação direta;
- As limitações das técnicas analisadas foram discutidas com base em evidências empíricas e comparadas com o que é reportado na literatura;
- A validação de boas práticas foi confirmada experimentalmente, reforçando a adequação de certas abordagens (como *synchronized*, *ReentrantLock* ou *ThreadLocal*) à construção de sistemas concorrentes robustos.

Por fim, importa destacar que este trabalho contribui para o domínio da engenharia de software com uma análise prática e acessível de construções de concorrência, muitas vezes discutidas apenas em termos teóricos. Através da sua aplicação controlada e observação sistemática, foi possível extrair conhecimento aplicável à conceção de sistemas mais fiáveis, promovendo a adoção fundamentada de boas práticas no desenvolvimento de aplicações Java *multithreaded.*

6.2 Reflexão sobre o alcance e limitações do estudo

O desenvolvimento deste trabalho permitiu uma exploração aprofundada das construções de programação concorrente em Java, bem como das ferramentas de suporte à sua validação. A opção por uma abordagem experimental, focada em testes realizados com o *JCStress*, possibilitou observar diretamente os comportamentos emergentes das técnicas analisadas, contribuindo para uma avaliação empiricamente fundamentada das suas garantias e limitações.

Contudo, importa reconhecer que o estudo decorreu dentro de um enquadramento metodológico e técnico que impôs restrições relevantes ao seu alcance. A primeira limitação prende-se com a natureza sintética dos cenários testados. Apesar de representativos de padrões recorrentes em sistemas concorrentes, os testes foram concebidos de forma isolada e limitada, sem contemplar arquiteturas complexas ou interações em larga escala. Como tal, os resultados observados, embora válidos no contexto em que foram gerados, não são diretamente generalizáveis para sistemas distribuídos, ambientes com latência variável ou contextos sujeitos a falhas externas.

Adicionalmente, a ferramenta *JCStress*, apesar da sua precisão na deteção de falhas ao nível da memória e da ordem de execução, não permite simular situações com múltiplos atores, componentes externos ou carga variável, o que limita a exploração de cenários mais realistas. A sua utilização requer ainda conhecimento detalhado do *Java Memory Model* e da lógica de *interleaving*, o que restringe o seu uso a contextos técnicos especializados. Por conseguinte, o estudo incidiu sobretudo em padrões de concorrência de baixo nível, deixando de fora aspetos como escalabilidade algorítmica, coordenação de tarefas em múltiplos nós, ou integração com sistemas de produção.

Ao nível da avaliação qualitativa, a comparação entre técnicas baseou-se na observação dos comportamentos produzidos pelos testes, mas não envolveu medições formais de desempenho, como tempos de execução, consumo de *CPU* ou métricas de latência. Estas dimensões, embora relevantes em ambientes de alta performance, ultrapassaram o escopo do estudo e foram assumidas como fora do seu foco principal.

Do ponto de vista temático, foi ainda abandonado o objetivo inicial relacionado com propostas de otimização de ferramentas, devido à complexidade associada à sua implementação prática no tempo disponível. A decisão de restringir o foco à análise comparativa permitiu, no entanto, aprofundar a dimensão empírica do trabalho, garantindo uma abordagem mais realista e operacional das técnicas existentes.

Em síntese, o trabalho desenvolvido proporcionou uma base sólida de observação e reflexão sobre mecanismos de concorrência em Java, mas não pretendeu ser exaustivo nem definitivo. As conclusões devem ser lidas à luz do seu contexto experimental e das opções metodológicas adotadas, sendo recomendável a sua extensão em estudos futuros que explorem dimensões adicionais da programação concorrente em ambientes reais e distribuídos.

6.3 Trabalhos futuros

Apesar dos avanços significativos observados nas ferramentas e *frameworks* analisadas, persistem limitações que apontam para várias oportunidades de investigação e desenvolvimento futuro na área da programação concorrente em Java. Esta secção propõe linhas de trabalho que poderão ser exploradas no sentido de aumentar a eficácia, escalabilidade e adaptabilidade das soluções existentes a contextos modernos e distribuídos.

6.3.1 Integração de Aprendizagem Automática

Uma das principais melhorias sugeridas é a integração de algoritmos de aprendizagem automática para a deteção preditiva de problemas de concorrência. Esta abordagem permitiria identificar padrões de comportamento que possam levar a *race conditions* ou *deadlocks* antes que ocorram, aumentando a precisão e reduzindo falsos positivos e negativos. Ferramentas como o *Seagence* seriam beneficiadas pela inclusão de técnicas baseadas em *machine learning*, que, ao analisar grandes volumes de *logs*, poderiam aprender comportamentos anómalos e sugerir intervenções proativas. Além disso, a utilização de *Deep Learning* ajudaria a reduzir falsos positivos e negativos, oferecendo diagnósticos mais confiáveis. Este avanço seria particularmente útil em sistemas de larga escala, como plataformas de *e-commerce* e redes de telecomunicações, onde a deteção antecipada de falhas é crítica para a manutenção da operação.

6.3.2 Otimização da Escalabilidade das Ferramentas

Outra área de melhoria foca-se na escalabilidade das ferramentas e *frameworks*. Foi observado que a *State Space Explosion* continua a ser um desafio em soluções como o JPF. Propostas como a utilização do *State Pruning Techniques e Advanced Heuristics* podem reduzir significativamente o impacto deste problema, melhorando a capacidade das ferramentas de lidar com sistemas complexos e de grande escala. Para além disso, estas técnicas podem ser complementadas por estratégias de paralelização, através do uso de *clusters* ou arquiteturas distribuídas para processar estados em paralelo, aumentando a eficiência da análise.

6.3.3 Melhoria da Usabilidade e Integração com *IDE's*

A simplificação da configuração e utilização das soluções também é uma prioridade. *Frameworks* como a *Fork-Join Framework* e ferramentas como o *ThreadSanitizer* poderiam beneficiar de interfaces de alto nível que tornem a adoção destas mais acessível a equipas de desenvolvimento menos especializadas em programação concorrente. Estas interfaces poderiam incluir documentação mais interativa e *user-friendly,* para além duma integração nativa com *IDEs*, como o *Eclipse* ou o *IntelliJ IDEA,* o que permitiria uma experiência mais fluida para os programadores, promovendo a utilização generalizada destas soluções.

6.3.4 Compatibilidade com Arquiteturas Modernas

A compatibilidade com ambientes tecnológicos contemporâneos, como plataformas de *microservices* e sistemas de orquestração de *containers* por exemplo, *Kubernetes*, deve ser ampliada para facilitar a integração com arquiteturas modernas. A integração de ferramentas e *frameworks* com estas plataformas poderia incluir módulos específicos para monitorização e análise de sistemas distribuídos, permitindo uma visão consolidada e em tempo real do desempenho e dos potenciais problemas de concorrência.

6.3.5 Criação de *Benchmarks* Representativos

A criação de *benchmarks* específicos é outra melhoria identificada. A ausência de cenários padronizados que representem desafios reais enfrentados por sistemas distribuídos limita a capacidade de avaliar e comparar ferramentas de forma rigorosa. Propõe-se, assim, o desenvolvimento de *benchmarks* que abranjam tanto cenários genéricos como casos de uso específicos, como transações financeiras ou gestão de recursos em redes de telecomunicações. Estes *benchmarks* devem incluir métricas detalhadas, como latência, taxa de falsos positivos, consumo de recursos e escalabilidade, permitindo análises mais completas e comparações mais justas entre as soluções.

6.3.6 Conclusões dos Trabalhos Futuros

A integração de algoritmos de aprendizagem automática, como redes neurais, pode melhorar significativamente ferramentas como o *Seagence* em ambientes industriais. Por exemplo, em sistemas de telecomunicações, algoritmos de *machine learning* podem prever congestionamentos de rede, evitando falhas críticas. A utilização de *State Pruning Techniques* no JPF, combinada com estratégias de paralelização em *clusters*, pode ser aplicada em sistemas financeiros para analisar rapidamente fluxos complexos de transações em mercados de alta frequência. Por sua vez, a criação de *benchmarks* específicos pode beneficiar a indústria automóvel, permitindo avaliar a robustez de sistemas concorrentes em cenários de condução autónoma. A integração com plataformas de orquestração de containers, como *Kubernetes*, é essencial para modernizar ferramentas de concorrência, facilitando sua adoção em sistemas baseados em *microservices*.

Estas propostas de melhoria têm como objetivo não apenas resolver as limitações observadas nas soluções atuais, mas também alinhar as ferramentas e *frameworks* às necessidades dos sistemas contemporâneos. Ao implementar estas melhorias, espera-se que a programação concorrente em Java se torne mais eficiente, escalável e adaptável a um ambiente tecnológico em constante evolução.

Referências

Ahmed, H., Biricik, S., Elbouchikhi, E., & Benbouzid, M. (2020). Adaptive filtering-based pseudo open-loop three-phase grid-synchronization technique. *Energies*, *13*(11). https://doi.org/10.3390/EN13112927

Al Mamun, A., Khanam, A., Grahn, H., & Feldt, R. (2010). *Comparing Four Static Analysis Tools for Java Concurrency Bugs*. https://www.researchgate.net/publication/48872915_Comparing_Four_Static_Analysis_Tools_for_Java_Concurrency_Bugs

Alippi, C., Fellow, I., Ntalampiras, S., & Roveri, M. (2016). Model-free fault detection and isolation in large-scale cyber-physical systems. *IEEE Transactions on Emerging Topics in Computational Intelligence*. https://doi.org/10.1109/TETCI.2016.2641452

Alvarez Cid-Fuentes, J., Szabo, C., & Falkner, K. (2020). Adaptive Performance Anomaly Detection in Distributed Systems Using Online SVMs. *IEEE Transactions on Dependable and Secure Computing*, *17*(5), 928–941. https://doi.org/10.1109/TDSC.2018.2821693

Andrianov, P., & Mutilin, V. (2024). *Software Model Checking for Memory Consistency Verification*. https://doi.org/388412712

Balaji Medewar, G. (2024). *Deadlock in Java with Examples*. https://www.scaler.com/topics/deadlock-in-java/

Beronic, D., Pufek, P., Mihaljevic, B., & Radovan, A. (2021). On Analyzing Virtual Threads - A Structured Concurrency Model for Scalable Applications on the JVM. *2021 44th International Convention on Information, Communication and Electronic Technology, MIPRO 2021 - Proceedings*, 1684–1689. https://doi.org/10.23919/MIPRO52101.2021.9596855

Bitla, S. (2023). *Concurrency Issues in Java: Your options to find and fix - Seagence*. https://seagence.com/java/concurrency-issues-in-java/

Boehm, B. W. (1984). Software Engineering Economics. *IEEE Transactions on Software Engineering*, *SE-10*(1), 4–21. https://doi.org/10.1109/TSE.1984.5010193

Bora, S., & Dikenelli, O. (2006). Applying feedback control in adaptive replication mechanisms in fault tolerant multi-agent organizations. *Proceedings - International Conference on Software Engineering*, 5–11. https://doi.org/10.1145/1138063.1138066

Carlstrom, B. D., Chung, J. W., Chafi, H., McDonald, A., Minh, C. C., Hammond, L., Kozyrakis, C., & Olukotun, K. (2006). Executing Java programs with transactional memory. *Science of Computer Programming*, *63*(2), 111–129. https://doi.org/10.1016/J.SCICO.2006.05.006

Charpentier, M. (2023). *Functional and Concurrent Programming: Core Concepts and Features*. Pearson Education.

Chen, Z., Lei, H., Yang, M., Liao, Y., & Qiao, L. (2021). A Finer-Grained Blocking Analysis for Parallel Real-Time Tasks with Spin-Locks. *2021 58th ACM/IEEE Design Automation Conference (DAC)*, *2021-December*, 1177–1182. https://doi.org/10.1109/DAC18074.2021.9586270

Cheng, F. (2018). Reactive Streams. *Exploring Java 9*, 87–97. https://doi.org/10.1007/978-1-4842-3330-6_7

Choksuchat, C., & Chantrapornchai, C. (2016). On the development of health tourism semantic web with its parallel engine. *International Journal of Metadata, Semantics and Ontologies*, *11*(1), 16–28. https://doi.org/10.1504/IJMSO.2016.078103

Cicirelli, F., Nigro, C., Nigro, L., & Pupo, F. (2022). Performance Comparison of Two Java-Based Actor Systems. *Lecture Notes in Networks and Systems*, *216*, 79–88. https://doi.org/10.1007/978-981-16-1781-2_9

Clarke, E. M., Klieber, W., Nováček, M., & Zuliani, P. (2012). Model Checking and the State Explosion Problem. Em *Tools for Practical Software Verification*. https://doi.org/https://doi.org/10.1007/978-3-642-35746-6_1

Cunha, C. A., Sobral, J. L., & Monteiro, M. P. (2006). Reusable aspect-oriented implementations of concurrency patterns and mechanisms. *Proceedings of the 5th International Conference on Aspect-oriented Software Development 2006, AOSD'06*, *2006*, 134–145. https://doi.org/10.1145/1119655.1119674

Daugherty, J. (2011). *Java Concurrency Framework*. https://home.cs.colorado.edu/~kena/classes/5448/s11/presentations/csci5448_daugherty_jcf_pres.pdf

De Wael, M., Marr, S., Van Cutsem, T., & Cutsem, T. Van. (2014). *Fork/Join Parallelism in the Wild: Documenting Patterns and Anti-patterns in Java Programs Using the Fork/Join Framework Fork/Join Parallelism in the Wild Documenting Patterns and Anti-Patterns in Java Programs using the Fork/Join Framework*. https://doi.org/10.1145/2647508.2647511ï

Devopedia. (2020). *Race Condition (Software)*. https://devopedia.org/race-condition-software

Domani, T., Goldshtein, G., Kolodner, E. K., Lewis, E., Petrank, E., & Sheinwald, D. (2003). Thread-local heaps for Java. *ACM SIGPLAN Notices*, *38*(2 SUPPL.), 183–194. https://doi.org/10.1145/773039.512439

Dragoni, N., Dustdar, S., Larsen, S. T., & Mazzara, M. (2017). *Microservices: Migration of a Mission Critical System*. http://arxiv.org/abs/1704.04173

Fernández, J. (2016). *Mastering Concurrency Programming with Java 8*. www.ebook3000.com

Fu, H., Wang, Z., Chen, X., Fan, X., Chen, X., & Fan, X. (2018). A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques. *Software Qual J*, *26*, 855–889. https://doi.org/10.1007/s11219-017-9385-3

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). *Java Concurrency in Practice*. Addison-Wesley. https://dl.acm.org/doi/book/10.5555/1076522

Havelund, K., & Roşu, G. (2004). An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, *24*(2), 189–215. https://doi.org/10.1023/B:FORM.0000017721.39909.4B/METRICS

Hera, J., & Jackson, D. (2024). *Applying the Saga Pattern to Ensure Transaction Consistency in Microservices-Based Systems*. https://doi.org/10.13140/RG.2.2.29292.78728

Hofer, P., Gnedt, D., Schörgenhumer, A., Doppler, C., & Mössenböck, H. (2016). *Efficient Tracing and Versatile Analysis of Lock Contention in Java Applications on the Virtual Machine Level*. https://doi.org/10.1145/2851553.2851559

Hovemeyer, D., & Pugh, W. (2004). *Finding Bugs is Easy*. https://doi.org/10.1145/1052883.1052895

Huang, J., & Zhang, C. (2016). Debugging concurrent software: Advances and challenges. *JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY*, *31*(5), 861–868. https://doi.org/10.1007/s11390-016-1669-8

IGNOU. (2017). *Unit-1 Multithreaded Programming*. http://egyankosh.ac.in//handle/123456789/10102

Imam, S., & Sarkar, V. (2015). The Eureka Programming Model for speculative task parallelism. *Leibniz International Proceedings in Informatics, LIPIcs*, *37*, 421–444. https://doi.org/10.4230/LIPICS.ECOOP.2015.421/-/STATS

Ishizaki, K., Daijavad, S., & Nakatani, T. (2011). *Refactoring Java Programs using Concurrent Libraries*. https://doi.org/10.1145/2002962.2002970

Jansson, J. F., & Hellberg, J. (2010). *Concurrency and Parallel Methods for multi-core Platforms*. https://publications.lib.chalmers.se/records/fulltext/124146.pdf

Jhala, R., & Majumdar, R. (2009). Software model checking. *ACM Computing Surveys*, *41*(4). https://doi.org/10.1145/1592434.1592438

Jiang, W., Wen, L., Zhan, J., & Jiang, K. (2020). Design optimization of confidentiality-critical cyber physical systems with fault detection. *Journal of Systems Architecture*, *107*, 101739. https://doi.org/10.1016/J.SYSARC.2020.101739

Joshi, P. K. (2022). Building High -Throughput Payment Transaction Systems with Kafka and Microservices. *International Journal of Science and Research (IJSR)*, *11*, 1658–1665. https://doi.org/10.21275/SR22032110641

Kester, D., Mwebesa, M., & Bradbury, J. S. (2010). How Good is Static Analysis at Finding Concurrency Bugs? *10th IEEE Working Conference on Source Code Analysis and Manipulation*. https://doi.org/10.1109/SCAM.2010.26

Kode, O., & Oyemade, T. (2024). Analysis of Synchronization Mechanisms in Operating Systems. *International Journal on Cybernetics & Informatics*, *13*(5), 43–61. https://doi.org/10.5121/IJCI.2024.130504

Laneve, C. (2019). A lightweight deadlock analysis for programs with threads and reentrant locks. *Science of Computer Programming*, *181*, 64–81. https://doi.org/10.1016/j.scico.2019.06.002

Lea, D. (1996). *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley. https://www.researchgate.net/publication/237005479

Lefort, A. (2023). *A Support for Persistent Memory in Java*. https://theses.hal.science/tel-04161004v1

Luo, Z. Da, Hillis, L., Das, R., & Qi, Y. (2010). Effective Static Analysis to Find Concurrency Bugs In Java. *10th IEEE Working Conference on Source Code Analysis and Manipulation*. https://doi.org/10.1109/SCAM.2010.20

Mahmood, T., Li, J., Pei, Y., Akhtar, F., Butt, S. A., Ditta, A., & Qureshi, S. (2022). An intelligent fault detection approach based on reinforcement learning system in wireless sensor network. *Journal of Supercomputing*, *78*(3), 3646–3675. https://doi.org/10.1007/S11227-021-04001-1

Malakar, S., Bin Haider, T., & Shahriar, R. (2024). *RaceFixer-An Automated Data Race Fixer*. https://doi.org/10.48550/arXiv.2401.04221

Malhotra, R. (2019). Java Reactive Development. *Rapid Java Persistence and Microservices*, 251–265. https://doi.org/10.1007/978-1-4842-4476-0_8

Manzoor, N., Munir, H., & Moayyed, M. (2012). *Comparison of static analysis tools for finding concurrency bugs*. https://doi.org/10.1109/ISSREW.2012.28

Midolo, A., & Tramontana, E. (2023). An Automatic Transformer from Sequential to Parallel Java Code. *Future Internet 2023, Vol. 15, Page 306*, *15*(9), 306. https://doi.org/10.3390/FI15090306

Milani, M. (2023). *Detecting and Resolving Deadlocks in Java Libraries* [Politecnico di Torino]. https://webthesis.biblio.polito.it/secure/28608/1/tesi.pdf

Nobakht, B. (2011). *Deploying active objects onto multicore* [Leiden University]. https://www.researchgate.net/publication/254890637_Deploying_active_objects_onto_multicore

Oaks, Scott., & Wong, Henry. (1999). *Java Threads*. O'Reilly & Associates. https://dl.acm.org/doi/10.5555/249656

Öztürk, S., & Kuzucuoğlu, A. E. (2015). Optimal bid valuation using path finding for multi-robot task allocation. *Journal of Intelligent Manufacturing*, *26*(5), 1049–1062. https://doi.org/10.1007/S10845-014-0909-4/FIGURES/10

Paulino, H., Parreira, D., Delgado, N., Ravara, A., & Matos, A. (2016). From Atomic variables to data-centric concurrency control. *Proceedings of the ACM Symposium on Applied Computing*, *04-08-April-2016*, 1806–1811. https://doi.org/10.1145/2851613.2851734

Penas, O., Plateaux, R., Patalano, S., & Hammadi, M. (2017). Multi-scale approach from mechatronic to Cyber-Physical Systems for the design of manufacturing systems. *Computers in Industry*, *86*, 52–69. https://doi.org/10.1016/J.COMPIND.2016.12.001

Pinto, G., Torres, W., Fernandes, B., Castor, F., & Barros, R. S. M. (2015). A large-scale study on the usage of Java's concurrent programming constructs. *Journal of Systems and Software*, *106*, 59–81. https://doi.org/10.1016/J.JSS.2015.04.064

Ponge, J. (2011). *Fork and Join: Java Can Excel at Painless Parallel Programming Too!* https://www.researchgate.net/publication/268024820_Fork_and_Join_Java_Can_Excel_at_Painless_Parallel_Programming_Too

Ponge, J., Navarro, A., Escoffier, C., Le Mouël, F., & Le, F. (2021). Analysing the Performance and Costs of Reactive Pro-gramming Libraries in Java. *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '21), October 18, 2021, Chicago, IL, USA*, *1*, 51–60. https://doi.org/10.1145/3486605.3486788ï

Protze, J., Thärigen, I., & Wahle, J. (2021). Understanding the Performance of Dynamic Data Race Detection. *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*. https://doi.org/10.1109/Correctness54621.2021.00010

Saad, M. M., & Ravindran, B. (2011). *Supporting STM in Distributed Systems: Mechanisms and a Java Framework*. https://sss.cs.purdue.edu/projects/transact11/papers/Saad.pdf

Sahoo, B. K., & Ray, M. (2018). Concurrency testing using symbolic path finder. *International Journal of Engineering and Technology(UAE)*, *7*(2.6), 275–282. https://doi.org/10.14419/IJET.V7I2.6.10782

Saufi, S. R., Ahmad, Z. A. Bin, Leong, M. S., & Lim, M. H. (2019). Challenges and opportunities of deep learning models for machinery fault detection and diagnosis: A review. *IEEE Access*, *7*, 122644–122662. https://doi.org/10.1109/ACCESS.2019.2938227

Schäfer, M., Sridharan, M., Dolby, J., & Tip, F. (2011). Refactoring Java programs for flexible locking. *Proceedings - International Conference on Software Engineering*, 71–80. https://doi.org/10.1145/1985793.1985804

Serebryany, K., & Iskhodzhanov, T. (2009). ThreadSanitizer-data race detection in practice. *WBIA '09: Proceedings of the Workshop on Binary Instrumentation and Applications*. https://doi.org/10.1145/1791194.1791203

Serebryany, K., Potapenko, A., Iskhodzhanov, T., & Vyukov, D. (2012). Dynamic Race Detection with LLVM Compiler. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *7186 LNCS*, 110–114. https://doi.org/10.1007/978-3-642-29860-8_9

Shao, Z., Peng, J., & Jin, H. (2017). Data race detection by understanding synchronization relationships of thread segments. *Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. https://doi.org/10.1109/PDP.2017.49

Smith, N. (2015). *Java 8: Completable Futures and Asynchronous Pipelines* [Vanderbilt University]. https://hdl.handle.net/1803/12796

Spathoulas, A. (2014). *Assessing Tools for Finding Bugs in Concurrent Java* [University of Edinburgh]. https://homepages.inf.ed.ac.uk/dts/students/spathoulas/spathoulas.pdf

Spieldenner, D., Antakli, A., Spieldenner, T., & Sahota, H. (2024). *Semantic Support Points for on the Fly Knowledge Encoding in Heterogenous Systems*. https://doi.org/10.5220/0012919000003838

Subramaniam, V. (2011). *Programming Concurrency on the JVM*. https://www.oreilly.com/library/view/programming-concurrency-on/9781941222973/

Touhafi, A., Braeken, A., Tahiri, A., & Zbakh, M. (2018). CoderLabs: A cloud-based platform for real-time online labs with user collaboration. *Concurrency and Computation: Practice and Experience*, *30*(12), e4377. https://doi.org/10.1002/CPE.4377

Urbańczyk, J. (2021). *Dynamic Verification of Concurrency in Operating Systems* [Uniwersytet Wrocławski]. https://wmi.uwr.edu.pl/wp-content/uploads/sites/288/2022/07/praca-urbanczyk.pdf

Veen, R., & Vlijmincx, D. (2024). *Virtual Threads, Structured Concurrency, and Scoped Values*. Apress. https://doi.org/10.1007/979-8-8688-0500-4

Visser, W., Havelund, K., Brat, G., Park, S., & Lerda, F. (2003). Model checking programs. *Automated Software Engineering*, *10*(2), 203–232. https://doi.org/10.1023/A:1022920129859/METRICS

Wu, Z., Lu, K., & Wang, X. (2017). *Surveying concurrency bug detectors based on types of detected bugs*. *60*(031101), 27. https://doi.org/10.1007/s11432-015-0203-2

Yönyül, B., Alatlı, O., & Erdur, R. C. (2024). MonARCh: an actor based architecture for dynamic linked data monitoring. *PeerJ Computer Science*, *10*, e2133. https://doi.org/10.7717/PEERJ-CS.2133/SUPP-1

Zeller, A. (2009). *Why Programs Fail: A Guide to Systematic Debugging* (2.ª ed.). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-374515-6.X0000-7

Zhang, Y., Xie, Z., Yue, Y., & Qi, L. (2024). Automatic Refactoring Approach for Asynchronous Mechanisms with CompletableFuture. *Applied Sciences 2024, Vol. 14, Page 8866*, *14*(19), 8866. https://doi.org/10.3390/APP14198866

Zhou, N. (2016). *Autonomic Thread Parallelism and Mapping Control for Software Transactional Memory* [Université Grenoble Alpes]. https://hal.science/tel-01408450v2

Zhou, Y. (2020). Execution Trace Visualization for Java Pathfinder using Trace Compass [School of Electrical Engineering and Computer Science]. Em *DEGREE PROJECT COMPUTER SCIENCE AND ENGINEERING*. https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-286313

ANEXO A – Project Charter

PREPD 2024/2025
Gestão de Projetos

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

PROJECT CHARTER

1. Informação geral	
Nome do Projeto:	Analysis of Tools for Detecting Concurrency Issues in Java: A Comparative Study and Optimization Proposals
Sponsor:	Luís Nogueira (LMN)
Departamento	**Departamento Engenharia Informática (DEI)**

2. Equipa do Projeto				
Cargo	**Nome**	**Departamento**	**Contact Tel**	**E-mail**
Estudante	Luís Vigário	DEI	925768427	1180990@isep.ipp.pt
Orientador	Luís Nogueira	DEI	----------	lmn@isep.ipp.pt

3. Stakeholders		
Nome	Poder	Interesse
Desenvolvedores de Software	Baixo	Alto
Estudante	Alto	Alto
Orientador	Alto	Alto

4. Âmbito

Problema / justificação

Problema:

Crescente Procura por Performance em Sistemas Computacionais

- O aumento da procura por maior performance em sistemas modernos levou ao uso intensivo de processamento paralelo e concorrente em aplicações Java.

Desafios da Concorrência em Java

- A introdução de concorrência nas aplicações adiciona complexidade significativa, com problemas comuns como:
  - **Condições de corrida** (*race conditions*)
  - **Deadlocks**
  - **Violações de atomicidade**

Impacto na Fiabilidade e Eficiência dos Sistemas

- Problemas de concorrência podem causar comportamentos imprevisíveis e complexos de analisar, afetando a fiabilidade e a eficiência geral dos sistemas.

Ferramentas e Frameworks para Deteção de Problemas de Concorrência

- Várias ferramentas e frameworks foram desenvolvidos para auxiliar na deteção de problemas de concorrência em Java.

Limitações e Lacunas das Ferramentas Existentes

- A eficácia das ferramentas atuais é limitada e varia conforme a aplicação, deixando lacunas na capacidade de detetar e resolver problemas de concorrência de forma completa e eficiente.

Research Topics:

Apresentados no documento **Método_Pesquisa.docx**

Objetivos do projeto

Objetivo geral:

- Analisar e comparar ferramentas de deteção de problemas de concorrência em Java, propondo optimizações que possam melhorar a eficiência e precisão na identificação e resolução de condições de corrida, deadlocks e violações de atomicidade.

Objetivos específicos:

- Revisão das Ferramentas Existentes: Realizar uma revisão detalhada das principais ferramentas e frameworks para deteção de problemas de concorrência em Java, incluindo Java Pathfinder (JPF), ThreadSanitizer, FindBugs com Concurrency Bug Detetor, entre outras.
- Estudo comparativo: Avaliar a eficácia dessas ferramentas em diferentes cenários de concorrência usando benchmarks estabelecidos e estudos de caso reais. Comparar as ferramentas em termos de cobertura, precisão, desempenho e usabilidade.
- Identificação de limitações: Identificar as principais limitações das ferramentas actuais, incluindo desafios na deteção de problemas em sistemas complexos e a presença de falsos positivos/negativos.
- Propostas de otimização: Desenvolver novas técnicas ou melhorar as abordagens existentes para melhorar a deteção de problemas de concorrência, concentrando-se na redução do número de falsos positivos e negativos e na melhoria do desempenho das análises.
- Implementação e teste: Implementar as optimizações propostas em uma ou mais estruturas de código aberto. Realizar testes comparativos para validar as melhorias em termos de precisão e eficiência.

Research Questions:

1. Quais são as principais limitações e desafios das ferramentas existentes para a deteção de problemas de concorrência em Java, como condições de corrida, deadlocks e violações de atomicidade, especialmente em sistemas de grande escala?
2. Como a precisão, cobertura, e desempenho das ferramentas de deteção de concorrência em Java variam em diferentes cenários de execução e que impacto têm os falsos positivos e falsos negativos na eficácia dessas ferramentas?
3. Como podem as ferramentas de deteção de problemas de concorrência em Java ser otimizadas, através de novas técnicas ou melhorias nas abordagens atuais, para aumentar a eficiência, escalabilidade e precisão, reduzindo falsos positivos e negativos, especialmente em ambientes com elevada concorrência?
4. Quais são os impactos das melhorias nas ferramentas de deteção de concorrência no ciclo de vida de desenvolvimento de software e na capacidade de assegurar a fiabilidade em sistemas críticos?

Benefícios

Redução de **tempo** em processos concorrentes (**2,5 meses** num ciclo de 12 meses)
Redução de **custo** na aplicação de concorência (**€175.000 a €295.000** num projeto anual, incluindo desenvolvimento e manutenção)
Desenvolvedores Software

Entregáveis

Plano de projeto
Relatório de Dissertação
Apresentação
Documentos intermédios com a pesquisa realizada

5. Tempo

Milestones / Datas

Datas que eu me comprometo:

1 de Dezembro – Entrega da Introdução

22 de Dezembro – Entrega Estado da Arte

Datas obrigatórias:

4 de Janeiro – Submissão de PREPD

6. Custo

Fontes de custo

N/A

7. Pressupostos	**8. Restrições**
N/A	Será apenas considerada a linguagem programação JAVA

9. Riscos

Riscos identificados

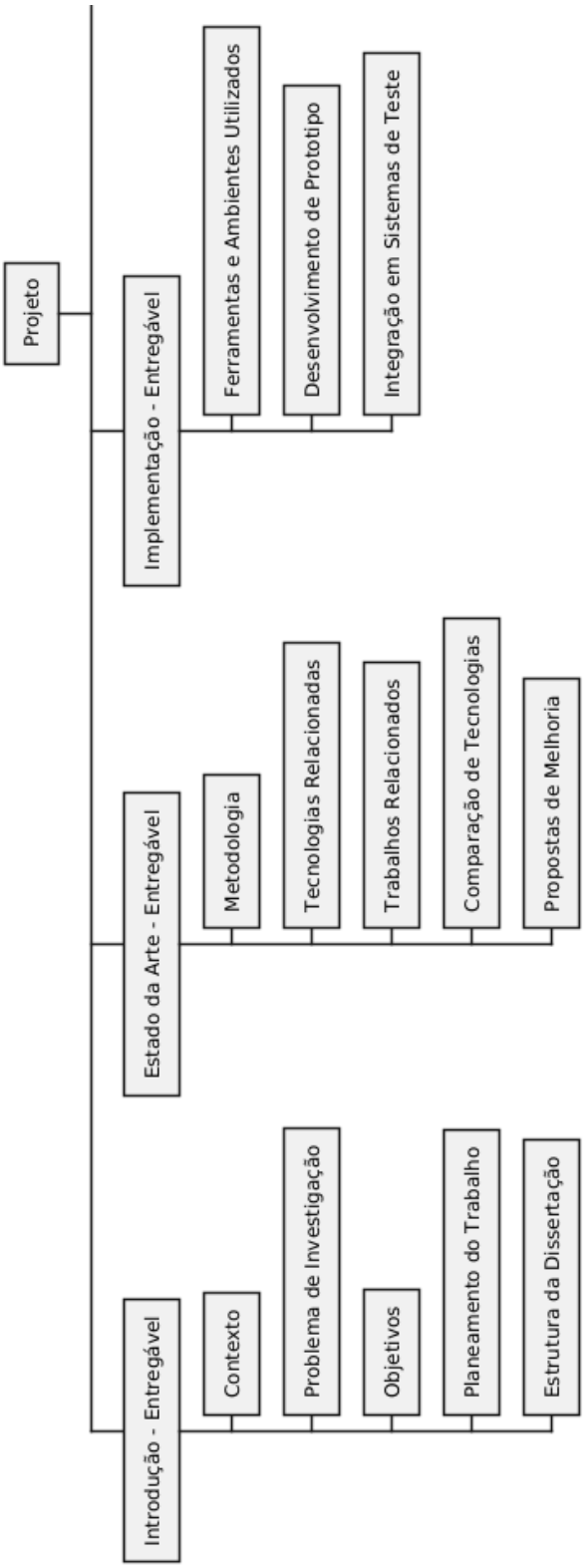
Descrição	Causa	Efeito
Deprecação das bilbiotecas e ferramentas	O suporte para as bibliotecas/ferramentas analisadas pode ser descontinuado	A pesquisa realizada para o documento pode tornar-se obsoleta
Desenvolvimentos futuros das novas versões de Java	Java pode implementar atualizações para abordar os problemas analisados neste documento	A pesquisa realizada para o documento pode tornar-se obsoleta

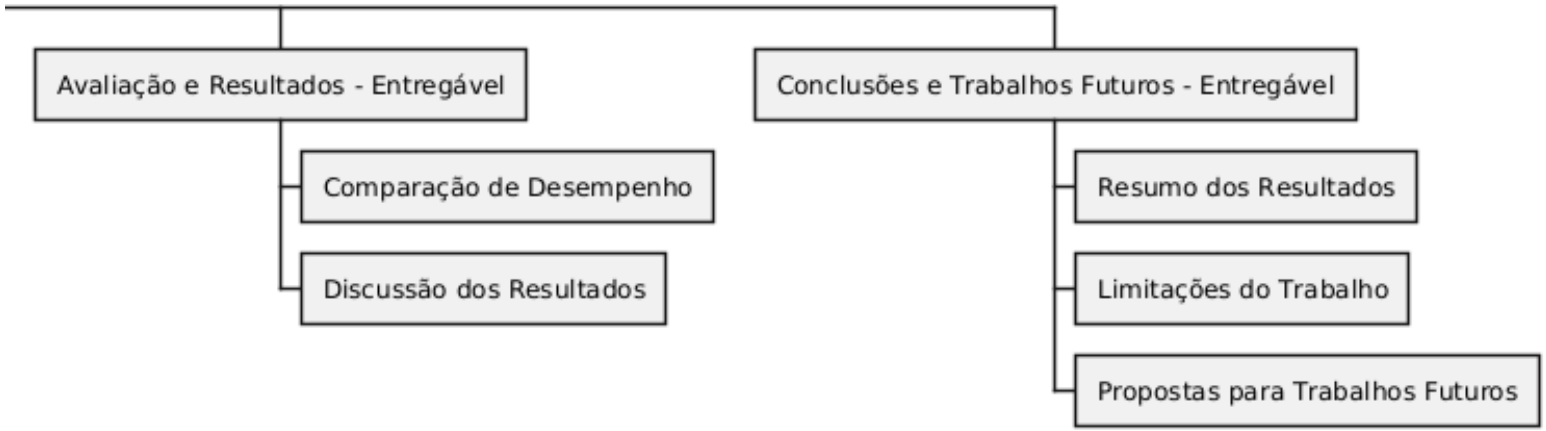
10. Aprovação

	Nome	**Assinatura**	**Data (DD/MM/YYYY)**
Sponsor			
Cliente			
Gestor do Projeto			

Notas

ANEXO B – WBS


Projeto
Introdução - Entregável
Contexto
Problema de Investigação
Objetivos
Planeamento do Trabalho
Estrutura da Dissertação
Estado da Arte - Entregável
Metodologia
Tecnologias Relacionadas
Trabalhos Relacionados
Comparação de Tecnologias
Propostas de Melhoria
Implementação - Entregável
Ferramentas e Ambientes Utilizados
Desenvolvimento de Prototipo
Integração em Sistemas de Teste


Avaliação e Resultados - Entregável
Comparação de Desempenho
Discussão dos Resultados
Conclusões e Trabalhos Futuros - Entregável
Resumo dos Resultados
Limitações do Trabalho
Propostas para Trabalhos Futuros

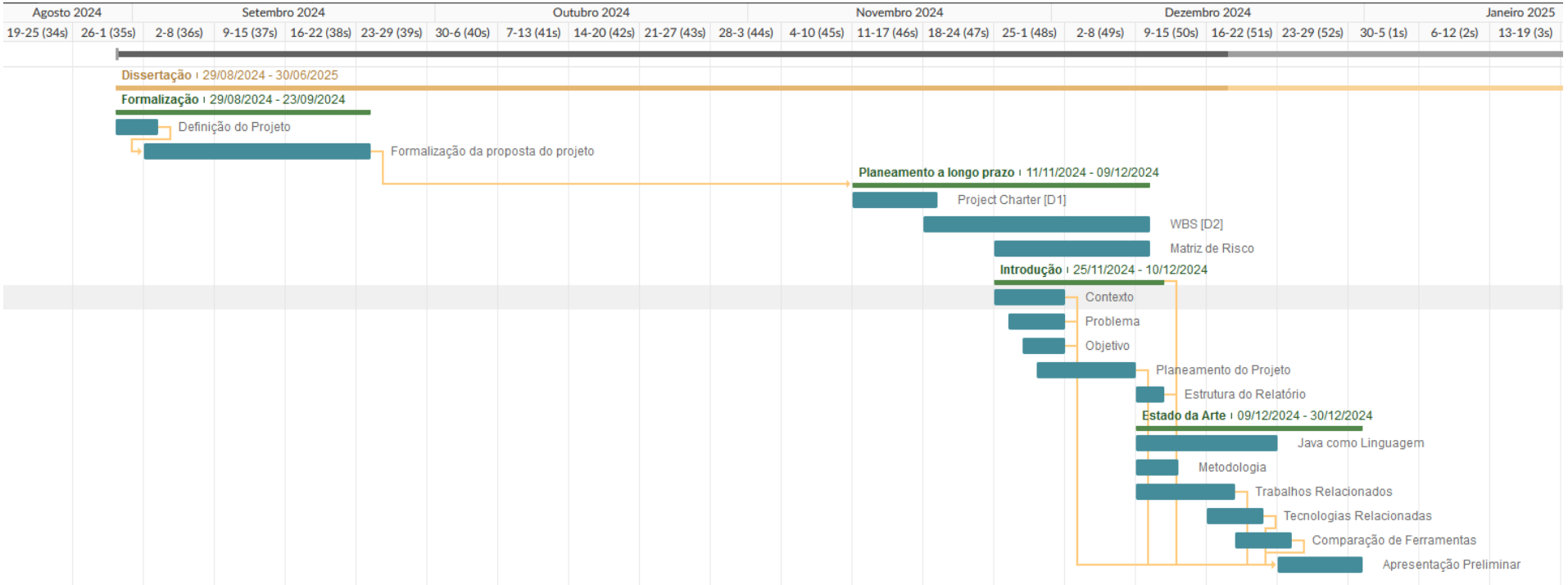
ANEXO C – Matriz de Risco

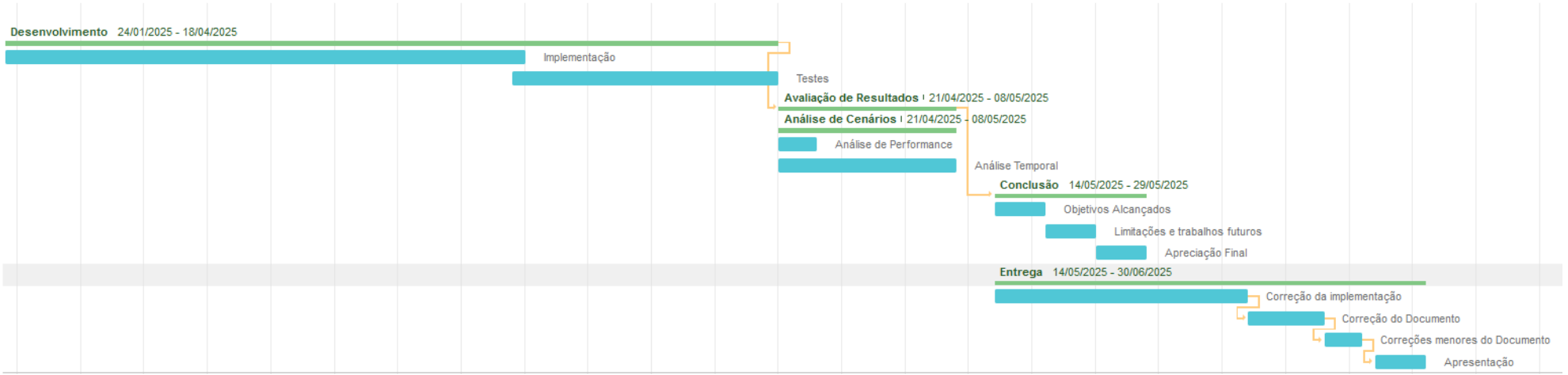
Risk ID	Description	Cause	Effect	Risk Owner	Probability (1-5)
	Description of the risk	Cause of the risk	Effect on the project	Name of person who monitors the risk	Group sourced rough estimate of how likely this is to occur
1	Deprecação das bilbiotecas e ferramentas	O suporte para as bibliotecas/ferramentas analisadas pode ser descontinuado	A pesquisa realizada para o documento pode tornar-se obsoleta	Luís Vigário	2
2	Desenvolvimentos futuros das novas versões de Java	Java pode implementar atualizações para abordar os problemas analisados neste	A pesquisa realizada para o documento pode tornar-se obsoleta	Luís Vigário	2

Impact (1-5)	PI Score	Expected Result, No Action	Risk Response Type	Response description
Rough estimate of how significant the impact of this risk	Probability multiplied by Impact	What will happen if the risk becomes an issue and no action is taken	Decision made by group on how to respond to this risk (see above in blue)	How do you know it is time to put the response into play
5	10	A pesquisa realizada para o documento tornar-se obsoleta	Accept	Quando uma nova realease é publicada
5	10	A pesquisa realizada para o documento tornar-se obsoleta	Accept	Quando uma nova realease é publicada

ANEXO D – GANTT CHART

Task Mode	Task Name	Duration	Start	Finish	Predecessors	Resou Name
	1 Dissertação	**217 days?**	**Sun 01/09/24**	**Mon 30/06/2**		
✓	**1.1 Formalização**	**18 days**	**Sun 01/09/24**	**Tue 24/09/24**		
✓	**1.2 Planeamento a longo prazo**	**22 days**	**Mon 11/11/2**	**Tue 10/12/24**	**4**	
✓	**1.3 Introdução**	**11 days**	**Mon 25/11/2**	**Sun 08/12/24**		
✓	1.3.1 Contexto	2 days	Mon 25/11/24	Tue 26/11/24		
✓	1.3.2 Problema	2 days	Tue 26/11/24	Wed 27/11/24		
✓	1.3.3 Objetivo	2 days	Wed 27/11/24	Thu 28/11/24		
✓	1.3.4 Planeamento do Projeto	7 days	Thu 28/11/24	Fri 06/12/24		
✓	1.3.5 Estrutura do Relatório	2 days	Sat 07/12/24	Sun 08/12/24		
✓	**1.4 Estado da Arte**	**17 days?**	**Sun 08/12/24**	**Sun 29/12/24**		
✓	1.4.1 Java como Linguagem	3 days	Sun 08/12/24	Tue 10/12/24		
✓	1.4.2 Metodologia	3 days	Sun 08/12/24	Tue 10/12/24		
✓	1.4.3 Trabalhos Relacionados	7 days	Sun 08/12/24	Sat 14/12/24		
✓	1.4.4 Tecnologias Relacionadas	4 days	Sat 14/12/24	Wed 18/12/24		
✓	1.4.5 Comparação de Ferramentas	4 days	Wed 18/12/24	Sun 22/12/24		
✓	1.4.6 Apresentação Preliminar	6 days	Mon 23/12/24	Sun 29/12/24	20;19;18;9;10;11;1	
	1.5 Desenvolvimento	**61 days**	**Sat 01/02/25**	**Fri 25/04/25**		
	1.6 Avaliação de Resultados	**14 days**	**Sun 25/05/25**	**Wed 11/06/2**	**22**	
	1.7 Conclusão	**34 days?**	**Wed 14/05/2**	**Mon 30/06/2**	**25**	
	1.8 Entrega	**34 days**	**Wed 14/05/2**	**Mon 30/06/2**		


Agosto 2024
Setembro 2024
Outubro 2024
Novembro 2024
Dezembro 2024
Janeiro 2025
19-25 (34s)
26-1 (35s)
2-8 (36s)
9-15 (37s)
16-22 (38s)
23-29 (39s)
30-6 (40s)
7-13 (41s)
14-20 (42s)
21-27 (43s)
28-3 (44s)
4-10 (45s)
11-17 (46s)
18-24 (47s)
25-1 (48s)
2-8 (49s)
9-15 (50s)
16-22 (51s)
23-29 (52s)
30-5 (1s)
6-12 (2s)
13-19 (3s)
Dissertação 29/08/2024 - 30/06/2025
Formalização 29/08/2024 - 23/09/2024
Definição do Projeto
Formalização da proposta do projeto
Planeamento a longo prazo 11/11/2024 - 09/12/2024
Project Charter [D1]
WBS [D2]
Matriz de Risco
Introdução 25/11/2024 - 10/12/2024
Contexto
Problema
Objetivo
Planeamento do Projeto
Estrutura do Relatório
Estado da Arte 09/12/2024 - 30/12/2024
Java como Linguagem
Metodologia
Trabalhos Relacionados
Tecnologias Relacionadas
Comparação de Ferramentas
Apresentação Preliminar


Desenvolvimento 24/01/2025 - 18/04/2025
Implementação
Testes
Avaliação de Resultados 21/04/2025 - 08/05/2025
Análise de Cenários 21/04/2025 - 08/05/2025
Análise de Performance
Análise Temporal
Conclusão 14/05/2025 - 29/05/2025
Objetivos Alcançados
Limitações e trabalhos futuros
Apreciação Final
Entrega 14/05/2025 - 30/06/2025
Correção da implementação
Correção do Documento
Correções menores do Documento
Apresentação