Execução paralela de programas como suporte ao teste de mutação

Stevão Alves de Andrade

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Stevão Alves de Andrade

Execução paralela de programas como suporte ao teste de mutação

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Márcio Eduardo Delamaro

USP – São Carlos
Maio de 2016

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

A553e Andrade, Stevão Alves de
 Execução paralela de programas como suporte
 ao teste de mutação / Stevão Alves de Andrade;
 orientador Márcio Eduardo Delamaro. - São Carlos
 - SP, 2016.
 122 p.

 Dissertação (Mestrado - Programa de Pós-Graduação
 em Ciências de Computação e Matemática Computacional)
 - Instituto de Ciências Matemáticas e de Computação,
 Universidade de São Paulo, 2016.

 1. engenharia de software. 2. teste de software.
 3. teste de mutação. 4. computação paralela. I.
 Delamaro, Márcio Eduardo, orient. II. Título.

Stevão Alves de Andrade

Parallel execution of programs as a support for mutation
testing

Master dissertation submitted to the Instituto de
Ciências Matemáticas e de Computação – ICMC-
USP, in partial fulfillment of the requirements for the
degree of the Master Program in Computer Science
and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and
Computational Mathematics

Advisor: Prof. Dr. Márcio Eduardo Delamaro

USP – São Carlos
May 2016

Dedico este trabalho para todas as pessoas que contribuíram direta ou indiretamente para a construção do mesmo. Sem sombra de dúvidas, sem o apoio destas o resultado jamais teria sido tão significativo. A elas, toda a honra e mérito.

AGRADECIMENTOS

Agradeço primeiramente a Deus, pois sem a sua vontade, a conclusão deste trabalho jamais seria possível.

Gostaria de agradecer ao meu orientador, Dr. Márcio Delamaro, pela oportunidade que me foi dada. Sem a sua orientação e postura o desenvolvimento deste trabalho não teria obtido êxito. Muito obrigado pelas reuniões, pelas orientações, pelas cobranças e por contribuir não apenas na minha formação acadêmica, mas também na minha formação pessoal.

Gostaria de agradecer a professora Dr. Simone do Rocio Senger de Souza por ter gentilmente me acolhido durante um ano quase como seu orientando quando meu orientador não esteve presente. Bem como ao professor Dr. Paulo Sérgio Lopes de Souza por juntamente com a professora Simone se fazer presente sempre que precisei. Os resultados deste trabalho não seriam os mesmos sem toda dedicação e esforços por vocês realizados.

Gostaria de agradecer aos meus pais Isabel e Raimundo, por sempre me motivarem a buscar o meu melhor e jamais medirem esforços para me ajudar. A vocês meu eterno amor e gratidão. Gostaria de agradecer meus irmãos Stênio e Stanley por toda a ajuda e suporte que me providenciaram, garantindo minha permanência e total dedicação na pós-graduação mesmo nos momentos mais difíceis. Jamais terei como retribuir tamanha dedicação.

A todos os amigos da república Eskoria que gentilmente me abrigou quando cheguei em São Carlos, por fazerem de tudo para que minha estadia e adaptação fosse mais agradável possível.

A todos as amizades que tive o prazer de conhecer em São Carlos. Não citarei nomes para não correr o risco de esquecer o nome de alguém. Todos os amigos que tive a incrível oportunidade de conhecer no LabES (Alunos e Professores), sem vocês não seria a mesma coisa. Aos amigos do grupo Testpar que tanto me ajudaram no início desta caminhada. Suas contribuições são inestimáveis e jamais serão esquecidas.

Aos amigos dos demais laboratórios de pesquisa com quem convivi esses últimos três anos e principalmente aos amigos que posteriormente vieram a dividir moradia comigo. A companhia de vocês é inestimável. Meu muito obrigado por tudo.

Por fim, agradeço o suporte financeiro da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), sob o processo nº DS-8435441/M e posteriormente ao suporte financeiro da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), sob o processo nº 2013/16242-3.

*“Se consegui ver mais longe,
é porque estava aos ombros de gigantes.”
(Isaac Newton)*

RESUMO

ANDRADE, S. A.. **Execução paralela de programas como suporte ao teste de mutação**. 2016. 122 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Teste de software desempenha um papel fundamental no processo de produção de um produto de software de qualidade. Com o passar dos anos, diversas técnicas e critérios de teste de software foram desenvolvidos a fim de estabelecer meios e métricas para guiar a criação de casos de teste efetivos, capazes de evidenciar defeitos no produto avaliado. Dentre os principais critérios para teste de software está o Teste de Mutação, que foi amplamente difundido e é tido como uma das abordagens mais eficazes para guiar a criação de conjuntos de casos de teste capazes de revelar defeitos em software. Entretanto, à medida que esse critério possui uma grande efetividade para revelar defeitos, ele peca pelo baixo poder de escalabilidade, o que acaba comprometendo diretamente a sua capacidade de aplicação. Neste sentido, diversos estudos foram desenvolvidos nesta área dedicando-se a aprimorar o seu desempenho e torná-lo uma alternativa viável para aplicação durante a fase de teste de software. Este trabalho apresenta indícios de que a utilização de estruturas complexas de processamento pode apoiar a aplicação do Teste de Mutação. Para tal, foi concebida uma arquitetura que possibilite a aplicação do Teste de Mutação em paralelo. Após a implementação da arquitetura foram avaliados cinco algoritmos de balanceamento de carga responsáveis por controlar a distribuição e execução do Teste de Mutação. Durante a avaliação experimental da arquitetura e dos algoritmos, observou-se que nos piores cenários avaliados foi possível atingir um ganho de desempenho acima de 70% em relação à aplicação sequencial convencional do Teste de Mutação enquanto nos melhores cenários o ganho de desempenho foi acima 95%, contudo, necessitando utilizar-se de uma infraestrutura mais robusta para a execução da arquitetura.

Palavras-chave: engenharia de software, teste de software, teste de mutação, computação paralela.

ABSTRACT

ANDRADE, S. A.. **Execução paralela de programas como suporte ao teste de mutação.** 2016. 122 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Software testing plays a fundamental role in the development process of quality software systems. Over the years, many software testing techniques and criteria were developed to provide means and metrics to guide the development of effective test cases, able to find defects on the product being assessed. Among major criteria for software testing is the mutation testing, which was broadly broadcast and is likely one of the most effective approaches for creating sets of test cases able to uncover software bugs. However, although mutating testing has a great effectiveness to uncover defects in a product, it suffers from low scalability, which directly compromises its applicability. In this sense, many studies were developed in this area aiming at improving the performance of that criterion and make it a viable alternative for its application throughout the software testing process. This work presents evidence that the use of complex structures of processing can support mutation testing application. For this, it was established an architecture that enables mutation testing to be performed in parallel. After implementing the architecture, five load balance algorithms to controlling the distribution and execution of mutation testing were analyzed. During the experimental evaluation of the architecture and algorithms, it was observed that in the worst evaluated scenarios it was possible to reach a gain in performance of up to 70% in comparison to the conventional application (sequential). In the best scenarios the gain was over 95% in exchange of using a more robust infrastructure for the execution of the architecture.

Key-words: software engineering, software testing, mutation testing, parallel computing.

LISTA DE ILUSTRAÇÕES

Figura 1 – Processo de geração de mutantes.	37
Figura 2 – Programa original P e Mutantes M_1 e M_2 gerados pelo uso do operador SMVB.	38
Figura 3 – Processo genérico do Teste de Mutação. Adaptado de (OFFUTT; UNTCH, 2001).	39
Figura 4 – Visão cronologica do desenvolvimento de técnicas para viabilizar a aplicação do Teste de Mutação. Adaptado de (JIA; HARMAN, 2011).	41
Figura 5 – Organização de computadores paralelos adaptado de (STALLINGS, 2010).	54
Figura 6 – Teste de Mutação em Paralelo - Arquitetura	69
Figura 7 – Teste de Mutação em Paralelo - Diagrama de Sequência	70
Figura 8 – Diagrama de Máquina de Estados - Comportamento do servidor	71
Figura 9 – Diagrama de Máquina de Estados - Comportamento do escalonador	72
Figura 10 – Diagrama de máquina de estados - Comportamento dos executores	73
Figura 11 – Comparativo da média do tempo de execução de 4 executores em relação à execução sequencial do Teste de Mutação.	83
Figura 12 – Comparativo da média do tempo de execução de 8 executores em relação à execução sequencial do Teste de Mutação.	84
Figura 13 – Comparativo da média do tempo de execução de 12 executores com relação à execução sequencial do Teste de Mutação.	85
Figura 14 – Comparativo da média do tempo de execução de 16 executores em relação à execução sequencial do Teste de Mutação.	86
Figura 15 – Comparativo da média do tempo de execução de 32 executores com relação à execução sequencial do Teste de Mutação.	86
Figura 16 – Redução do tempo de resposta à medida que o número de executores aumenta.	87
Figura 17 – Gráficos de probabilidade normal dos tempos de execução para cada algoritmo utilizando 4 executores.	95
Figura 18 – Gráficos de probabilidade normal dos tempos de execução para cada algoritmo utilizando 8 executores.	95
Figura 19 – Gráficos de probabilidade normal dos tempos de execução para cada algoritmo utilizando 12 executores.	96
Figura 20 – Gráficos de probabilidade normal dos tempos de execução para cada algoritmo utilizando 16 executores.	96
Figura 21 – Gráficos de probabilidade normal dos tempos de execução para cada algoritmo utilizando 32 executores.	97

Figura 22 – *Boxplot* para comparação da diferença entre grupos (I) 101

Figura 23 – *Boxplot* para comparação da diferença entre grupos (II) 102

Figura 24 – *Boxplot* para comparação da diferença entre grupos utilizando 32 executores. 102

LISTA DE QUADROS

Quadro 1 – Lista de ferramentas que apoiam a aplicação do Teste de Mutação	46
Quadro 2 – Programas utilizados para execução do experimento	80
Quadro 3 – Algoritmo com melhor desempenho para cada objeto de estudo utilizado no experimento.	93
Quadro 4 – Programas utilizados no experimento por Mateo e Usaola (2013)	104

LISTA DE ALGORITMOS

Algoritmo 1 – Algoritmo de Distribuição de Mutantes por Operador	63
Algoritmo 2 – Conjuntos de Teste Distribuídos	64
Algoritmo 3 – Mutantes Distribuídos sob Demanda	65
Algoritmo 4 – Casos de teste distribuídos sob demanda	66
Algoritmo 5 – Execução Paralela com Ordenação e Ranqueamento Dinâmicos	67

LISTA DE TABELAS

Tabela 1 – Características dos computadores utilizados no experimento	78
Tabela 2 – Média do tempo de execução para o experimento completo utilizando 4 executores.	84
Tabela 3 – Média do tempo de execução para o experimento completo utilizando 8 executores.	84
Tabela 4 – Média do tempo de execução para o experimento completo utilizando 12 executores.	85
Tabela 5 – Média do tempo de execução para o experimento completo utilizando 16 executores.	85
Tabela 6 – Média do tempo de execução para o experimento completo utilizando 32 executores.	86
Tabela 7 – Resultados individuais do experimento para o algoritmo DMBO	88
Tabela 8 – Resultados individuais do experimento para o algoritmo DTC	89
Tabela 9 – Resultados individuais do experimento para o algoritmo GMOD	90
Tabela 10 – Resultados individuais do experimento para o algoritmo GTCOD.	91
Tabela 11 – Resultados individuais do experimento para o algoritmo PEDRO.	92
Tabela 12 – Valor de significância <i>p-value</i> para o teste de <i>Shapiro-Wilk</i>	96
Tabela 13 – Resultados do <i>T-Test</i> para 4 executores.	97
Tabela 14 – Resultados do <i>T-Test</i> para 8 executores.	97
Tabela 15 – Resultados do <i>T-Test</i> para 12 executores.	98
Tabela 16 – Resultados do <i>T-Test</i> para 16 executores.	98
Tabela 17 – Resultados do <i>T-Test</i> para 32 executores.	98
Tabela 18 – Resultados do teste estatístico <i>ANOVA-Test</i>	99
Tabela 19 – Resultados para teste <i>TUKEY HSD-Test</i> para configurações com 4 e 8 execu- tores.	100
Tabela 20 – Resultados para teste <i>TUKEY HSD-Test</i> para configurações com 12 e 16 executores.	100
Tabela 21 – Resultados para teste <i>TUKEY HSD-Test</i> para configuração com 32 executores.	101
Tabela 22 – Resultados para execução com 8 e 19 máquinas (Mateo e Usaola (2013)) . .	104
Tabela 23 – Resultados para execução com 40 máquinas (Mateo e Usaola (2013))	104
Tabela 24 – Resultados para execução com 80 máquinas (Mateo e Usaola (2013))	104
Tabela 25 – Segunda etapa - Avaliação dos algoritmos PEDRO e GMOD (<i>Analysis</i>) . . .	105

SUMÁRIO

1	INTRODUÇÃO	25
1.1	Motivação e Objetivos	28
1.2	Estrutura	29
2	FUNDAMENTOS SOBRE TESTE DE SOFTWARE	31
2.1	Teste de Software	31
2.1.1	<i>Técnicas e Critérios de Teste</i>	33
2.1.1.1	<i>Teste Funcional</i>	33
2.1.1.2	<i>Teste Estrutural</i>	34
2.1.1.3	<i>Teste Baseado em Defeitos</i>	35
2.2	Visão sobre o Teste de Mutação	36
2.3	Técnicas para Redução de Custo	40
2.3.1	<i>Redução de mutantes</i>	41
2.3.2	<i>Redução do custo de aplicação</i>	42
2.4	Ferramentas de Apoio	46
2.5	Considerações Finais	48
3	FUNDAMENTOS SOBRE COMPUTAÇÃO CONCORRENTE	51
3.1	Fundamentos de Programação Concorrente	51
3.2	Arquitetura de Computadores Paralelos	52
3.3	Desenvolvimento de Aplicações Concorrentes	54
3.4	Avaliação de Desempenho	57
3.5	Considerações Finais	58
4	POTENCIALIZANDO O TESTE DE MUTAÇÃO COM COMPU- TAÇÃO PARALELA	61
4.1	Algoritmos	62
4.1.1	<i>Algoritmo DMBO: Distribute Mutants Between Operators</i>	62
4.1.2	<i>Algoritmo DTC: Distribute Test Cases</i>	63
4.1.3	<i>Algoritmo GMOD: Given Mutants On Demand</i>	64
4.1.4	<i>Algoritmo GTCOD: Given Test Cases on Demand</i>	65
4.1.5	<i>Algoritmo PEDRO: Parallel Execution with Dynamic Ranking and Ordering</i>	66
4.2	Arquitetura	68

4.2.1	<i>Servidor</i>	71
4.2.2	<i>Escalonador</i>	72
4.2.3	<i>Executor</i>	73
4.3	<i>Considerações Finais</i>	74
5	TESTE DE MUTAÇÃO EM PARALELO: EXPERIMENTOS	75
5.1	Planejamento	75
5.1.1	<i>Definição dos objetivos</i>	76
5.1.2	<i>Ambiente</i>	77
5.1.3	<i>Formulação das hipóteses</i>	78
5.1.4	<i>Seleção de variáveis</i>	78
5.1.5	<i>Seleção dos objetos experimentais</i>	79
5.1.6	<i>Descrição dos ambientes de execução</i>	81
5.2	Condução	81
5.3	Coleta e Análise dos Dados	82
5.3.1	<i>Coleta</i>	82
5.3.2	<i>Análise descritiva</i>	82
5.3.2.1	<i>Teste de Mutação sequencial x Teste de Mutação em paralelo</i>	83
5.3.2.2	<i>Avaliando o desempenho dos algoritmos de balanceamento de carga</i>	87
5.3.2.3	<i>Testando hipóteses</i>	94
5.3.3	<i>Ameaças à validade</i>	102
5.4	Comparando resultados	103
5.5	Considerações Finais	106
6	CONCLUSÕES E TRABALHOS FUTUROS	109
6.1	Problema de Pesquisa e Questões de Pesquisa	109
6.2	Teste de Mutação em Paralelo: Vantagens na Adoção	111
6.3	Contribuições	111
6.4	Limitações	112
6.5	Trabalhos Futuros	113
6.6	Conclusões	113
	REFERÊNCIAS	115

INTRODUÇÃO

O processo de desenvolvimento de um software envolve uma série de atividades nas quais são aplicados diversos métodos, técnicas e ferramentas com objetivo de auxiliar este processo. Contudo, a produção de um software ainda está sujeita a problemas e tais problemas podem comprometer diretamente a qualidade do produto.

Segundo [Pressman \(2010\)](#), a Engenharia de Software é definida como sendo uma área que tem como objetivo aplicar métodos científicos em benefício da produção de software com objetivo de produzir produtos com baixo custo e alta qualidade. Durante o processo de desenvolvimento, são conduzidas atividades que têm por finalidade a obtenção dos níveis de qualidade desejados para o produto desenvolvido.

Dentre as técnicas de verificação e validação de software, a atividade de teste é uma das mais utilizadas, constituindo-se em um importante métrica para fornecer evidências sobre a confiabilidade de um produto. A atividade de teste consiste de uma análise dinâmica do produto, sendo uma atividade relevante para a identificação e eliminação de possíveis defeitos que podem ser erroneamente introduzidos durante o seu desenvolvimento ([DELAMARO; MALDONADO; JINO, 2007](#)).

Conforme mostrado por [Myers et al. \(2004\)](#) e [Delamaro, Maldonado e Jino \(2007\)](#), o processo de teste de produtos de software pode ser dividido basicamente em quatro etapas:

- **Planejamento de testes:** define os objetivos e especifica as atividades a serem desenvolvidas a fim de atingir os objetivos propostos
- **Projeto de casos de teste:** define o processo de construção dos casos de teste, os quais são projetados levando em consideração os requisitos de teste.
- **Execução:** consiste na atividade de execução dos casos de teste e envolve rotinas de preparação de ambiente de execução, execução dos casos de teste, definição de rastreabilidade,

definição de estimativas, coleta de resultados, etc.

- **Avaliação dos resultados dos testes:** atividade de checagem de relatórios de execução, verificar a necessidade de adesão de mais casos de teste bem como a divulgação de *status report* para todas as pessoas envolvidas no projeto em teste.

Essas atividades devem ser planejadas e conduzidas durante todo o processo de desenvolvimento do software. Para isso, um ponto fundamental é a seleção dos casos de teste que serão utilizados. Delamaro, Maldonado e Jino (2007) afirmam que o programa em teste deveria ser idealmente executado com todo o domínio de valores de entrada válidos, para garantir que o mesmo está livre de defeitos. Contudo, esta é uma tarefa impraticável, pelo fato de o domínio de entradas válidas mesmo para programas simples tender ao infinito. Portanto, é necessária a criação de subdomínios de teste. Esta prática tem por objetivo subdividir o domínio de entradas em subconjuntos que possuam dados de teste semelhantes, ou seja, que façam com que o programa em teste comporte-se de maneira similar.

Para identificar dados de teste que pertençam ao mesmo subdomínio são estabelecidas regras (chamadas de critérios), assim, dependendo do critério utilizado, é possível obter diferentes subdomínios e, portanto, conjuntos de casos de teste com características diferentes. Em geral, critérios de teste são definidos com objetivo de estabelecer como selecionar casos de teste que possuam alta probabilidade de encontrar defeitos com um mínimo de esforço possível. Portanto, o teste dito como bem-sucedido é aquele capaz de fazer com que o resultado produzido pela sua execução seja diferente do resultado esperado.

Em geral, os critérios de teste de software são estabelecidos a partir de três técnicas: a funcional, a estrutural e a baseada em defeitos. Na técnica funcional, os critérios e requisitos de teste são definidos com base na especificação do software; na técnica estrutural, os critérios e requisitos são derivados essencialmente a partir das características de uma implementação em teste; e, na técnica baseada em defeitos, os critérios e requisitos de teste são definidos com base no conhecimento sobre defeitos típicos cometidos por desenvolvedores durante o processo de desenvolvimento de software. As técnicas e critérios de teste consideram essencialmente duas questões importantes em relação à atividade de teste: (i) seleção de casos de teste, e (ii) avaliação da adequação dos casos de testes em relação ao programa em teste (MYERS *et al.*, 2004).

Um dos critérios baseados em defeitos é o Teste de Mutação. Esse critério surgiu com o objetivo de avaliar a efetividade/qualidade de um dado conjunto de casos de teste. O Teste de Mutação utiliza informações sobre os enganos típicos cometidos pelos desenvolvedores durante o processo de desenvolvimento para derivar requisitos de teste.

Assim, os enganos típicos de um determinado domínio ou paradigma de desenvolvimento são definidos, caracterizados e implementados como operadores de mutação que, durante a atividade de teste, geram versões modificadas (mutantes) do produto em teste. O objetivo é

auxiliar o testador na seleção de casos de teste que demonstrem que os defeitos modelados pelos operadores de mutação não estão presentes no produto avaliado.

Essa técnica é extremamente dependente dos enganos típicos do domínio em que é aplicada, desse modo, é necessário que os mesmos sejam caracterizados para que o critério possa ser aplicado de maneira efetiva. Neste sentido, existem diversos trabalhos que propõem essa abordagem para diferentes domínios, como por exemplo o teste de programas (DEMILLO, 1980; OFFUTT; VOAS; PAYN, 1996; DELAMARO; MALDONADO, 1999) e o teste de especificações formais (FABBRI *et al.*, 1995; AICHERNIG; JIFENG, 2009).

Além de ser uma técnica eficaz quando aplicada para testar um programa, o Teste de Mutação tem sido amplamente utilizado com objetivo de avaliar a eficácia de outras técnicas de validação em revelar defeitos. Para isso, o Teste de Mutação é utilizado para gerar versões do programa contendo defeitos e experimentos são conduzidos para avaliar o quanto uma técnica ou critério de teste é capaz de revelar os defeitos modelados pelos operadores de mutação. A vantagem do uso de Teste de Mutação nesse contexto é o fato dessa abordagem apresentar um processo bem definido e preciso para semear os defeitos no produto a ser avaliado. Isso facilita a replicação dos resultados em outros experimentos (ANDREWS; BRIAND; LABICHE, 2005).

Um dos problemas para a aplicação do Teste de Mutação é o alto custo computacional envolvido na sua aplicação. Uma vez que o número de mutantes gerados pode ser muito grande até mesmo para programas de pequeno porte. Esse custo está relacionado ao tempo necessário para executar os mutantes gerados e o tempo de identificação de mutantes equivalentes, esse último normalmente realizado manualmente.

Esta pesquisa sugere que o Teste de Mutação pode ser potencializado com a utilização de uma arquitetura para possibilitar a execução paralela do Teste de Mutação e de algoritmos de balanceamento de carga para coordenar a distribuição dos dados durante a execução do teste. Para o desenvolvimento desta pesquisa foi adotada a fórmula proposta por Booth, Colomb e Williams (2009), assim o problema de pesquisa e as questões de pesquisa que nortearam o desenvolvimento deste trabalho, bem como contribuições em potencial são apresentadas a seguir:

- **Problema de Pesquisa:** pesquisas relacionadas à melhoria da eficiência na aplicação do Teste de Mutação direcionam-se em como reduzir o número de mutantes gerados sem impactar a eficiência e aplicabilidade da técnica. Contudo, pouco tem sido investigado sobre como ambientes de execução paralela podem ser utilizados para melhorar os seus resultados e apoiar a aplicabilidade da técnica.
- **Questões de Pesquisa (QPs):**
 - **QP₁** : É possível melhorar a eficiência computacional na aplicabilidade do Teste de Mutação utilizando técnicas de computação paralela?

- **QP₂** : Existe diferença significativa entre o tempo de resposta dos algoritmos de balanceamento de carga implementados na arquitetura proposta?
- **QP₃** : Como os resultados obtidos se comportam em relação a trabalhos similares existentes na literatura?
- **Contribuições em Potencial:** Esta pesquisa tem como interesse entender quanto a aplicabilidade do Teste de Mutação pode ser beneficiada ao utilizar técnicas de computação paralela. Outro aspecto fundamental é comparar as diferentes abordagens apresentadas e como os resultados obtidos se comparam com outros trabalhos existentes na literatura que serviram de inspiração para a condução desta pesquisa. Assim, as consequências conceituais (BOOTH; COLOMB; WILLIAMS, 2009) desta investigação contribuirão para que pesquisadores e práticos tenham um maior entendimento das características desta abordagem e possam pensar em aplicá-la ou estendê-la a fim de atingir novos desdobramentos.

1.1 Motivação e Objetivos

Abordagens como a geração automática de casos de testes possibilitam a verificação de maneira intensiva. A geração de dados de teste é um processo de identificação de dados do domínio de entrada para um programa de acordo com os critérios de teste e é utilizada com o objetivo de verificar se uma dada implementação contém falhas. Quando aplicado em conjunto com Teste de Mutação, os mutantes gerados precisam ser executados diversas vezes com um número elevado de casos de teste, o que faz com que a eficiência na execução do critério se torne ainda mais importante. Além disso, abordagens como avaliações experimentais, em que todo o conjunto de mutantes necessita ser executado com todo o domínio de casos de teste, para que dados referentes ao processo possam ser obtidos com fins de análise reforçam a necessidade de uma abordagem que flexibilize a aplicação do Teste de Mutação.

Uma importante técnica para redução de custo é a execução paralela, a qual é amplamente explorada em outros contextos, como previsão de tempo, dinâmica de fluídos, processamento de imagens, química quântica entre outros (QUINN, 2004).

Jia e Harman (2011) apresentam um *survey* sobre o Teste de Mutação e descrevem técnicas, investigadas durante a década de 90, para possibilitar a execução paralela do Teste de Mutação. Tais abordagens concentram-se em explorar as diferentes arquiteturas computacionais disponíveis na época. Com os avanços computacionais atuais, novas abordagens podem ser exploradas a fim de aperfeiçoar esta abordagem.

Recentemente Mateo e Usaola (2013) apresentaram um trabalho que explora técnicas de computação paralela para superar o gargalo de processamento existente na aplicação do Teste de Mutação para programas desenvolvidos na linguagem de programação Java. Em resumo, esse

trabalho implementa um conjunto de cinco algoritmos de balanceamento de carga aplicados a uma arquitetura para possibilitar a coordenação e execução do Teste de Mutação em paralelo.

Um aspecto importante no processo de experimentação é a replicação. Replicações experimentais são conduzidas a fim de consolidar o conhecimento produzido a cerca de um determinado tema de pesquisa. A principal finalidade dessa atividade é verificar se os resultados obtidos em trabalhos anteriores podem ser reproduzidos. [Juristo e Gómez \(2012\)](#) apontam que se o contexto e os artefatos utilizados na replicação diferem dos utilizados em um estudo original, então a replicação ajuda a aumentar a confiança de que os resultados obtidos no estudo original não foram gerados por conta da utilização de uma configuração específica de execução. Essa atividade contribui para mitigar ameaças a validades sobre os resultados tanto do experimento original, quanto da replicação.

Outro ponto chave para aumentar a confiança nos resultados é a disponibilização dos dados produzidos nos experimentos. Essa atitude pode encorajar e facilitar a realizarem replicações por outros pesquisadores. Para [Shull *et al.* \(2008\)](#), essa prática deve ser constantemente reforçada, pois pesquisadores estão sujeitos a cometer enganos durante a análise e extração de resultados. Então, o compartilhamento dos dados possibilita que outros pesquisadores possam analisar e replicar o estudo, contribuindo para aumentar a confiança dos resultados produzidos.

Os resultados de um experimento podem ser influenciados por diversos fatores, como as ferramentas utilizadas, linguagem de programação, configurações de rede, configurações de ambientes de execução, etc. Portanto, o objetivo deste trabalho é propor uma alternativa para a redução do custo computacional envolvido na aplicação do Teste de Mutação no contexto de programas escritos na linguagem de programação C. Esse trabalho foi desenvolvido como uma replicação do trabalho desenvolvido por [Mateo e Usaola \(2013\)](#), foi desenvolvido um modelo para a aplicação do Teste de Mutação em paralelo com base no modelo proposto em ([MATEO; USAOLA, 2013](#)) com a finalidade de aperfeiçoar os recursos computacionais envolvidos durante essa atividade no contexto do teste de software aplicado a programas desenvolvidos na linguagem de programação C, considerando questões fundamentais como disponibilidade e escalabilidade.

1.2 Estrutura

Esta dissertação está organizada em seis capítulos. Este capítulo fez uma pequena introdução sobre a área de pesquisa investigada apresentando as questões que nortearam o desenvolvimento desta pesquisa, bem como a motivação necessária para o desenvolvimento deste trabalho.

O Capítulo 2 apresenta os principais conceitos relacionados a teste de software, e trata em específico sobre o Teste de Mutação, suas hipóteses e suas limitações. É dada uma visão sobre o problema relacionado ao seu custo computacional, descrevendo os principais estudos que visam solucionar este problema. O capítulo divide as técnicas em três abordagens (*I*) *do fewer*,

(II) *do smarter* e (III) *do faster* e descreve as principais técnicas utilizadas em cada abordagem. Por fim, o capítulo encerra fazendo uma breve descrição da ferramenta de Teste de Mutação *Proteum* que é utilizada durante os experimentos para validar a proposta deste trabalho.

No Capítulo 3 são descritos os principais conceitos sobre programação paralela, demonstrando os principais conceitos relacionados a paralelismo de dados, que é a abordagem utilizada na arquitetura desenvolvida nesse trabalho.

No Capítulo 4 é apresentada a abordagem proposta para aplicação do Teste de Mutação em paralelo. A ênfase deste capítulo é descrever como é possível se aproveitar de uma estrutura de computadores ligados em rede para flexibilizar a aplicação do Teste de Mutação em paralelo.

O Capítulo 5 descreve os experimentos conduzidos para avaliar a implementação proposta. São apresentadas avaliações e comparações da abordagem desenvolvida nesta dissertação e os resultados são discutidos com trabalhos relacionados existentes na literatura.

Por fim, o Capítulo 6 revisita o problema de pesquisa desta dissertação e as suas questões de pesquisa. Nele também é feito um sumário das contribuições desta pesquisa, bem como as suas limitações e sugestões para direção de trabalhos futuros. O capítulo encerra-se com a discussão sobre a relevância desta pesquisa e os desafios enfrentados durante a o desenvolvimento da pesquisa.

FUNDAMENTOS SOBRE TESTE DE SOFTWARE

Neste capítulo são apresentados os principais conceitos sobre teste de software relacionados a este trabalho. Os conceitos apresentados neste capítulo fornecem um embasamento teórico que é de fundamental importância para o entendimento a respeito do problema de pesquisa abordado.

Inicialmente são apresentados conceitos básicos sobre teste de software que representam parte da motivação para o desenvolvimento deste trabalho (Seção 2.1). Na Seção 2.2 são definidos os conceitos relacionados sobre a técnica de testes baseado em defeitos e sobre o critério de Teste de Mutação. O foco dessa seção é abordar a definição do critério, as hipóteses que suportam a sua aplicação e suas principais características e limitações. A Seção 2.3 aborda um conjunto de trabalhos que investigam como solucionar as limitações da aplicação do Teste de Mutação, levando em consideração principalmente técnicas para melhorar a eficiência da sua aplicação. A Seção 2.4 faz um resumo das principais ferramentas que dão suporte à aplicação do Teste de Mutação e dá destaque à ferramenta *Proteum*, que foi utilizada durante deste trabalho. Nessa seção são apresentados os módulos que compõem a ferramenta, descrevendo suas principais funções e características de implementação. Por fim a Seção 2.5 apresenta as considerações finais deste capítulo.

2.1 Teste de Software

Técnicas e critérios de teste de software permitem que o projetista possa utilizar uma abordagem sistemática e teoricamente bem fundamentada para conduzir a atividade de teste. Estas abordagens formam um mecanismo que tem como principal objetivo auxiliar na garantia da qualidade dos casos de teste produzidos e de identificar o maior número de defeitos no produto de software avaliado. Diversas técnicas são descritas na literatura para auxiliar a atividade de

teste de software, variando essencialmente na origem das informações a serem utilizadas no processo de definição dos requisitos de teste (SANCHES, 2001).

Para garantir que o produto avaliado seja entregue livre de defeitos, ou seja, para que tais defeitos sejam descobertos antes da entrega final do produto, é necessário a aplicação de atividades de Validação, Verificação e Teste (VV&T), que tem como principal objetivo garantir que o software esteja em conformidade com a especificação base que foi definida durante o seu ciclo de desenvolvimento. Dentre as técnicas de VV&T, a atividade de verificação tem como objetivo identificar se o desenvolvedor está conduzindo de maneira correta o processo de desenvolvimento do software, enquanto a atividade de validação permite que o desenvolvedor confronte o produto que está sendo desenvolvido com os requisitos definidos pelo cliente, visando estabelecer se o produto está adequado às suas necessidades para as quais foi projetado (MYERS *et al.*, 2004).

A atividade de teste de software é composta por uma série de processos desenvolvidos para garantir que o software possa realizar o conjunto de tarefas para as quais fora previamente projetado. Um produto de software deve então ser previsível e consistente, jamais oferecendo alguma ação inesperada para os usuários. A atividade de teste ajuda a garantir que o software avaliado atenda aos requisitos para os quais foi desenvolvido, contudo, não garante que o produto esteja isento de falhas. A principal finalidade do teste é a identificação de possíveis defeitos em sistemas de software. Um defeito pode ser decorrentes de enganos, omissões ou mesmo intenções equivocadas por parte do desenvolvedor durante o processo de desenvolvimento (MCGREGOR; SYKES, 2001).

O padrão IEEE 610.12 (The Institute of Electrical and Eletronics Engineers, 1990) define e conceitua os principais termos utilizados no contexto de teste de software, a fim de que sejam evitadas interpretações incorretas e com objetivo de garantir uma padronização desta nomenclatura.

- **defeito (*fault*):** passo ou processo ou definição de dados incorretos;
- **engano (*mistake*):** ação humana que produz um resultado incorreto;
- **erro (*error*):** diferença entre o valor obtido e o valor esperado, a existência de um defeito pode ocasionar um erro; e
- **falha (*failure*):** resultado produzido na execução do programa seja diferente do resultado esperado.

A atividade de teste é dividida em fases e cada etapa do teste possui objetivos distintos. A finalidade deste modelo é garantir que cada etapa das fases de teste apresente características únicas que sirvam para guiar o processo de teste de software. Em geral, as fases podem ser definidas em (DELAMARO; MALDONADO; JINO, 2007):

- **Teste de unidade:** tem como objetivo testar as menores unidades que compõem o sistema (métodos, funções, procedimentos, classes, etc.). Espera-se encontrar com maior facilidade defeitos de programação, algoritmos incorretos ou mal implementados, estruturas de dados incorretas, limitando-se a lógica interna dentro dos limites da unidade;
- **Teste de integração:** o foco principal é verificar as estruturas de comunicação entre as unidades que compõem o sistema. As técnicas de projeto de casos de teste que exploram entradas e saídas são as mais utilizadas durante a fase de integração ([PRESSMAN, 2010](#)); e
- **Teste de sistema:** o objetivo é verificar se os requisitos satisfazem a especificação e se as funcionalidades do sistema foram implementadas corretamente, isto é, o sistema é testado como um todo procurando simular um ambiente de execução real.

2.1.1 Técnicas e Critérios de Teste

O critério de teste ideal capaz de avaliar o grau de correção de um software deveria ser capaz de avaliá-lo com todos os valores possíveis do seu domínio de entrada, tal abordagem é definida na literatura como teste exaustivo ([DELAMARO; MALDONADO; JINO, 2007](#)). Contudo, é uma abordagem dita como impraticável, pois na grande maioria dos casos, o domínio de entradas do software tende ao infinito ou, quando finito, grande o bastante para tornar tal atividade inviável.

Assim, as técnicas de teste de software existentes procuram estabelecer regras para definir subdomínios, com intuito de criar conjunto de casos de teste que satisfaçam os requisitos de teste pertencentes ao subdomínio abordado. Porém, nenhuma delas é suficiente para atestar de fato a qualidade dos testes produzidos. O cenário ideal é que as técnicas de teste sejam utilizadas de forma complementar, considerando a fase dos testes em que são aplicadas, de modo que seja possível extrair as vantagens de cada abordagem.

As técnicas e os critérios de teste são fundamentais para a seleção dos casos de teste, pois por meio destas abordagens é possível reduzir o número de casos de teste criados, além de garantir que partes específicas do software sejam exercitadas pelos testes. As técnicas de teste diferem entre si essencialmente pela origem da informação que é utilizada para seleção dos dados de teste.

As próximas subseções exemplificam brevemente as técnicas de teste funcional, estrutural e baseada em defeitos. A técnica baseada em defeitos é abordada com ênfase pois é objeto de estudo deste trabalho.

2.1.1.1 Teste Funcional

O teste funcional trata o sistema como uma caixa preta, devido ao fato de não se utilizar de detalhes de código fonte para a derivação dos requisitos de teste. Assim, os conjuntos de

casos de testes são elaborados apenas com base nos artefatos produzidos durante a elicitacão dos requisitos do software, assim como modelos produzidos para representar as especificacões das suas funcionalidades. O objetivo da técnica funcional de teste é avaliar as funções do sistema sem levar em consideracão detalhes de implementacão. Esse teste ignora os mecanismos internos do programa e foca apenas nas saídas geradas em resposta às entradas utilizadas e às condições de execucao (FABBRI; VINCENZI; MALDONADO, 2007).

Como todos os critérios da técnica funcional baseiam-se em artefatos produzidos durante o levantamento de requisitos e modelagem do sistema, é fundamental que os mesmos estejam corretos e sejam equivalentes entre si para que seja possível garantir a criação de casos de teste efetivos em detectar defeitos no produto avaliado.

Existem diversos critérios baseados na técnica de teste funcional, dentre eles, pode-se destacar o **Particionamento em classes de equivalência**, que define um conjunto de condições válidas ou inválidas para um subconjunto de entradas. Outro critério bastante difundido é a **Análise de valor limite**, que complementa o critério particionamento em classes, exigindo que os casos de teste exercitem os limites de cada classe de equivalência. Tal abordagem sugere que casos de teste que exploram condições próximas do limite das classes tem uma maior probabilidade de evidenciar possíveis defeitos (MYERS *et al.*, 2004).

Ainda que possuam algumas limitações, é importante o uso de critérios funcionais para que seja possível explorar determinados tipos de defeitos. A utilização do teste funcional possui vários pontos fortes, dentre os quais, Fabbri, Vincenzi e Maldonado (2007) destacam:

- **Reutilização de casos de teste:** por ser um critério baseado nas especificações, portanto independente da implementação, mesmo que haja alteracão no código os casos de teste podem permanecer os mesmos;
- **Independência de paradigma:** uma vez que o código fonte não é necessário, esses critérios podem ser aplicados em qualquer programa, seja ele procedimental ou orientado a objetos;
- **Qualidade de conjuntos de teste:** os critérios funcionais resultam um conjunto de teste com boa representacão de todo domínio de entrada das variáveis em teste; e
- **Tempo reduzido:** a derivacão dos casos de teste pode ser realizada paralelamente à implementacão, reduzindo o tempo do projeto.

2.1.1.2 Teste Estrutural

Na técnica Estrutural os critérios e requisitos de teste são derivados exclusivamente a partir das características do código-fonte do software em teste, permitindo que haja uma verificacão mais profunda do seu comportamento. Os casos de teste são essencialmente derivados a partir dos detalhes da estrutura interna do código-fonte do programa. Esse tipo de teste possibilita que

uma maior atenção seja dada aos pontos mais importantes do código (KOSCIANSKI; SOARES, 2007). A partir do código-fonte é definido um conjunto de elementos de software que devem ser executados para que se atinja a cobertura mínima para um determinado critério. A maioria dos critérios desta técnica utiliza uma representação do programa conhecida como grafo de fluxo de controle (GFC) (MALDONADO, 1991).

A técnica estrutural tem como objetivo ser aplicada a pequenas partes do código, como sub-rotinas, ou operações relacionadas a um objeto. Assim, o testador pode analisar o código antes de iniciar os testes e buscar dados que possivelmente possam ser tratados de forma inesperada, além de poder garantir que o programa foi liberado tendo seus comandos executados pelo menos uma vez por pelo menos um caso de teste (BARBOSA *et al.*, 2007).

Os critérios estruturais aplicados em programas sequenciais são, em geral, classificados em:

- **Critérios baseados na complexidade:** utilizam informações sobre a complexidade do programa para derivar os requisitos de teste. Ex.: o critério de McCabe (teste de caminho básico) , que utiliza a complexidade ciclomática do GFC para derivar os requisitos de teste (MCCABE, 1976);
- **Critérios baseados em fluxo de controle:** utilizam características de controle da execução do programa (comandos, desvios de execução) para determinar os requisitos de teste. Alguns critérios dessa técnica são: Todos-Nós, Todas-Arestas (ou Todos-Arcos) e Todos-Caminhos;
- **Critérios baseados em fluxo de dados:** é utilizada a análise de fluxo de dados como fonte de informação para derivar os requisitos de teste. Rapps e Weyuker (1985) definiram uma família de critérios de fluxo de dados, sendo que o critério Todos-Usos é o mais conhecido. Tendo como base esta abordagem, Maldonado (1991) definiu uma família de critérios, chamada Potenciais-Usos, que avalia associações independentemente da ocorrência explícita de uma referência ou uso a uma determinada definição. Os critérios que fazem parte da família de critérios Potenciais-Usos são: Todos-Potenciais-Usos, Todos-Potenciais-Usos-Du e Todos-Potenciais-Du-Caminhos.

2.1.1.3 Teste Baseado em Defeitos

Nesta técnica são utilizados defeitos típicos do processo de implementação de um software para derivar os requisitos de teste. A técnica Baseada em Defeitos usa de forma direta um modelo de defeitos para criar hipóteses sobre possíveis falhas que possam estar presentes no programa em teste (PRESSMAN, 2010), bem como com objetivo de criar ou avaliar conjuntos de testes com base na sua eficácia em revelar os defeitos hipoteticamente modelados.

Esses defeitos são constituídos de enganos típicos cometidos pelos desenvolvedores de software, e por meio destes defeitos são derivados os requisitos de teste. No entanto, para não inviabilizar a aplicabilidade da técnica, somente defeitos considerados plausíveis devem ser inseridos na implementação a ser avaliada (YOUNG, 2008).

Se o modelo de defeitos utilizado representar o conhecimento sobre os principais enganos cometidos, então essa abordagem tende a encontrar uma grande quantidade de defeito com um baixo esforço. A finalidade da técnica é testar a correção de uma dada implementação P com relação à sua vizinhança $\Phi(P)$. A correção de P se dá por meio de um processo em que prova-se que nenhuma implementação contida em $\Phi(P)$, não equivalente a P , é correta (DELAMARO *et al.*, 2007). Esse processo é feito projetando casos de testes capazes de mostrar o comportamento anômalo de $\Phi(P)$ em relação a P .

Contudo, essa não é uma abordagem prática, pelo simples fato de $\Phi(P)$ tender ao infinito. Para possibilitar a aplicação da técnica, é necessário restringir $\Phi(P)$ para que seja possível então atingir uma correção relativa de P , analisando apenas um subconjunto de sua vizinhança. Os principais critérios dessa técnica são: injeção de falhas e o Teste de Mutação, que é o foco deste trabalho e é detalhado a seguir.

2.2 Visão sobre o Teste de Mutação

O Teste de Mutação é um critério da técnica de teste baseado em defeitos, que tem por finalidade medir o grau de adequação de um conjunto de casos de teste (DEMILLO; LIPTON; SAYWARD, 1978; DEMILLO, 1980). Para atingir este objetivo, o Teste de Mutação utiliza informações sobre os defeitos cometidos com maior frequência durante o processo de desenvolvimento de software para derivar os requisitos de teste. O foco deste critério está nos defeitos que o desenvolvedor ou projetista de software está propenso a cometer durante o processo de produção do software e nas abordagens que são utilizadas para detectar tais defeitos.

Segundo DeMillo, Lipton e Sayward (1978), a definição do critério Teste de Mutação é baseada em duas hipóteses:

- **Hipótese do Programador Competente:** assume que programas desenvolvidos por programadores competentes estão corretos ou bem próximos do correto; e
- **Hipótese do Efeito de Acoplamento:** assume que um conjunto de dados de teste capaz de evidenciar defeitos simples em um programa, também é capaz de detectar defeitos complexos, devido ao fato de estudos empíricos terem evidenciado que falhas complexas produzidas na execução de um programa normalmente estarem relacionados a defeitos simples.

Considerando tais hipóteses, seguindo processo para aplicação do Teste de Mutação, o testador deve aplicar um conjunto de casos de teste T contra o programa P que deseja-se verificar, com objetivo de verificar se o resultado obtido pelos casos de teste é igual ao resultado esperado. Caso este resultado seja diferente, então o programa possui um defeito que deve ser corrigido para que possa dar continuidade ao Teste de Mutação.

A partir do programa P é gerado um conjunto de programas ($M_1, M_2, M_3, M_4, \dots, M_n$) denominados mutantes, conforme representado na Figura 1. Tais programas mutantes são gerados com objetivo de modelar enganos comuns que podem acontecer durante o processo de desenvolvimento de software.

Figura 1 – Processo de geração de mutantes.

Para geração dos programas mutantes, é necessária a utilização de operadores de mutação. Entende-se por operadores de mutação regras responsáveis por definirem alterações que devem ser aplicadas ao programa original P , a fim de satisfazer duas condições: (i) induzir mudanças sintáticas simples com base nos enganos típicos que os desenvolvedores de software estão propensos a cometer durante o processo de desenvolvimento ou (ii) forçar objetivos de teste específicos.

A Figura 2 apresenta o trecho de código da versão original de um programa, e um mutante gerado a partir da aplicação do operador de mutação *Move Brace Up and Down* (SMVB).

Não há uma maneira específica para definição do conjunto de operadores de mutação, normalmente os operadores são determinados com base em um modelo de falhas que é definido essencialmente pela experiência do uso da linguagem, assim como com base em observações a respeito dos enganos cometidos com maior frequência pelos desenvolvedores durante a sua utilização. Nesse sentido, cada linguagem de programação possui um conjunto único de operadores de mutação. Na literatura existe uma variedade de operadores de mutação definidos para diversas linguagens como: Fortran (DEMILLO; OFFUTT, 1991), C (AGRAWAL *et al.*, 1989;

Figura 2 – Programa original P e Mutantes M_1 e M_2 gerados pelo uso do operador SMVB.

DELAMARO; MALDONADO; MATHUR, 2001), Java (KIM; CLARK; MCDERMID, 1999; MA; OFFUTT; KWON, 2005).

Após garantir que o programa P atende às saídas esperadas para o conjunto de casos de teste, o mesmo conjunto de teste T aplicado em P é executado contra os mutantes gerados, podendo apresentar duas saídas distintas: (a) se a saída do mutante M_i for diferente de P , ou (b) se a saída de M_i for idêntica a P , para qualquer caso de teste executado.

No caso da saída (a) o mutante é definido como “morto”, ou seja, dentro do conjunto de casos de teste T , existe pelo menos um caso de teste que foi capaz de distinguir o comportamento do mutante M_i em relação ao programa original P . No caso da saída (b) o programa mutante é definido como “vivo”, ou seja, não há no conjunto de casos de teste T um caso de teste capaz de evidenciar a perturbação de código inserida em M_i . Nos casos em que não é possível identificar um caso de teste capaz de distinguir M_i de P , então M_i é dito como equivalente a P .

O processo geral envolvido no Teste de Mutação é representado na Figura 3, sendo normalmente dividido nos seguintes passos:

- geração dos mutantes;
- execução do programa em teste;
- execução dos mutantes; e
- análise dos mutantes.

O processo se encerra quando todos os mutantes que foram gerados são mortos pelo conjunto de casos de teste, ou há a identificação de mutantes que não podem ser mortos, no caso da existência de mutantes equivalentes.

Figura 3 – Processo genérico do Teste de Mutação. Adaptado de (OFFUTT; UNTCH, 2001).

Um problema inerente ao Teste de Mutação é a possível existência de programas mutantes equivalentes ao programa original. A equivalência entre programas é uma questão indecidível e requer a intervenção manual do testador. Contudo, existem estudos que trabalham no sentido de minimizar este problema. Orzeszyna (2011) mostra por meio de uma revisão sistemática que existem diversos trabalhos que investigam alternativas para solucionar este problema, entre as alternativas propostas estão: (i) técnicas para detecção mutantes equivalentes, (ii) técnicas para evitar a geração de mutantes equivalentes, (iii) técnicas para a sugestão de mutantes equivalentes.

Um fator importante no Teste de Mutação é o **escore de mutação**, que fornece uma medida de confiança do nível de adequação do conjunto de casos de teste utilizado durante a aplicação do critério. Para tal, é necessária a realização de um cálculo que relaciona o número de mutantes gerados, com o número de mutantes mortos.

O cálculo do escore de mutação é dado da seguinte forma:

$$ms(P, T) = \frac{DM(P, T)}{M(P) - EM(P)}$$

- $DM(P, T)$ = número de mutantes mortos pelos conjunto de casos de teste em T .
- $M(P)$ = número total de mutantes gerados.

- $EM(P)$ = número de mutantes gerados equivalentes a P .

O escore de mutação é uma medida que varia entre 0 e 1. Quanto maior o escore de mutação, melhor avaliado é o conjunto de casos de teste. Contudo, atingir 100% do escore de mutação exige lidar com o problema da equivalência de mutantes, que muitas vezes se torna um grande desafio devido ao grande número de mutantes gerados, mesmo para programas relativamente simples.

Com objetivo de reduzir o número de mutantes gerados, foram definidos critérios alternativos que usam abordagens específicas para definir um subconjunto de mutantes dentre os mutantes gerados, sem que a eficácia do Teste de Mutação seja comprometida. [Jia e Harman \(2011\)](#) apresentaram um *survey* que apresenta diversas abordagens para a redução de custo na aplicação do Teste de Mutação. A seção seguinte mostra uma visão geral destas informações, dando destaque para abordagens que objetivem redução de custo na aplicação do critério.

2.3 Técnicas para Redução de Custo

Apesar de ser considerado um bom critério para a detecção de defeitos, ainda não há consenso sobre uma abordagem definitiva para solucionar o problema relacionado ao custo de aplicação do Teste de Mutação ([HARROLD, 2000](#)). Assim, diversas abordagens foram propostas para solucionar este problema, cada uma explorando características específicas do problema. [Offutt e Untch \(2001\)](#) propuseram uma classificação em três tipos para as abordagens desenvolvidas para redução do custo de aplicação do critério. São elas:

- *do fewer*: tentar encontrar formas de reduzir os mutantes que precisam ser executados, sem comprometer a aplicabilidade do critério.
- *do faster*: procurar maneiras de acelerar a execução dos mutantes.
- *do smarter*: dividir o custo computacional, evitar a execução completa de mutantes ou guardar informações sobre o estado da execução para possibilitar execuções inteligentes.

Observando esta classificação, a Figura 4 mostra uma linha do tempo que contempla o desenvolvimento de trabalhos que seguem estas estratégias dividindo-os em duas categorias: 1) Redução de mutantes (*do fewer*) e 2) Redução do custo de aplicação (*do faster, do smarter*).

Figura 4 – Visão cronológica do desenvolvimento de técnicas para viabilizar a aplicação do Teste de Mutação. Adaptado de (JIA; HARMAN, 2011).

As subseções seguintes fazem um resumo sobre os principais trabalhos apresentados com objetivo de redução do custo na aplicação do Teste de Mutação.

2.3.1 Redução de mutantes

Um dos maiores problemas que limitam a aplicação do Teste de Mutação está relacionado com o tempo gasto na execução do conjunto de mutantes contra o conjunto de casos de teste. Assim, reduzir o número de mutantes a serem executados, sem comprometer a efetividade do critério, tornou-se um problema de pesquisa bastante pesquisado.

Dentre as abordagens criadas para selecionar um conjunto de mutantes, *Mutant Sampling* propõe escolher aleatoriamente um subconjunto do conjunto de mutantes (ACREE, 1980; BUDD, 1980). Nessa abordagem todos os mutantes são gerados e então um percentual dos mutantes gerados é selecionado aleatoriamente para execução e os demais mutantes são descartados. Estudos empíricos sustentam que a utilização desta abordagem utilizando amostras de 10% de mutantes são apenas 16% menos efetivas que a utilização de todo o conjunto de mutantes quando analisado o escore de mutação.

Hussain (2008) propôs uma abordagem de *Mutant Clustering*. Essa técnica seleciona um subconjunto de mutantes utilizando algoritmos de *clustering*. Os mutantes são classificados em *clusters* com base nos casos de teste. Isto é, os mutantes pertencentes ao mesmo *cluster* tendem a ser mortos por um conjunto similar de casos de teste. Então apenas um pequeno número de mutantes em cada *cluster* é selecionado para a aplicação do Teste de Mutação. Outra abordagem relacionada a *Mutant Clustering* foi proposta por Ji *et al.* (2009), neste trabalho são utilizadas técnicas de redução de domínio para evitar a execução de todos os mutantes.

A redução do número de mutantes gerados também pode ser obtida por meio da redução do número de operadores de mutação utilizados durante a aplicação do Teste de Mutação. Essa é a ideia da técnica *Selective Mutation*, procura-se encontrar o menor conjunto de operadores de mutação que gere um subconjunto de mutantes de tal modo que a eficiência do critério não seja afetada. Essa ideia foi proposta inicialmente por Mathur (1991) como *Constrained Mutation* e posteriormente Offutt, Rothermel e Zapf (1993) a estenderam, chamando-a de *Selective Mutation*.

Com base nessas experiências, Barbosa, Maldonado e Vincenzi (2001) definiram um *guideline* para a seleção de um conjunto suficiente de operadores de mutação dentro do universo de todos os operadores de mutação existentes. Namin e Andrews (2006) formularam *Selective Mutation* como um problema estatístico, em que foram aplicadas abordagens de estatística linear para identificar um subconjunto de operadores.

High Order Mutation consiste na criação de mutantes que sejam capazes de simular um maior número de defeitos. De acordo com Jia e Harman (2008a), mutantes de primeira ordem, são em geral composto de defeitos considerados triviais, isso acaba levando a criação de casos de teste não muito efetivos, pois os mutantes são mortos com relativa facilidade.

Em contraste, mutantes criados por *High Order Mutation* contém dois ou mais defeitos. A ideia fundamental é gerar mutantes mais difíceis de se matar em relação aos mutantes modelados com um único defeito. Assim, mutantes são classificados de duas formas: mutantes modelados com um único defeito são denominados de *First Order Mutation* e mutantes que contenham dois ou mais defeitos são denominados de *High Order Mutation*.

Experimentos com objetivo de gerar mutantes *High Order* a partir de mutantes *First Order* foram conduzidos por Polo, Piattini e Garcia-Rodriguez (2008) e os resultados indicaram que a utilização de mutantes de *Second Order* tendem a reduzir o custo da aplicação do Teste de Mutação em 50%, mantendo a efetividade do teste.

2.3.2 Redução do custo de aplicação

Além de reduzir o número de mutantes gerados, também é possível reduzir o custo computacional da aplicação da técnica otimizando o processo de execução dos mutantes. Essa seção introduz três tipos de abordagens que são consideradas na literatura para otimizar o processo de execução: (i) tipos de mutação, (ii) técnicas de otimização e (iii) plataformas avançadas de execução.

O Teste de Mutação pode ser agrupado em três diferentes tipos: *Strong*, *Weak* e *Firm mutation*. Em sua essência eles basicamente variam na forma utilizada para avaliar se um determinado mutante foi morto ou não durante o processo de execução.

Strong mutation está relacionada ao conceito clássico de aplicação do Teste de Mutação. Ou seja, é formulada com base nas definições propostas inicialmente por DeMillo, Lipton e Sayward (1978). Em *Strong Mutation*, dado um programa P , um mutante M de P é dito como

morto se e somente se a saída produzida por M for diferente da saída produzida por P .

Para otimizar a execução de *Strong Mutation*, [Howden \(1982\)](#) propôs *Weak Mutation*. Em *Weak Mutation*, um programa P é assumido como formado por um conjunto de componentes. É pressuposto então que um mutante M é formado por uma mudança em um dado componente. O mutante é dito como morto se a execução do componente diferir da execução do componente correspondente no programa P . Como resultado em *Weak Mutation*, diferentemente de checar o mutantes após a execução completa do programa, o mutante deve ser verificado imediatamente após o ponto de mutação, ou seja, no componente mutado.

A vantagem do *Weak Mutation* é que cada mutante não requer uma execução completa. Além da vantagem de não haver necessidade de se gerar todos os mutantes. Contudo, como diferentes componentes do programa original podem gerar diferentes saídas, casos de testes projetados para *Weak Mutation* tendem a ser menos efetivos em relação a *Strong Mutation*. Assim, *Weak Mutation* sacrifica um pouco da efetividade na criação dos casos de teste a fim de melhorar a performance na execução. Isso levanta uma série de discussões acerca do *trade-off* entre executar mais rápido e testar melhor.

[Durelli, Offutt e Delamaro \(2012\)](#) propuseram uma abordagem de teste de software integrada a máquinas virtuais de linguagens de programação para apoiar a aplicação do Teste de Mutação, utilizando a variante *Weak Mutation* e os experimentos conduzidos mostraram que foi possível atingir um desempenho 95% superior em relação a ferramentas que utilizam a abordagem tradicional do Teste de Mutação (*Strong Mutation*).

Firm Mutation foi proposta inicialmente por [Woodward e Halewood \(1988\)](#). A ideia de *Firm Mutation* é superar as desvantagens de *Weak Mutation* e *Strong Mutation* provendo um meio termo de possibilidades. Isto é, *Firm Mutation* considera tanto o estado da execução, logo após o ponto de mutação (*Weak Mutation*), quanto a saída final produzida (*Strong Mutation*). [Jackson e Woodward \(2001\)](#) propuseram uma abordagem para execução paralela de *Firm Mutation* para programas em Java, contudo, na época não existiam ferramentas públicas que possibilitassem a aplicação de *Firm Mutation*.

Dentre os trabalhos que abordam técnicas de otimização, as técnicas baseadas em interpretação foram as mais utilizadas durante a primeira geração de ferramentas desenvolvidas para aplicação do Teste de Mutação. Nessa abordagem, o resultado de um mutante é interpretado diretamente do código fonte. Para otimizar essa abordagem, [King e Offutt \(1991\)](#) propuseram traduzir o programa original em teste para uma forma intermediária. Essa abordagem permite flexibilidade e é suficientemente eficiente para mutação em pequenos programas. Contudo, devido à natureza dos interpretadores, a técnica acaba se tornando mais lenta quando o tamanho do programa em testes aumenta.

Técnicas baseadas em compilação são as abordagens mais comuns para possibilitar a mutação em programas ([DELAMARO, 1993](#); [DELAMARO; MALDONADO, 1996a](#)). Nessa

técnica, cada mutante é primeiramente compilado em um programa executável. O mutante compilado é então executado contra todo o conjunto de casos de teste.

Comparado com técnicas de interpretação, essa abordagem é muito mais efetiva por conta da execução compilada do código binário ser bem mais rápida que a interpretada. Contudo, há ainda uma limitação de velocidade conhecida como gargalo de compilação, devido ao grande tempo gasto para a compilação dos mutantes.

Uma abordagem de geração de mutantes denominada de *Mutant Schemata Generation* foi projetada por [Untch \(1992\)](#) com objetivo de reduzir o custo de técnicas baseadas em compilação. Ao invés de compilar cada mutante separadamente, a técnica *Mutant Schemata* gera um meta programa capaz de representar todo o conjunto de mutantes. Assim, para executar todos os programas contra o conjunto de casos de teste, somente o metaprograma deve ser compilado. O custo dessa técnica é composto do custo único de compilação do metaprograma e do custo de execução. Como o metaprograma é compilado, então o desempenho da abordagem é mais eficiente que abordagens interpretadas.

Os trabalhos mais recentes relacionados a técnicas para redução de custo em compilação trabalham com *Bytecode Translation* ([MA; OFFUTT; KWON, 2005](#)). Nessa abordagem os mutantes são gerados a partir do objeto compilado do programa original. Como resultado, os mutantes gerados por *bytecode* podem ser executados diretamente sem o processo de compilação. Além de salvar o tempo de execução, *Bytecode Translation* também pode lidar com programas que não tem o código fonte disponível.

Essa técnica foi adotada na linguagem de programação Java. Contudo, nem todas as linguagens de programação possuem uma maneira simples de manipular objetos de código intermediários. Existem também limitações para aplicação de *Bytecode translation* em Java, como o fato de não ser possível representar todos os operadores de mutação a nível de *bytecode*.

Com relação a plataformas avançadas para suportar a aplicação do Teste de Mutação, existem diversos trabalhos na literatura que tratam sobre a execução paralela de mutantes. É possível destacar duas principais linhas de investigação quando se aborda execução paralela de mutantes: execução de mutantes em máquinas *single instruction multiple data* (SIMD) e execução de mutantes em máquinas *multiple instruction multiple data* (MIMD).

[Mathur e Krauser \(1988a\)](#) propuseram executar diversos mutantes em uma máquina vetorial. Especificamente, os algoritmos descritos neste trabalho foram projetados para executarem sobre mutantes obtidos com o uso do operador de mutação de *Scalar Variable Replacment* (SVR), devido ao fato deste operador gerar um grande número de mutantes.

A ideia básica dos autores era mapear simultaneamente a execução dos mutantes SVR para uma sequência de instruções em um vetor, que pode ser executado em paralelo por uma máquina de processamento vetorial. Posteriormente, a ideia foi estendida por [Krauser, Mathur e Rego \(1991\)](#) para possibilitar a aplicação dos demais operadores de mutação. Os resultados

experimentais obtidos evidenciaram que o maior gargalo para aplicação da técnica seriam problemas relacionados ao tempo gasto para compilação dos mutantes, problema este que foi posteriormente resolvido conforme anteriormente descrito.

A segunda linha de pesquisa foi baseada em máquinas MIMD. O primeiro estudo sobre mutação aplicado em máquinas MIMD foi o trabalho proposto por [Choi e Mathur \(1993\)](#). Os autores apresentaram uma adaptação da ferramenta *Mothra*, dividindo-a em módulos, de forma a possibilitar a aplicação do Teste de Mutação em paralelo. Nesse estudo foi observado que o tempo de comunicação entre os módulos paralelos da ferramenta possuía um grande impacto na aplicabilidade da técnica.

Outro importante trabalho proposto por [Offutt et al. \(1992\)](#), apresentou outra variação da ferramenta *Mothra*, desta vez, possibilitando a execução de mutantes em máquinas *Hypercube*. Nesta nova abordagem os autores olharam três diferentes algoritmos estáticos de distribuição de dados que dividiam o conjunto de mutantes: na sua ordem original de geração, distribuíam de maneira aleatória e por fim a distribuição uniforme de mutantes levando em consideração os operadores de mutação.

Também foram analisadas diferentes configurações sobre o número de nós da máquina *Hypercube* utilizada. Os autores concluíram que essa abordagem atingiria um *speedup* significativo somente com aplicações maiores, apontando que técnicas de paralelização seriam eficientes apenas para aplicação em sistemas de grande escala.

Com relação aos algoritmos de distribuição os autores determinaram que o algoritmo mais eficiente foi o algoritmo de distribuição de mutantes ordenados por tipo de operador, pelo fato de ter sido o algoritmo que atingiu os melhores resultados para o tempo de execução dos programas. Uma outra importante conclusão deste trabalho para a época, foi notar que a comunicação entre os nós do *Hypercube* gerava um gargalo na aplicação da técnica contribuindo para perda de desempenho. Esse fator levou à conclusão de que algoritmos de distribuição dinâmicos (como algoritmos sob demanda) que necessitam de maior comunicação entre os nós levariam a resultados inferiores em comparação aos algoritmos estáticos avaliados.

Posteriormente, com o avanço das tecnologia de processamento, [Mateo e Usaola \(2013\)](#) decidiram voltar a abordar a execução paralela como forma de melhorar a aplicabilidade do Teste de Mutação. O artigo aborda cinco algoritmos de balanceamento de carga, que tem como principal objetivo dividir a execução de atividades realizadas no Teste de Mutação a fim de melhorar a eficiência de aplicação da técnica reduzindo o tempo de execução e maximizando o uso dos recursos computacionais. Este artigo foi utilizado como base para o desenvolvimento deste trabalho e os capítulos futuros possuem informações mais aprofundadas acerca dos algoritmos utilizados bem como uma discussão sobre resultados obtidos.

2.4 Ferramentas de Apoio

Um ponto fundamental para o apoio à aplicação do Teste de Mutação é a utilização de ferramentas que deem suporte à sua utilização. Sem a utilização de uma ferramenta que auxilie nas etapas para a aplicação deste critério, sua aplicação pode ser comprometida, devido a uma série de etapas envolvidas durante o processo.

Diversas ferramentas foram desenvolvidas nesse sentido, principalmente durante os primeiros anos de utilização da técnica (DELAMARO *et al.*, 2007). O *survey* conduzido por Jia e Harman (2011) apresenta um catalogo das principais ferramentas utilizadas em âmbito acadêmico para apoiar a aplicação do Teste de Mutação.

O Quadro 1 resume essas informações mostrando o nome das ferramentas, bem como a linguagem de programação alvo, além trazer informações sobre novas ferramentas. É possível notar as linguagens C e Java receberam maior atenção por parte dos pesquisadores, com um maior número de ferramentas para apoiar a aplicação do critério. Dentre elas, este trabalho destaca a ferramenta *Program Testing Using Mutants (Proteum)* (DELAMARO; MALDONADO, 1996b) que apoia a aplicação do Teste de Mutação para programas em C.

Quadro 1 – Lista de ferramentas que apoiam a aplicação do Teste de Mutação

Nome	Linguagem	Referência
AjMutator	AspectJ	(DELAMARE; BAUDRY; TRAON, 2009)
Bacterio	Java	(MATEO; USAOLA; OFFUTT, 2013)
CREAM	C#	(DEREZIŃSKA; SZUSTEK, 2008)
ExMAn	C/Java	(BRADBURY; CORDY; DINGEL, 2006)
JavaMut	Java	(CHEVALLEY; THÉVENOD-FOSSE, 2002)
Javalanche	Java	(SCHULER; ZELLER, 2009)
Jester	Java	(MOORE, 2005)
Judy	Java	(MADEYSKI; RADYK, 2010)
Jumble	Java	(IRVINE <i>et al.</i> , 2007)
MILU	C	(JIA; HARMAN, 2008b)
Mothra	Fortran	(DEMILLO <i>et al.</i> , 1988)
muJava	Java	(MA; OFFUTT; KWON, 2005)
MuTMuT	Java	(GLIGORIC <i>et al.</i> , 2013)
MutPy	Python	(HAIAS, 2014)
Paraμ	Java	(MADIRAJU; NAMIN, 2011)
PIMS	Fortran	(BUDD <i>et al.</i> , 1978)
Pitest	Java	(COLES, 2015)
Proteum	C	(DELAMARO; MALDONADO, 1996b)
Proteum/AJ	AspectJ	(FERRARI <i>et al.</i> , 2011)
Smutant	Smaltalk	(GLIGORIC; BADAME; JOHNSON, 2011)
Testooj	Java	(POLO; TENDERO; PIATTINI, 2007)

A atividade de teste na *Proteum* é conduzida por meio de sessões de teste. Uma sessão de testes é caracterizada por uma base de dados composta por um conjunto de casos de testes e

um conjunto de mutantes (DELAMARO; MALDONADO; VINCENZI, 2001). Na ferramenta *Proteum* pode-se especificar dois tipos distintos de sessão de testes:

- **Modo Teste:** este tipo de sessão de teste é considerado o método convencional, nessa abordagem todos os mutantes são executados contra os casos de teste somente enquanto encontram-se vivos;
- **Modo Research:** neste tipo de sessão de teste todos os mutantes são executados contra todos os casos de teste independentemente do estado do mutante. Este tipo de sessão é útil principalmente para a realização de experimentos e para testar a efetividade individual de cada caso de teste, contudo, torna o processo do Teste de Mutação mais oneroso.

A ferramenta *Proteum* é composta por uma série de módulos que atuam sobre a base de dados. Cada módulo é responsável por realizar uma atividade ligada ao Teste de Mutação. A seguir são descritos os módulos existentes na ferramenta *Proteum* e suas respectivas funções:

- **Test-new:** cria uma sessão de testes, ou seja, cria todos os arquivos necessários para a aplicação do teste de mutação para o programa em teste alvo;
- **Tcase:** módulo utilizado para gerenciar o conjunto de casos de teste utilizados em uma sessão de teste na ferramenta *Proteum*, possibilitando: adição, deleção, seleção dos casos de teste a serem executados durante a sessão;
- **Muta-gen:** módulo responsável pela geração dos mutantes a serem utilizados na sessão de testes;
- **Exemuta:** constrói e executa os mutantes a partir de descritores de mutação; também é utilizado para selecionar um determinado grupo de mutantes;
- **Muta:** utilizado para o gerenciamento do conjunto de mutantes existente na sessão de teste. Permite a seleção de um sub-conjunto de mutantes, além de possibilitar a alteração do *status* dos mutantes individualmente;
- **Muta-view:** permite a visualização e análise dos mutantes criados/executados na sessão de testes; e
- **Report:** é utilizado para a produção de dois tipos de relatórios: relatórios sobre os casos de teste utilizados e relatórios sobre os mutantes gerados/executados.

Cada um desses módulos pode ser executado de maneira independente a partir do console do sistema operacional, ou dentro de scripts. Essa característica permite que sessões completas de teste possam ser programadas e executadas em modo *batch*. A ferramenta possui uma interface

gráfica, mas funciona apenas como uma Shell que invoca cada um dos programas que compõem a ferramenta.

Dentre as principais características presentes na ferramenta *Proteum*, pode se destacar (DELAMARO; MALDONADO, 1996b):

- Abordagem compilada em vez de utilizar uma abordagem interpretada, a principal vantagem dessa abordagem é que a execução de um programa compilado tende a ser muito mais rápida que a execução de um programa interpretado. Como em geral no Teste de Mutação existe um número grande de mutantes, o tempo salvo na condução dos testes é grande;
- Utilização de informações de fluxo de controle para evitar a execução de certos mutantes. Durante a execução dos mutantes, a ferramenta verifica se determinado caso de teste executam o ponto do programa que sofreu a mutação. Caso não o execute, o caso de teste não precisa ser utilizado, pois se sabe que o resultado da execução será o mesmo do programa original;
- Facilidade de importação de conjuntos de teste fornecidos à ferramenta, que não precisam ser inseridos interativamente um a um pelo testador. A ferramenta possibilita a adição de casos de teste de maneira manual, utilizando sessões de teste anteriores, ou importando casos de teste da ferramenta *POKE TOOL* (CHAIM, 1991);
- Dois modos de execução dos mutantes: teste e pesquisa. O primeiro apresenta o comportamento original do Teste de Mutação, que é a interrupção da execução dos casos de teste quando se encontra um caso de teste que distingue o mutante do programa original. No modo pesquisa, todos os casos de teste são aplicados contra todos os mutantes. Essa atividade onera o tempo necessário para aplicação do Teste de Mutação, contudo, permite a coleta de informações sobre a efetividade de um dado conjunto de testes ou sobre determinados mutantes. Tais informações se tornam importantes principalmente no ambiente de pesquisa científica;
- Operações para habilitar/desabilitar casos de testes de maneira flexível, garantindo assim que o testador possa experimentar diferentes combinações de casos de teste sem necessitar alterar o conjunto de casos de teste; e
- De maneira semelhante, também está presente a possibilidade de selecionar os mutantes utilizados durante o teste, garantindo por exemplo, que o testador possa avaliar o efeito do conjunto de casos de teste para um grupo específico de mutantes.

2.5 Considerações Finais

Nesse capítulo foram apresentados e discutidos conceitos fundamentais sobre teste de software. Foram abordados conceitos relacionados a técnicas e critérios de teste de software,

dando ênfase especificamente ao Teste de Mutação que é objeto de estudo deste trabalho.

Em suma, o Teste de Mutação é uma técnica altamente eficaz para evidenciar defeitos em um produto de software e é uma ótima alternativa para medir o nível de adequação de um conjunto de casos de teste, contudo, possui grandes desafios que precisam ser superados. Entre as principais dificuldades está o grande número de mutantes gerados que acaba se tornando um empecilho para a aplicação da técnica em ambientes reais.

No capítulo seguinte são apresentados conceitos sobre programação concorrente. O principal objetivo desta revisão bibliográfica é fornecer subsídios para os próximos capítulos desta dissertação que tratam da apresentação e avaliação de uma arquitetura que dá suporte à execução do Teste de Mutação em paralelo.

FUNDAMENTOS SOBRE COMPUTAÇÃO CONCORRENTE

Este capítulo apresenta uma visão geral sobre programação concorrente, abordando seus principais conceitos. Inicialmente, serão destacados conceitos básicos, aplicações, vantagens e desvantagens do modelo de software concorrente. Em seguida, será apresentada a classificação dos tipos de arquiteturas de computadores paralelos existentes, segundo a definição clássica proposta por [Flynn \(1966\)](#). Posteriormente serão apresentados aspectos de softwares concorrentes desenvolvidos para máquinas com suporte a múltiplas instruções, destacando os seus principais desafios e por fim serão mostradas as principais métricas para avaliação de desempenho de software.

3.1 Fundamentos de Programação Concorrente

Primordialmente, o computador sempre foi tido como uma máquina capaz de executar uma sequência de instruções e, por consequência, a grande maioria das linguagens de programação sempre impôs que o programador especificasse um algoritmo que definisse uma sequência de passos para a realização de uma determinada atividade ([STALLINGS, 2010](#)). Assim, os processadores executam programas por meio de uma pilha de instruções sequenciais de máquina, neste cenário cada instrução é executada como uma sequência de operações.

Com o passar dos anos, o resultado dos avanços tecnológicos na área de hardware fez com que a computação distribuída passasse a fazer cada vez mais parte dos sistemas atuais devido à grande demanda da indústria por computadores com alto poder de desempenho. Com isso, a programação concorrente tornou-se cada vez mais popular no desenvolvimento de softwares modernos. Esse paradigma de desenvolvimento é essencial para a construção de aplicações capazes de reduzir o tempo computacional e tem sido aplicado com sucesso em vários domínios, tais como: previsão de tempo, dinâmica de fluídos, processamento de imagens, química quântica

entre outros (GRAMA, 2003).

A programação concorrente tem como objetivo o desenvolvimento de programas que executem tarefas de maneira paralela. Segundo Andrews (2000) um programa concorrente contém dois ou mais processo ou *threads* que trabalham juntos para realizar uma determinada tarefa. Esse estilo de programação se difere da programação puramente sequencial por permitir aperfeiçoar o desempenho das aplicações explorando melhor os recursos disponíveis.

A ideia básica por trás da programação concorrente é, portanto, que determinadas partes da computação sejam subdivididas em processos concorrentes, sendo cada processo responsável pela resolução de uma parte de um problema maior. Essa divisão em processos concorrentes é obtida por meio de recursos de software adicionais, esses responsáveis, por exemplo, pela ativação e finalização de processos concorrentes e que não estão disponíveis em programas sequenciais (PACHECO, 2011).

Essa abordagem pode viabilizar a solução de problemas em que é exigido um melhor aproveitamento dos recursos de hardware disponíveis, como, por exemplo, quando há a necessidade de realizar uma grande quantidade de processamento. Neste tipo de cenário, a execução sequencial de atividades acaba não aproveitando completamente os recursos oferecidos, culminando com um desempenho menos satisfatório. Assim, a programação concorrente pode apresentar grandes vantagens em relação a execução sequencial.

3.2 Arquitetura de Computadores Paralelos

O paralelismo é um subtipo da programação concorrente e ocorre em arquiteturas que disponibilizam mais de um recurso de processamento, ou seja, quando não há concorrência pelos recursos disponíveis e os processos executam ao mesmo tempo, por exemplo, em núcleos distintos de uma arquitetura *multicore* (PACHECO, 2011). Contudo, para se desenvolver programas concorrentes de maneira eficiente, é necessário que o desenvolvedor possua conhecimento acerca da arquitetura de software com que está trabalhando.

A taxonomia clássica proposta por Flynn (1966) classifica arquiteturas de computadores paralelos conforme o grau de paralelismo existente no processamento de suas instruções e no fluxo de dados que podem ser gerados simultaneamente. Um fluxo de instruções é definido como uma sequência de instruções realizadas por um computador e o fluxo de dados como uma sequência que descreve o fluxo de informação e as transformações que são aplicadas à medida que os dados se movimentam da entrada para a saída (FLYNN, 1966).

Com base nessas definições, Flynn (1966) definiu quatro categorias de arquiteturas para computadores paralelos:

- SISD (*Single Instruction Stream, Single Data Stream*) - máquinas que possuem fluxo único de instruções e fluxo único de dados, podendo executar uma única sequência de instruções,

bem como uma sequência de dados por vez. A grande maioria dos computadores com apenas um único processador se enquadra nesta categoria ([PACHECO, 2011](#)). Nesta arquitetura não existe paralelismo em nível de instrução ou dados, contudo, é possível encontrar certo nível de paralelismo, graças à utilização de técnicas de *pipeline*;

- SIMD (*Single Instruction Stream, Multiple Data Stream*) - máquinas que possuem fluxo único de instruções e fluxo múltiplo de dados. Neste modelo, uma única sequência de instruções de máquinas pode controlar simultaneamente a execução de múltiplas sequências de dados. Cada elemento de processamento pode possuir um conjunto de dados associados a ele, permitindo assim que cada instrução executada possa trabalhar com um conjunto diferente de dados. Os computadores SIMD trabalham com dados escalares e, portanto, processam dados sobre vetores simultaneamente, possibilitando a execução de uma instrução sobre diferentes elementos de um ou mais vetores simultaneamente. Entre os exemplos de máquinas que utilizam essa abordagem estão as chamadas máquinas de processamento vetorial (*Vector Machines*);
- MISD (*Multiple Instruction Stream, Single Data Stream*) - máquinas que possuem múltiplo fluxo de instruções e fluxo único de dados. Esta arquitetura permite que uma sequência única de dados seja aplicada a múltiplas sequências de instruções. Este tipo de arquitetura é conceitual e não possui aplicações práticas, assim, não vem sendo utilizada comercialmente ([STALLINGS, 2010](#)); e
- MIMD (*Multiple Instruction Stream, Multiple Data Stream*) - máquinas que possuem múltiplo fluxo de instruções e múltiplo fluxo de dados. As arquiteturas MIMD podem ser classificadas considerando diversos fatores, como: compartilhamento de memória principal, distância física entre os processadores, tempo de acesso a memória principal, mecanismos de comunicação e sincronização utilizados e a velocidade de comunicação entre os processadores ([MACHADO; MAIA, 1996](#)). A partir desses fatores, é possível determinar o grau de acoplamento de um sistema, permitindo que arquiteturas MIMD sejam classificadas em sistemas fortemente acoplados ou fracamente acoplados.

Nessa arquitetura, cada elemento processado identifica sua própria instrução e acessa seus dados de maneira compartilhada ou distribuída. A organização da arquitetura MIMD é muito utilizada atualmente por ser bastante flexível e por ser uma arquitetura que possibilita a construção de computadores que possam ser utilizados para propósitos gerais ([STALLINGS, 2010](#)).

Segundo [Pacheco \(2011\)](#), a arquitetura MIMD pode ser dividida em dois tipos de arquitetura: Memória Compartilhada e Memória Distribuída.

A grande diferença entre esses dois tipos de sistemas é que sistemas com memória compartilhada apresentam um único espaço para endereçamento de memória que é dividido por todos os processadores, daí vem o outro nome que pode se dar para classificação: sistemas

fortemente acoplados. Utilizando essa abordagem, a comunicação entre os processadores é realizada com o uso de variáveis globais presentes na memória principal, responsável por operações de leitura e escrita. Esse tipo de arquitetura é tipicamente utilizado em computadores com processadores *multicore* (STALLINGS, 2010).

Os sistemas de memória distribuída são normalmente formados por elementos processadores, em que cada sistema tem seu próprio espaço de endereçamento individual e são interligados por meio de uma rede que permite a troca de dados entre seus integrantes utilizando mecanismos de troca de mensagens (*send* e *recv*) (MACHADO; MAIA, 1996). Um exemplo de aplicação para sistemas de memória distribuída são *clusters*, outro exemplo é a formação de uma infraestrutura de computadores ligados em rede, comumente denominados de *grid*.

A Figura 5 apresenta um resumo das características descritas acima, conforme a classificação proposta por (FLYNN, 1966). Em (a) está representada a arquitetura SISD, a arquitetura SIMD é representada em (b) e por fim a arquitetura MIMD é representada em (c) para ambientes de memória compartilhada e em (d) para ambientes de memória distribuída.

Figura 5 – Organização de computadores paralelos adaptado de (STALLINGS, 2010).

3.3 Desenvolvimento de Aplicações Concorrentes

O desenvolvimento de programas em paralelo pode ser dividido em duas camadas: (i) a camada mais baixa que contém o núcleo da computação, no qual os processos manipulam

uma porção dos dados para produção do resultado. Uma linguagem de programação sequencial é normalmente a escolha mais adequada para a realização dessas atividades. E (ii) a camada superior controla a criação e sincronização de processos e o particionamento dos dados entre os processos. Essas ações podem ser desenvolvidas utilizando uma linguagem de programação que dê suporte nativo à criação de software em paralelo ou por meio de bibliotecas que facilitem o seu desenvolvimento utilizando uma linguagem de programação sequencial convencional (QUINN, 2004).

Contudo, diferente da produção de algoritmos sequenciais, o desenvolvimento de programas paralelos envolve mais do que apenas a especificação de um conjunto de passos necessários para a resolução de um determinado problema. É necessário que os desenvolvedores aprendam a programar aplicações que tirem proveitos das arquiteturas paralelas, a fim de se atingir um melhor desempenho. Com exceção de sistemas operacionais, sistemas de bancos de dados e servidores *web*, existem poucos softwares que exploram de fato os benefícios provenientes destas arquiteturas (PACHECO, 2011).

Algoritmos paralelos adicionam um grau de concorrência à aplicação e é necessário que o desenvolvedor especifique um conjunto de passos que possa ser executado de maneira simultânea. Essa tarefa é essencial para que seja possível atingir um ganho de desempenho. Segundo Grama (2003), para se projetar algoritmos paralelos eficientes, é necessário atentar para os seguintes fatores:

- identificar porções de trabalho que podem ser realizadas de maneira concorrente;
- mapear as porções de trabalho que podem ser executadas concorrentemente em múltiplos processos que executem em paralelo;
- distribuir entradas, saídas e dados intermediários, associados ao programa;
- gerenciar o acesso aos dados compartilhados entre os múltiplos processadores; e
- sincronizar os processadores durante os vários estágios da execução paralela do programa.

O processo de divisão da computação em pequenas partes, para que seja possível a identificação das que podem ser executadas paralelamente é chamado de decomposição. Após a decomposição é necessário que sejam definidas tarefas, que é a atividade de dividir a computação principal em subcomputações por meio da decomposição que foi inicialmente realizada. A execução simultânea de múltiplas tarefas é a chave para a redução do tempo de execução necessário para a solução de um problema.

Segundo Grama (2003), uma abstração utilizada para representar a dependência entre as tarefas e a sua relativa ordem de execução, é a utilização de um grafo orientado acíclico chamado de grafo de dependência de tarefas. Com a criação deste grafo é possível identificar as porções de

trabalho que são dependentes de outras tarefas e precisam, por exemplo, esperar por algum dado para concluir a sua computação, bem como tarefas que são independentes. Assim, é possível definir uma ordem precisa de execução das tarefas de forma a atingir a execução do algoritmo de maneira correta.

O número e o tamanho das tarefas em que o problema foi dividido determina a granularidade da decomposição do problema. Um problema decomposto em um grande número de pequenas tarefas possui granularidade fina, enquanto a decomposição do problema em um menor número de tarefas com uma porção maior de trabalho é chamada de granularidade grossa. A granularidade é responsável por definir o grau de concorrência da aplicação, sendo que o máximo de concorrência que pode ser obtido é definido pelo número máximo de tarefas que se pode executar em paralelo. A escolha entre o grau de granularidade a ser adotado está diretamente ligada ao modelo de arquitetura para o qual o software está sendo desenvolvido.

Na fase de mapeamento dos processos, são definidas quais tarefas serão executadas em cada processador ou *thread*. O objetivo principal do passo de mapeamento é garantir a utilização uniforme das unidades de processamento de forma a minimizar sua ociosidade e garantir que haja o menor taxa de comunicação possível entre os processadores (RAUBER; RÜNGER, 2010).

O método mais eficiente para determinar a ordem de execução de um conjunto de tarefas é definir um algoritmo de escalonamento que seja capaz de distribuir as tarefas respeitando suas restrições. Existem várias técnicas para realizar a decomposição e o mapeamento das tarefas a serem executadas. Um modelo de algoritmo paralelo é a melhor alternativa para estruturar o algoritmo paralelo, aplicando uma estratégia apropriada para minimizar a interação entre os processos.

Segundo Rauber e Rünger (2010) a execução de um programa provê diversas oportunidades para a paralelização de execução em diferentes níveis, sendo os principais modelos adotados:

- **Paralelismo em nível de instrução:** múltiplas instruções de um programa podem ser executadas simultaneamente, desde que sejam independentes umas das outras. Processadores superescalares com múltiplas unidades funcionais podem executar diversas instruções em paralelo. Eles utilizam um escalonamento de instruções dinâmico realizado em hardware que extrai instruções independentes;
- **Paralelismo de dados:** em diversos programas a mesma operação deve ser aplicada a diferentes elementos de uma grande estrutura de dados. Se as operações são independentes umas das outras, então é possível aplicar a paralelização de dados. Neste modelo as tarefas são estaticamente definidas, assim o trabalho pode ser realizado em etapas e as operações sobre os dados podem ser realizadas em diferentes tarefas.

Esse modelo de paralelismo se aplica tanto a arquiteturas SIMD aplicando a distribuição dos elementos em estruturas de dados como vetores, assim como para arquiteturas MIMD,

adotando uma abordagem *Single Program Multiple Data* (SPMD), em que um programa executa em vários processadores diferentes alterando apenas os valores utilizados na computação. Para tanto, é necessário que as etapas de computação sejam intercaladas com etapas de sincronização para que a semântica dos dados não seja perdida (GRAMA, 2003);

- **Paralelismo em *loop*:** algoritmos que realizam computações iterativas, extensamente utilizadas em linguagens de programação imperativas. Esse esquema de execução é chamado de *loop* sequencial e pode ser executados em paralelo desde que não haja dependência entre os elementos da execução iterativa;
- **Paralelismo funcional:** diversos programas sequenciais possuem partes que são independentes umas das outras e podem ser executadas em paralelo. As partes independentes de um programa podem ser: expressões simples, blocos de expressões, *loops* ou chamadas a funções. Considerando as partes independentes de um programa, esse paralelismo pode ser definido como paralelismo de tarefas ou paralelismo funcional; e
- **Representação explícita ou implícita de paralelismo:** o desenvolvimento de programas em paralelo é facilitado se não é necessária a representação explícita do paralelismo durante o desenvolvimento do software, neste caso é necessário que estejam disponíveis avançados compiladores capazes de identificar um código serial e produzir um código paralelo eficiente utilizando técnicas de compilação. Outra alternativa é a utilização de linguagens de programação funcionais, que podem executar determinados cálculos em paralelo devido às suas particularidades de desenvolvimento.

Representação implícita de paralelismo se dá em casos que é necessária a intervenção do programador no código especificando diretivas para que o compilador tenha conhecimento de quais atividades podem ser executadas de maneira concorrente. Entre as formas de utilização desta técnica estão bibliotecas de programação como *OpenMP*, *MPI*, *Pthreads*, etc.

3.4 Avaliação de Desempenho

A avaliação de desempenho pode ser definida como a identificação e medição das ações que um software realiza durante um intervalo de tempo. A avaliação de desempenho tem como objetivo avaliar e analisar o desempenho do software com intuito entender melhor o seu desempenho. Além disso, fornece informações importantes para tomadas de decisões de projeto e implementação, necessárias para a melhoria do produto.

Para avaliação de desempenho de um sistema existem basicamente três técnicas utilizadas: modelagem analítica, simulações e medições (JAIN, 1996).

A modelagem analítica realiza a avaliação de desempenho de sistemas criando métricas de desempenho definidas pelo desenvolvedor. Tais métricas são representadas com o uso de

variáveis que buscam representar as características do sistema e são medidas por meio de uma distribuição de probabilidade.

A técnica de simulação tem como objetivo representar a execução do sistema que se deseja avaliar, com intuito de medir seu comportamento e garantir que suas atividades satisfazem de fato os requisitos para o qual foi desenvolvido e, caso contrário, sejam realizadas as modificações necessárias.

A técnica de medição consiste em retirar informações diretamente do sistema. Sua grande vantagem é a exatidão dos resultados coletados, porém depende diretamente da existência do produto que se deseja avaliar. Uma das técnicas utilizadas para realizar a medição é a utilização de *benchmarks*. Na computação o termo *benchmarking* corresponde à atividade de executar um conjunto de programas representativos no sistema computacional que se tem como objetivo avaliar, a fim de se obter medidas de desempenho e realizar comparações (WU, 1999).

Segundo Jain (1996) as métricas de desempenho de software mais comuns encontradas são:

- **Tempo de resposta:** intervalo de tempo entre uma requisição realizada a sistema e a sua resposta. Essa métrica cresce proporcionalmente à carga computacional aplicada ao sistema. É uma métrica classificada como *Lower is Better* (menor é melhor);
- **Throughput ou Vazão:** taxa em que as requisições são servidas pelo sistema, ou seja, é a relação de ações ou transações que podem ser realizadas por um sistema em uma unidade de tempo. Tende a crescer proporcionalmente ao aumento de processamento no sistema, até estabilizar-se. É classificado como uma métrica *Higher is Better* (maior é melhor);
- **Utilização:** mede a relação entre a capacidade computacional utilizada e a capacidade computacional disponível;
- **Confiabilidade:** medida que define a probabilidade de erros de um determinado sistema ou tempo médio entre a ocorrência de erros; e
- **Disponibilidade:** medida que define a fração de tempo em que o sistema se encontra disponível para receber novas requisições.

3.5 Considerações Finais

Este capítulo abordou os principais conceitos de programação concorrente, apresentando suas definições, principais vantagens e desvantagens. Foram apresentadas características de computadores paralelos e como são definidas as arquiteturas desses computadores segundo a classificação de Flynn (1966).

Foi apresentado com destaque o padrão de arquitetura MIMD, máquinas que possuem múltiplo fluxo de instruções e múltiplo fluxo de dados, por serem um padrão muito utilizado devido a flexibilidade da arquitetura, que possibilita a construção de computadores que possam ser utilizados para propósitos gerais. Foram discutidos os passos fundamentais necessários para a criação de programas concorrentes destacando os principais modelos utilizados no desenvolvimento desse tipo de software, e por fim foram apresentados os aspectos gerais relacionados à análise de desempenho de programas. Essas informações são importantes para embasamento acerca dos dados coletados durante a execução dos experimentos realizados neste trabalho.

Após a apresentação dos conceitos relacionados ao Teste de Mutação (Capítulo 2) e sobre os conceitos relacionados a programação concorrente abordados neste capítulo, o próximo capítulo da continuidade na proposta apresentada no capítulo introdutório e descreve um conjunto de algoritmos desenvolvidos para distribuir a execução do Teste de Mutação em paralelo e uma arquitetura para dar apoio à aplicação destes algoritmos.

POTENCIALIZANDO O TESTE DE MUTAÇÃO COM COMPUTAÇÃO PARALELA

Como mencionado nos capítulos anteriores, um dos grandes problemas do teste de mutação está relacionado ao alto custo computacional envolvido na sua aplicação. Para tentar superar esta barreira, diversas abordagens foram propostas, conforme discutido na Seção 2.3.

Uma importante técnica de redução de custo é a execução em paralelo, que é largamente utilizada em outros contextos, como por exemplo, simulação de processos naturais (previsão do tempo e simulação de reações químicas). Existem diversos trabalhos na literatura que tratam sobre a execução paralela de mutantes. Diversas técnicas tentaram impulsionar o Teste de Mutação principalmente aplicando técnicas voltadas a arquiteturas de computadores específicas como máquinas *Hyper Cube* (FICHTER, 1991) ou máquinas de processamento vetorial (*Vector Machines*) (MATHUR; KRAUSER, 1988b).

A abordagem explorada neste capítulo utiliza como premissa o fato de a grande evolução na capacidade de processamento dos computadores atuais. Isso permite explorar como é possível potencializar a execução do Teste de Mutação aplicando técnicas de computação paralela em conjunto com algoritmos de balanceamento de carga a fim de contribuir para melhorar a sua eficiência.

Para avaliar esta ideia, foi construída uma arquitetura para possibilitar a aplicação do Teste de Mutação em paralelo, utilizando como suporte a ferramenta *Proteum*, descrita na Seção 2.4. Foram então acrescentados cinco algoritmos de balanceamento de carga para controlar a distribuição dos dados durante a execução do teste. Dois utilizam abordagem de distribuição de dados estática, dois usam uma abordagem de distribuição de dados dinâmica e um algoritmo uma abordagem mista.

Este capítulo apresenta uma descrição detalhada de cada algoritmo, bem como a descrição da arquitetura proposta para possibilitar a sua execução. O restante deste capítulo é organizado

da seguinte forma: A Seção 4.1 descreve cada algoritmo e as suas particularidades, a Seção 4.2 descreve em detalhes a arquitetura construída para possibilitar a aplicação dos algoritmos de balanceamento de carga e, por fim, a Seção 4.3 apresenta as considerações finais do capítulo.

4.1 Algoritmos

Uma abordagem prática para melhorar a eficiência de um programa é utilizar os recursos computacionais de maneira mais eficiente. Nesse sentido, abordagens de execução de programas em paralelo são consideradas uma boa alternativa (GRAMA, 2003). A ideia de um algoritmo paralelo consiste em dividir a execução de uma tarefa entre processadores ou máquinas de forma a reduzir a ociosidade de processamento e aumentar a eficiência na execução.

Um ponto fundamental para possibilitar a execução paralela do Teste de Mutação são os algoritmos de balanceamento de carga utilizados. Os algoritmos de balanceamento de carga coordenam a distribuição de dados e executam sempre no escalonador (Seção 4.2.2) e possuem um impacto fundamental também nos executores, pois para cada algoritmo utilizado no escalonador, os executores agem de modo a atendê-los de maneira apropriada, podendo interferir diretamente na forma como os dados são tratados, impactando diretamente no desempenho da estrutura como um todo.

Um algoritmo de balanceamento de carga básico divide o processamento de uma tarefa em P partes, cada parte P corresponde ao número de máquinas disponíveis para processar a demanda existente. O balanceamento de carga é um problema de escalonamento que consiste em determinar qual máquina irá executar determinada tarefa, de maneira a otimizar o desempenho do sistema.

Esta seção aborda cinco algoritmos de balanceamento de carga, que dividem a execução de atividades realizadas no Teste de Mutação entre várias máquinas a fim de melhorar a eficiência de aplicação da técnica, reduzindo o tempo de execução e maximizando o uso dos recursos computacionais.

Entre os algoritmos utilizados, são apresentadas duas abordagens estáticas e duas dinâmicas e uma abordagem mista. A análise realizada nas subseções seguintes foi feita com base na descrição dos algoritmos feita em (MATEO; USAOLA, 2013), contudo, faz-se aqui necessária para possibilitar entendimento e discussão acerca dos experimentos apresentados no Capítulo 5.

4.1.1 Algoritmo DMBO: Distribute Mutants Between Operators

DMBO é um algoritmo estático que distribui os mutantes gerados por um determinado operador de mutação em diversas partes. Dado um conjunto de máquinas M e um conjunto de mutantes definido por M_{OP} , gerados por meio do operador de mutação OP , cada parte contém um total de M_{OP}/M mutantes.

Cada executor recebe sua carga de trabalho somente durante o processo de geração dos mutantes. Contudo, graças à natureza de cada mutante e do conjunto de casos de teste utilizado, não existe garantia que os executores encerrarem suas atividades ao mesmo tempo, fazendo com que alguns executores possam encerrar suas atividades antes dos outros.

Esta abordagem de divisão de tarefas foi proposta por [Offutt et al. \(1992\)](#) e o Algoritmo 1 mostra uma versão em pseudocódigo da abordagem. As linhas 1, 2, 3 e 4 são responsáveis por inicializar variáveis, as linhas 5 até 10 criam as partes utilizando um laço de repetição separando os mutantes em partes iguais. Finalmente, nas linhas 11 e 12 as partes são enviadas para as máquinas que ficam responsáveis por processar os mutantes.

Este algoritmo tende a reduzir a comunicação entre as máquinas que compõem a estrutura. Assim, o tráfego na rede será mínimo, uma vez que cada máquina recebe dados somente durante o início do processo.

Contudo, devido à natureza dos mutantes e dos casos de teste utilizados, nem todas as execuções terão o mesmo tempo de execução. Então, conforme mostrado e discutido em ([OFFUTT et al., 1992](#)), algumas máquinas tendem a terminar seu processamento primeiro e o tempo total de execução do teste será definido em função das máquinas que tiverem processamento mais demorado.

Algoritmo 1: Algoritmo de Distribuição de Mutantes por Operador

Input: DicionarioMutantes:*ms*, CasosDeTeste:*ts*, listaExecutores:*executores*

```

1 p ← tamanho(executores)           // Retorna o tamanho da lista de executores
2 index ← 0
3 partes ← lista vazia com tamanho p
4 listaOp ← lista com operadores de mutação
5 for op em listaOp do
6   lista_mop ← ms[op]                // lista com mutantes gerados por op
7   k ← tamanho(lista_mop)            // Retorna o tamanho da lista lista_mop
8   for i = 0 até k do
9     partes[index] ← partes[index] ∪ {lista_mop[i], ts}
10    index ← (index + 1)%p
11 for (parte,executor) em (partes,executores) do
12   Enviar parte para executor

```

4.1.2 Algoritmo DTC: Distribute Test Cases

DTC é um algoritmo adaptado de idéias propostas por [Kapfhammer \(2001\)](#) e é outra abordagem de divisão de dados estática e similar ao algoritmo DMBO. Contudo utiliza como abordagem a distribuição de casos de teste entre as máquinas, ao invés de utilizar o conjunto de mutantes. Novamente, o número de porções de trabalho criadas é dado em função do número

de máquinas disponíveis para processamento, ou seja, o número de partes é dado por T/M . T representa o número de casos de teste e M é o número de máquinas disponíveis para execução.

Nessa abordagem o conjunto de casos de teste é dividido entre o número de máquinas disponíveis e cada executor recebe todos os mutantes gerados. Devido a esta característica, existe uma grande possibilidade de que um mutante seja morto várias vezes desnecessariamente.

Algoritmo 2: Conjuntos de Teste Distribuídos

Input: DicionarioMutantes:*ms*, CasosDeTeste:*ts*, listaExecutores:*executores*

```

1  $p \leftarrow \text{tamanho}(\text{executores})$       // Retorna o tamanho da lista de executores
2  $\text{index} \leftarrow 0$ 
3  $\text{partes} \leftarrow$  lista vazia com tamanho  $p$ 
4 for  $tc$  em  $ts$  do
5    $\text{partes}[\text{index}] \leftarrow \text{partes}[\text{index}] \cup \{tc, ms\}$ 
6    $\text{index} \leftarrow (\text{index} + 1) \% p$ 
7 for ( $\text{parte}, \text{executor}$ ) em ( $\text{partes}, \text{executores}$ ) do
8   Enviar  $\text{parte}$  para  $\text{executor}$ 
  
```

No Algoritmo 2 as linhas 1, 2 e 3 iniciam variáveis. As linhas 4, 5 e 6 criam as partes com um laço de repetição que distribui partes com o mesmo número de casos de teste e todo o conjunto de mutantes. Finalmente nas linhas 7 e 8, as partes são enviadas às máquinas executoras.

Assim como no DMBO, o tráfego na rede é minimizado pela distribuição dos dados ser realizada somente durante o início do processo de teste. Contudo, nem todas as execuções tendem a ter o mesmo intervalo de tempo de execução, pois cada caso de teste costuma ter particularidades, fazendo com que cada porção de trabalho distribuída venha a ter um valor de execução único.

Este algoritmo tende a apresentar um problema. Uma vez que não há comunicação entre as máquinas executoras (Figura 6), diversas máquinas tendem a matar mutantes de maneira repetida, causando assim um processamento desnecessário. Uma vantagem dessa abordagem, no entanto, pode ser observada quando o testador tem interesse em medir a eficiência de cada caso de teste de maneira individual, necessitando assim que todos os casos de testes sejam executados contra todos os mutantes, independente do estado do mutante no momento da execução.

4.1.3 Algoritmo GMOD: Given Mutants On Demand

GMOD foi um algoritmo proposto inicialmente por Choi e Mathur (1993) e é um algoritmo que utiliza uma abordagem dinâmica que divide a execução em partes, cada parte é composta por um único mutante e todo o conjunto de casos de teste. As partes são distribuídas entre as máquinas conforme exista demanda de processamento, até que todas as partes sejam distribuídas e processadas.

O algoritmo executa um laço de repetição que verifica se há algum executor disponível. Caso haja, ele distribui um mutante e todo o conjunto de casos de teste para o executor e aguarda até que um próximo executor esteja disponível para repetir a operação. O algoritmo encerra-se quando todos os mutantes são distribuídos entre os executores.

Algoritmo 3: Mutantes Distribuídos sob Demanda

Input: Mutantes:*ms*, CasosDeTeste:*ts*

```

1 index  $\leftarrow$  0
2 for m em ms do
3   partes[index]  $\leftarrow$  partes[index]  $\cup$  {m, ts}
4   index  $\leftarrow$  index + 1
5 indexProcessado  $\leftarrow$  0
6 while indexProcessado < index do
7   executor  $\leftarrow$  executorDisponivel() // Receber sinal de executor disponivel
8   Enviar partes[indexProcessado] para executor
9   indexProcessado  $\leftarrow$  indexProcessado + 1

```

No Algoritmo 3 a linha 1 inicia uma variável. As linhas 2, 3 e 4 são responsáveis por criar as partes com um laço de repetição. Então, as linhas 6, 7, 8 e 9 são responsáveis pelas rotinas de envio das partes para as máquinas que irão processar as partes sob demanda. Na linha 7, o algoritmo espera que uma máquina se encontre disponível. Então, quando uma máquina está disponível, na linha 8 são enviados dados para processamento. Esta rotina repete-se até que todas as porções de trabalho sejam enviadas e processadas.

Diferente dos algoritmos DMBO e DTC, o GMOD aumenta muito a comunicação na rede, pois é necessária uma comunicação constante entre as máquinas executoras e o escalonador. Assim, as características da rede podem ser um fator determinante no tempo de execução total. Contudo, devido à natureza dinâmica do algoritmo, todas as máquinas tendem a terminar a sua execução em um intervalo de tempo semelhante.

4.1.4 Algoritmo GTCOD: Given Test Cases on Demand

GTCOD é um algoritmo dinâmico adaptado ao contexto do Teste de Mutação também proposto por [Kapfhammer \(2001\)](#). Neste algoritmo as execuções são divididas utilizando como base os casos de teste disponíveis.

As partes são inicialmente divididas em tuplas (*tc*, *ms*) e cada tupla é constituída de um único caso de teste *tc* e *ms* que se refere ao conjunto completo de mutantes. Semelhante ao algoritmo GMOD, as partes são divididas entre as máquinas disponíveis para processamento sob demanda. Esta operação é repetida até que todas as partes sejam distribuídas e processadas.

O algoritmo distribui a execução dos casos de teste a serem processados. Assim como na abordagem anterior, a execução encerra-se, quando todos os casos de teste são distribuídos entre os processadores.

Algoritmo 4: Casos de teste distribuídos sob demanda

Input: Mutantes: ms , CasosDeTeste: ts

```

1  $index \leftarrow 0$ 
2 for  $tc$  em  $ts$  do
3    $partes[index] \leftarrow partes[index] \cup \{ms, tc\}$ 
4    $index \leftarrow index + 1$ 
5  $indexProcessado \leftarrow 0$ 
6 while  $indexProcessado < index$  do
7    $executor \leftarrow executorDisponivel()$  // Receber sinal de executor disponivel
   Enviar  $partes[indexProcessado]$  para  $executor$ 
8    $indexProcessado \leftarrow indexProcessado + 1$ 

```

No Algoritmo 4, a linha 1 inicializa variáveis. As linhas 2, 3 e 4 criam as porções de trabalho com um laço de repetição e cada parte é composta por uma tupla formada por (tc, ms) . Nas linhas 6, 7, 8 e 9 as partes são enviadas sob demanda às máquinas disponíveis. Quando uma máquina se encontra disponível, uma porção de trabalho é enviada. O laço se encerra quando todas as partes são enviadas e processadas pelas máquinas executoras.

Assim como o GMOD, o algoritmo GTCOD aumenta o tráfego de informações na rede. Contudo, este algoritmo tende a possuir um impacto menor em comparação ao GMOD, devido o número de mutantes ser normalmente bem superior ao número de casos de teste. Assim como o DTC, no GTCOD as máquinas tendem a matar mutantes de maneira repetida, pelo fato de não haver comunicação entre as máquinas executoras, causando assim um possível processamento desnecessário.

Diferente dos algoritmos estáticos, assim como o GMOD, existe um intenso fluxo na rede devido à grande comunicação entre o escalonador e o executores. Assim, a infraestrutura da rede pode influenciar diretamente no desempenho do algoritmo.

4.1.5 Algoritmo PEDRO: Parallel Execution with Dynamic Ranking and Ordering

O algoritmo PEDRO é um algoritmo híbrido com características estáticas e dinâmicas proposto por [Mateo e Usaola \(2013\)](#). Este algoritmo foi inspirado em ideias propostas por [Hummel, Schonberg e Flynn \(1992\)](#), porém foi modelado com o objetivo de ser aplicado ao contexto do Teste de Mutação.

O algoritmo PEDRO é dividido basicamente em duas etapas: A primeira fase é denominada de fase de **ranqueamento**, nessa fase o algoritmo tem um comportamento similar ao algoritmo DMBO; Na segunda fase denominada de fase de **ordenação**, o algoritmo se assemelha ao algoritmo GMOD. Com esta estrutura, o algoritmo propõe uma abordagem mista que graças à primeira etapa tem por finalidade reduzir o tráfego na rede em relação ao que acontece com o

algoritmo GMOD. E por conta da segunda etapa, previne que máquinas fiquem com processamento ocioso durante o processo, evitando que algumas máquinas encerrem seu processamento antes das outras.

Ambas as fases são configuradas por dois parâmetros, que podem definir o comportamento do algoritmo como mais próximo dos algoritmos DMBO ou GMOD. O primeiro parâmetro é um número inteiro controlado pelo usuário que determina o número de porções de trabalho a serem executado em cada fase. Quanto menor o parâmetro, maior o número de partes criadas na primeira parte e vice-versa. O segundo parâmetro é o número de máquinas disponíveis para execução. O algoritmo PEDRO não divide o conjunto de casos de teste para evitar o problema de múltiplas mortes de mutantes descritos nos algoritmos DTC e GTCOD (Seções 4.1.2, 4.1.4).

Algoritmo 5: Execução Paralela com Ordenação e Ranqueamento Dinâmicos

Input: Mutantes:*ms*, CasosDeTeste:*ts*, listaExecutores:*executores*, inteiro: α

```

1 //Primeira Fase
2  $p \leftarrow \text{tamanho}(\text{executores})$            // Retorna o tamanho da lista de executores
3  $k \leftarrow \text{tamanho}(ms)$                    // Retorna o tamanho da lista de mutantes
4  $index \leftarrow 0$ 
5  $partes1 \leftarrow$  lista vazia com tamanho número de máquinas usadas na sessão de teste
6  $tamanhoPartes \leftarrow \max(k/(p * \alpha), 1)$ 
7 for  $i = 0$  até  $p$  do
8   for  $j = 0$  até  $tamanhoPartes$  do
9      $partes1[i] \leftarrow partes1[i] \cup \{ms[index], ts\}$ 
10     $index \leftarrow (index + 1) \% p$ 
11 for ( $parte, executor$ ) em ( $partes, executores$ ) do
12   Enviar  $parte$  para  $executor$ 
13 // Segunda Fase
14 while  $index < k$  do
15    $executor \leftarrow \text{executorDisponivel}()$  // Receber sinal de executor disponivel
16    $tamanhoPartes \leftarrow \max(k - index/(p * \alpha), 1)$ 
17    $parte \leftarrow \emptyset$ 
18   for  $i = 0$  até  $tamanhoPartes$  do
19      $parte \leftarrow parte \cup \{ms[index], ts\}$ 
20      $index \leftarrow (index + 1)$ 
21   Enviar  $parte$  para  $executor$ 

```

As entradas deste algoritmo são o número de máquinas disponível, o conjunto de mutantes, o conjunto de casos de teste e um valor α definido pelo usuário, que determina o número de partes que serão processadas em cada fase do algoritmo.

As linhas 2, 3, 4 e 5 inicializam variáveis. As linhas 6 até 11 criam o primeiro conjunto de partes com um laço, distribuindo em cada parte o mesmo número de mutantes. Na linha 12 o primeiro conjunto de partes é enviado para as máquinas executoras disponíveis. As linhas 14 até

23 distribuem os mutantes que não foram enviados na primeira fase em diferentes partes que são criadas e enviadas sob demanda para as máquinas. Na linha 15 o algoritmo espera até que a primeira máquina encerre o processamento da primeira fase. Quando isto ocorre, o tamanho da nova parte é calculado na linha 16 e então as linhas 17 até 21 criam a nova porção a ser processada. Na linha 22 a parte é enviada à máquina executora disponível. O laço se encerra quando todos os mutantes são processados.

Como definido anteriormente, o algoritmo é dividido em duas etapas: a fase de ranqueamento (primeira etapa) e a fase de ordenação (segunda etapa). A fase de ranqueamento envia um número fixo de mutantes para cada máquina. Esta fase se torna útil para realizar um ranqueamento das máquinas para a próxima fase, garantindo que as máquinas com melhor desempenho na primeira fase processem uma maior quantidade de dados na segunda. Na primeira fase os dados são enviados uma única vez, reduzindo assim o tráfego na rede.

Quando uma máquina encerra a execução da primeira fase, a etapa de ordenação é iniciada. Nesta fase, o algoritmo divide a execução em partes de tamanho decrescente. Quando uma máquina finaliza sua execução, ela recebe uma nova porção de trabalho para ser executada. O tamanho desta parte depende do número de mutantes a ser executado. Assim, quanto mais a execução dos mutantes se aproxima do fim, menor é o tamanho das partes criadas. Este comportamento indica que a segunda fase do algoritmo PEDRO tende a se assemelhar ao algoritmo GMOD no fim do processo.

Este algoritmo, no entanto, reduz o tráfego na rede, quando comparado com GMOD e previne que as máquinas fiquem em estado ocioso, desperdiçando processamento. Contudo, de acordo com as configurações, o algoritmo pode ser mais parecido com GMOD (aumentando o tráfego na rede) ou mais similar ao DMBO (com uma grande diferença entre o tempo de processamento das máquinas envolvidas no processo).

4.2 Arquitetura

A ferramenta *Proteum* foi estendida com objetivo de avaliar o comportamento de algoritmos de balanceamento de carga, propostos na literatura e abordados na Seção 4.1, quando aplicados ao contexto de programas desenvolvidos na linguagem C¹.

Devido às características de independência entre os módulos da ferramenta *Proteum* (Seção 2.4), foi possível criar uma arquitetura que possibilite a execução do Teste de Mutação em paralelo. O foco desta abordagem é disponibilizar uma infraestrutura que apoie a execução do Teste de Mutação, atendendo de forma transparente à demanda de execução por meio de algoritmos de balanceamento de carga. Para isso, foi desenvolvida uma interface para possibilitar a comunicação com os módulos da versão sequencial da ferramenta e posteriormente uma

¹ <http://www.open-std.org/JTC1/SC22/WG14/>

arquitetura que controla a distribuição dos dados e execução dos módulos de forma a possibilitar a comunicação entre várias instâncias da versão sequencial da ferramenta *Proteum*.

Figura 6 – Teste de Mutação em Paralelo - Arquitetura

A Figura 6 mostra como foi projetada a arquitetura para a utilização da ferramenta *Proteum* em paralelo. Foi criada uma interface com objetivo de se comunicar com a versão sequencial da ferramenta. A interface foi construída utilizando a linguagem de programação Python², e é constituída de um conjunto de rotinas responsáveis por realizar chamadas aos módulos da ferramenta, além de tratar os dados recebidos e apresentá-los de maneira adequada à cada módulo.

A comunicação entre as máquinas também foi construída utilizando a linguagem de programação Python, com o auxílio da API (*Application Program Interface*) ZMQ³, utilizada para abstrair a camada de redes responsável pela comunicação entre as máquinas.

A arquitetura foi concebida utilizando três tipos distintos de máquinas (servidor, escalonador e executor), a Figura 7 mostra um diagrama de sequência que exemplifica de maneira resumida os passos envolvidos durante o processo de execução do Teste de Mutação em paralelo.

² www.python.org

³ www.zeromq.org

Figura 7 – Teste de Mutação em Paralelo - Diagrama de Sequência

É possível observar que cada tipo de máquina realiza um conjunto distinto de tarefas. As ações enfatizadas por ❶ e ❷ podem repetir em um ciclo conforme o algoritmo de balanceamento de carga utilizado.

O servidor é responsável por controlar a sessão de teste e interagir com a ferramenta *Proteum*. O escalonador fica responsável por prover a comunicação entre o servidor e as máquinas executoras. Assim, ele define a política de distribuição de dados (algoritmo de balanceamento de carga) e controla a distribuição de dados entre o servidor e os executores.

As máquinas executoras também utilizam uma versão sequencial da ferramenta *Proteum* e possuem a única função de executar a carga de trabalho que lhes é atribuída, de acordo com a abordagem informada pelo escalonador, e ao fim, informar os resultados da computação realizada.

As próximas subseções destacam individualmente cada tipo de máquina, bem como o seu comportamento esperado.

4.2.1 Servidor

O servidor é a máquina responsável por controlar o processo de execução do Teste de Mutação. Ele é responsável por definir o grupo de mutantes e o conjunto de casos de teste que serão utilizados na sessão de teste.

A Figura 8 mostra o comportamento do servidor, que é responsável por enviar para o escalonador as informações do programa em teste e informar quais conjuntos de mutantes e casos de testes serão utilizados durante a sessão de teste.

Figura 8 – Diagrama de Máquina de Estados - Comportamento do servidor

O processo inicia de maneira idêntica ao processo de execução do Teste de Mutação em

um ambiente sequencial (Seção 2.2). São gerados os mutantes e definidos os casos de teste que serão utilizados na execução. Após este processo, as informações são enviadas ao escalonador. O servidor então aguarda que o escalonador distribua os dados e receba o resultado da execução para processá-los e informar ao servidor.

Ao receber os dados no fim da execução o servidor sumariza os resultados e calcula o escore de mutação (Seção 2.2). Neste ponto é possível iniciar uma nova iteração com novos dados para teste ou encerrar a execução. Ao encerrar a sua execução, o servidor notifica o escalonador para que este também tenha o seu processo encerrado.

4.2.2 Escalonador

O escalonador é a máquina responsável por distribuir porções de trabalho (mutantes e casos de teste) entre as máquinas executoras. Como este não processa nenhum dado, apenas o distribui, o mesmo é executado independente da presença da ferramenta *Proteum* no ambiente.

Figura 9 – Diagrama de Máquina de Estados - Comportamento do escalonador

O comportamento do escalonador pode ser observado na Figura 9. Esta é a única máquina que possui ligação com todas as outras que participam do processo de teste. Conforme especificado na Figura 6, ele é responsável por receber e controlar a distribuição dos dados durante a execução dos testes.

Ao iniciar o seu processo, o escalonador define o número de máquinas executoras a serem utilizadas na sessão de teste, bem como o algoritmo de balanceamento de carga que será utilizado. Em seguida, aguarda por conexões tanto do servidor quanto dos executores até que o número de conexões definidas seja satisfeito.

À medida que as restrições de início são atendidas, o escalonador inicia então o processo de distribuição dos dados, podendo dividir cargas de trabalho de maneira estática ou dinâmica, conforme o algoritmo de balanceamento de carga informado. Para todos os algoritmos o escalonador sempre recebe do servidor todas as informações sobre mutantes e casos de testes existentes na

sessão de testes. Contudo, a forma como as informações são repassadas aos executores depende da abordagem de balanceamento de carga utilizada.

Quando os executores encerram o seu processamento, o escalonador recebe os resultados processados e os informa ao servidor. Neste ponto, o escalonador aguarda receber novos dados de teste para distribuição ou por uma mensagem que determine o fim da sua execução.

4.2.3 Executor

Os executores são máquinas responsáveis por realizar a execução dos mutantes com o conjunto de casos de teste fornecidos. Em linhas gerais, cada executor comporta-se como uma versão sequencial da ferramenta *Proteum*. Contudo, a forma como os dados são processados difere de uma execução sequencial convencional devido ao fato do executor processar sua base de mutantes com base em algoritmos de balanceamento de carga, processando apenas uma porção do conjunto total de mutantes.

Ao iniciar a execução, todos os executores conectam-se ao escalonador e recebem como resposta desta ação uma cópia do programa em teste. Com base nesta informação, cada executor age como uma versão sequencial da ferramenta *Proteum* e cria a sua própria sessão de teste. Após criar sua sessão de teste os executores passam a realizar requisições ao escalonador informando que estão prontos para processar dados. Como resposta a estas requisições, os executores recebem do escalonador um conjunto composto por identificadores de mutantes e casos de teste a serem processados.

A informação repassada pelo escalonador consiste em identificadores de mutantes e casos de teste. Esta informação é processada pelo executor, com base no algoritmo de balanceamento de carga adotado pelo escalonador para que o executor saiba quais mutantes da sua sessão de testes que devem ser processados.

Figura 10 – Diagrama de máquina de estados - Comportamento dos executores

A Figura 10 mostra o comportamento de cada executor. À medida que o executor

recebe requisições, elas são processadas e, ao fim do processo, o executor informa os resultados processados ao escalonador. Ao processar todos os dados solicitados, o executor tentará se registrar novamente ao escalonador. Caso o escalonador não informe dados de teste ao executor, o mesmo encerrará seu processo por *timeout*.

4.3 Considerações Finais

Conforme discutido no capítulo 2, desempenho ainda é um grande entrave na aplicação do Teste de Mutação, o que garante a relevância da prova de conceito apresentada neste capítulo. Dado que a execução dos mutantes pode ser tratada como um conjunto de processos independentes, este é um problema que acaba se encaixando muito bem em uma estrutura para execução em paralelo.

Neste capítulo foram apresentados o projeto e as características de uma arquitetura para possibilitar a aplicação do Teste de Mutação em paralelo. Foi apresentado um conjunto de cinco algoritmos de balanceamento de carga que quando aplicados a esta arquitetura permitem a coordenação e distribuição dos dados entre os computadores incorporados à arquitetura.

TESTE DE MUTAÇÃO EM PARALELO: EXPERIMENTOS

Conforme mencionado nos capítulos anteriores, um dos maiores desafios encontrados na aplicação do Teste de Mutação, é o problema relacionado ao grande custo computacional envolvido na sua aplicação. Para superar esse problema diversas abordagens foram propostas ao longo dos anos, conforme discutido anteriormente (Seção 2.3). Nessa direção, este capítulo apresenta os experimentos realizados com objetivo de validar a arquitetura proposta no Capítulo 4 e de responder às questões de pesquisa que nortearam o desenvolvimento deste trabalho.

A Seção 5.1 descreve o planejamento do experimento, além de sumarizar a forma que os resultados obtidos foram analisados. Na Seção 5.2 é descrito o procedimento para a execução do experimento, a Seção 5.3 descreve como foi realizada a coleta dos dados gerados após a execução do experimento e traz a discussão acerca dos resultados. Essa discussão é realizada por meio de análises descritivas e de testes de hipótese. A Seção 5.4 discute a viabilidade de aplicação de cada algoritmo e faz um comparativo dos resultados obtidos nos experimentos realizados neste trabalho com os resultados apresentados por [Mateo e Usaola \(2013\)](#). Por fim, a Seção 5.5 apresenta as considerações finais para este capítulo.

5.1 Planejamento

Este experimento tem como objetivo analisar como algoritmos de balanceamento de carga apresentados no Capítulo 4 se comportam ao serem inseridos durante a aplicação do Teste de Mutação em paralelo e como podem contribuir para melhorar a eficiência e aplicabilidade do critério. Foi definido um ambiente de execução para a paralelização do teste de mutação com base na arquitetura descrita na Seção 4.2 e os diferentes algoritmos de balanceamento de carga implementados foram avaliados. Os resultados mostram os algoritmos mais eficientes e é discutido como os resultados obtidos neste experimento se relacionam com os resultados

apresentados em (MATEO; USAOLA, 2013).

Esta seção descreve a configuração dos experimentos realizados para medir a eficiência da aplicação do Teste de Mutação em paralelo. Especificamente, pretende-se investigar as seguintes questões de pesquisa:

- **QP₁** : É possível melhorar a eficiência computacional na aplicabilidade do Teste de Mutação utilizando técnicas de computação paralela?
- **QP₂** : Existe diferença significativa entre o tempo de execução dos algoritmos de balanceamento de carga implementados na arquitetura proposta?
- **QP₃** : Como os resultados obtidos se comportam em relação a trabalhos similares existentes na literatura?

Para avaliar a **QP₁**, a implementação foi comparada com a versão sequencial da ferramenta de Teste de Mutação *Proteum* (DELAMARO; MALDONADO, 1996b).

Para avaliar a **QP₂**, os resultados dos experimentos realizados foram comparados entre si. O objetivo desta comparação é determinar como cada algoritmo de balanceamento de carga se comportou durante a execução dos experimentos e quais aqueles que possuem melhor desempenho.

Para avaliar a **QP₃**, os resultados obtidos nos experimentos serão comparados com os resultados apresentados por Mateo e Usaola (2013), que também implementam os algoritmos de balanceamento de carga. O objetivo dessa avaliação é medir o comportamento de cada algoritmo quando aplicado em diferentes linguagens de programação e diferentes ambientes. No artigo supracitado os resultados são apresentados no contexto do Teste de Mutação aplicado para programas em Java e no experimento realizado nesta dissertação, o foco são programas desenvolvidos na linguagem C.

5.1.1 Definição dos objetivos

Foi utilizado o modelo proposto por Wohlin *et al.* (2000) para aplicação do método *Goal/Question/Metric* (GQM) (BASILI; WEISS, 1984). O modelo para aplicação do GQM apresenta o experimento dividido em cinco partes: objeto de estudo, propósito, perspectiva, foco qualitativo e contexto.

- **Objetos de estudo:** Os objetos de estudo são os cinco algoritmos de balanceamento de carga apresentados na Seção 4.1 e a ferramenta *Proteum*.
- **Propósito:** O propósito deste experimento é avaliar a aplicabilidade do Teste de Mutação utilizando técnicas de execução paralela de programas. Especificamente pretende-se

investigar o comportamento de cinco algoritmos de balanceamento de carga aplicados ao Teste de Mutação. O experimento oferece uma visão do quanto é possível melhorar a performance computacional na aplicabilidade do Teste de Mutação utilizando esta abordagem.

- **Perspectiva:** Este experimento é executado do ponto de vista de um pesquisador.
- **Foco qualitativo:** O efeito primário em investigação é o custo computacional medido pelo tempo de execução. O tempo de execução de um programa em teste é definido como sendo o tempo gasto pela ferramenta para executar o programa e todos os mutantes contra o conjunto de casos de teste.
- **Contexto:** Este experimento foi conduzido utilizando a ferramenta de Teste de Mutação *Proteum Ver. 2.1*¹ em um conjunto de máquinas Intel Core I7 com 8GB de memória RAM executando o sistema operacional *Ubuntu Linux 15.04 LTS*. Como a implementação é uma prova de conceito acadêmica, os programas utilizados como objetos experimentais deste experimento não possuem uma significância industrial. O código fonte dos programas utilizados foi extraído de livros sobre Engenharia de Software e os resultados aqui apresentados se destinam exclusivamente para ambientes acadêmicos.

Assim, o experimento realizado pode ser resumido utilizando o seguinte modelo proposto por Wohlin *et al.* (2000):

Analisar algoritmos de balanceamento de carga e a ferramenta *Proteum*

com o propósito de avaliar a aplicabilidade do Teste de Mutação usando computação paralela

no que diz respeito ao custo computacional

do ponto de vista de um pesquisador

no contexto de um conjunto de programas para a linguagem C, extraídos de livros de Engenharia de Software.

5.1.2 Ambiente

Este experimento foi conduzido nas dependências da Universidade de São Paulo (USP), Campus São Carlos. Foram utilizadas 34 máquinas do laboratório EC-105 para execução dos testes. A configuração das máquinas utilizadas pode ser visualizada na Tabela 1.

¹ <https://drive.google.com/open?id=0B4n1OQ2HWrYVZGtqYmowTkIiNOTg>

Tabela 1 – Características dos computadores utilizados no experimento

Código	Tipo de Máquina	Nº Máquinas	Processador	Clock (GHz)	Memoria (GBs)	Cores
1	Servidor	1	Core I7	3 GHz	8GB	4
2	Escalonador	1	Core I7	3 GHz	8GB	4
3	Executor	32 (Max)	Core I7	3 GHz	8GB	4

5.1.3 Formulação das hipóteses

As questões de pesquisa (**QP₁**, **QP₂**) apresentadas no início desta Seção foram convertidas em hipóteses para que fosse possível a realização de testes estatísticos.

Para a questão de pesquisa **QP₁** foram definidas as seguintes hipóteses:

Hipótese Nula, H_0 : Não há diferença significativa no tempo de execução gasto na aplicação do Teste de Mutação utilizando uma execução sequencial e na execução paralela.

$$H_0: {}^{\mu}Proteum = {}^{\mu}Proteum^p$$

Hipótese Alternativa, H_1 : Há uma diferença significativa no tempo de execução gasto na aplicação do Teste de Mutação em paralelo com relação a uma execução sequencial.

$$H_1: {}^{\mu}Proteum \neq {}^{\mu}Proteum^p$$

O mesmo procedimento foi realizado para a questão de pesquisa **QP₂**. Assim, foram definidas as seguintes hipóteses:

Hipótese Nula, H_0 : Não há diferença significativa entre o tempo de execução gasto por cada algoritmo na aplicação do Teste de Mutação em paralelo.

$$H_0: DMBO = DTC = GMOD = GTCOD = PEDRO$$

Hipótese Alternativa, H_1 : Há diferença significativa entre o tempo de execução gasto por cada algoritmo na aplicação do Teste de Mutação em paralelo.

$$H_1: DMBO \neq DTC \neq GMOD \neq GTCOD \neq PEDRO$$

5.1.4 Seleção de variáveis

Para todos os experimentos realizados foram controladas duas variáveis independentes, sendo a primeira ordinal que define o número de máquinas executoras utilizadas e a segunda cardinal, que define qual algoritmo de balanceamento de carga deve ser utilizado.

As variáveis independentes alteradas pelo tratamento são:

- Número de máquinas executoras: Variável ordinal (inteira/discreta) que define o número de máquinas executoras a serem utilizadas na sessão de testes;
- Tipo de algoritmo utilizado: Variável cardinal (opções não numéricas) que é alternada para definir o algoritmo de balanceamento de carga (DMBO, DTC, GMOD, GTCOD, PEDRO) que será utilizado na sessão de teste.

A variável dependente coletada pelo tratamento é o tempo utilizado por todas as máquinas envolvidas na sessão de teste, desde o início do processo até a mensagem de interrupção final. Esta é uma variável contínua (paramétrica) coletada em segundos.

O experimento é dividido em duas etapas. O objetivo da primeira é realizar um teste de hipótese para evidenciar qual abordagem (sequencial ou paralela) é mais eficiente para execução do experimento. Por ser um fato conhecido que implementações que se utilizam de uma infraestrutura complexa tenderem a possuir um desempenho superior a estruturas convencionais, a confirmação desse fato aumentaria a confiança nos dados obtidos, o que fornece subsídios para a segunda fase do experimento. Além de possibilitar estabelecer o quão efetivo pode ser o ganho na utilização de computação paralela com relação à aplicação sequencial. A segunda etapa do experimento tem como objetivo avaliar individualmente cada algoritmos de balanceamento de carga, para que se possa atestar o algoritmo mais adequado e para que seja possível ao fim traçar um comparativo com os resultados apresentados por [Mateo e Usaola \(2013\)](#).

Conforme descrito na Subseção 4.1.5, o algoritmo PEDRO possui um parâmetro α , que ajusta o número de porções que o mesmo adota para a divisão das sua carga de trabalho. Para possibilitar comparação com os resultados apresentados em ([MATEO; USAOLA, 2013](#)), foi adotado o mesmo critério para a definição deste parâmetro, o critério consiste em definir o valor de α como sendo igual ao número de máquinas utilizadas durante a execução do algoritmo.

5.1.5 Seleção dos objetos experimentais

Os objetos experimentais utilizados neste experimento consistem em uma lista com trinta e nove programas escritos na linguagem C, utilizados para aplicação de experimentos anteriores no contexto de Tesde de Mutação ([DELAMARO; OFFUTT; AMMANN, 2014](#); [DELAMARO et al., 2014](#); [AMMANN; DELAMARO; OFFUTT, 2014](#)). A lista completa com o nome dos programas, número de funções, linhas de código (LC), número de mutantes e número de casos de teste utilizados é descrita no Quadro 2.

O conjunto de casos de teste utilizados neste experimento foram aproveitados com base em trabalhos realizados anteriormente. Como o foco deste experimento é medir a capacidade de execução dos mutantes, foram utilizados todos os operadores de mutação clássicos propostos por [Agrawal et al. \(1989\)](#) para a linguagem C, assim como os operadores de deleção apresentados

por [Delamaro, Offutt e Ammann \(2014\)](#). Também foram ignoradas informações relacionadas à equivalência de mutantes.

Quadro 2 – Programas utilizados para execução do experimento

Programas	Funções	LC	Mutantes	Casos de Testes
boundedQueue	6	49	1121	13
cal	71	18	891	8
Calculation	7	46	1118	13
checkIt	1	9	104	9
CheckPalindrome	1	10	166	8
countPositive	1	9	151	5
date-plus	3	132	2421	44
DigitReverser	1	17	496	5
findLast	1	10	198	6
findVal	1	7	190	7
Gaussian	6	23	1086	21
Heap	7	41	1079	8
InversePermutation	1	15	576	12
jday-jdate	2	49	2 821	27
lastZero	1	9	173	5
LRS	5	51	1132	8
MergeSort	3	32	991	18
numZero	1	10	151	5
oddOrPos	1	9	361	7
pcal	8	204	6419	49
power	1	11	268	9
printtokens	17	349	4322	34
printtokens2	1	275	4734	27
printPrimes	2	35	715	7
Queue	6	64	469	12
quicksort	1	23	1026	13
RecursiveSort	1	17	555	8
replace	20	390	11.100	142
schedule	18	213	2108	45
schedule2	16	195	2626	41
Stack	6	56	460	11
stats	1	19	884	7
sum	1	7	165	6
tcas	8	63	2384	62
testPad	1	24	629	14
totInfo	7	214	6693	49
trashAndTakeOut	2	19	599	12
twoPred	1	10	246	10
UnixCal	4	119	4852	27

5.1.6 Descrição dos ambientes de execução

Foi definido um conjunto de ambientes para avaliar o desempenho da execução do Teste de Mutação em paralelo. A Tabela 1 descreve a configuração das máquinas utilizadas. Os ambientes de execução do experimento são compostos de:

- **Servidor:** Uma máquina responsável por hospedar a aplicação que controla a sessão de teste.
- **Escalonador:** Uma máquina responsável por executar os algoritmos de balanceamento de carga e controlar a distribuição de dados durante a execução.
- **Executores:** Computadores responsáveis por processar a demanda de trabalho. Para esse tipo de máquina foram variadas durante o experimento configurações com 4, 8, 12, 16 e 32 executores.

5.2 Condução

O procedimento realizado durante a execução do experimento consiste na execução das seguintes etapas:

1. Configuração do ambiente das máquinas do laboratório para execução da abordagem proposta.
 - a) Configuração da máquina *servidor*: informar os programas que serão utilizados na sessão de testes, informar conjunto de casos de teste a ser utilizado, informar o endereço IP da máquina escalonador e iniciar o processo.
 - b) Configuração da máquina *escalonador*: informar o número de máquinas executoras que será utilizado, informar o algoritmo de balanceamento de carga que será adotado e iniciar o processo.
 - c) Configuração das máquinas *executoras*: informar o endereço da máquina *escaladora* em cada uma das máquinas executoras e iniciar o processo.
2. Geração dos mutantes no computador *servidor*.
3. Repassar dados da sessão de teste para o *escalonador*.
4. Distribuição dos dados de teste do *escalonador* para as máquinas *executoras*.
5. Execução dos mutantes contra o conjunto de casos de teste, conforme a abordagem selecionada em *1b*.
6. Sumarização dos resultados em cada *executor*.

7. Repasse dos resultados de cada *executor* ao *escalonador*.
8. Repasse final dos resultados do *escalonador* ao *servidor*.

5.3 Coleta e Análise dos Dados

Esta Seção descreve o processo para coleta (subseção 5.3.1) dos dados obtidos na execução do experimento e a análise dos resultados obtidos após a execução. A etapa de análise está dividida em duas partes (i) análise descritiva e (ii) testes hipótese.

5.3.1 Coleta

A coleta de informações sobre as variáveis citadas na subseção 5.1.4 é feita utilizando um módulo de *profiling* disponível para *Python*. Assim, são coletadas informações acerca de todas as chamadas realizadas durante a execução do experimento.

Por conta do fator não determinístico dos ambientes de execução, as variáveis de tempo coletadas no experimento podem ser afetadas por vários elementos como escalonamento do sistema operacional, interrupções, etc. Para mitigar este risco, o experimento realizado foi executado quinze vezes e as variáveis de tempo apresentadas nas seções posteriores correspondem à média das quinze execuções realizadas.

5.3.2 Análise descritiva

Esta primeira etapa compreende a uma análise descritiva dos dados coletados durante a execução do experimento. Conforme descrito anteriormente (Seção 5.2), o experimento foi executado quinze vezes e todos os valores aqui apresentados correspondem à média do tempo de execução das quinze execuções.

A primeira análise a ser realizada diz respeito ao *speedup* alcançado ao utilizar uma abordagem de execução paralela de mutantes quando comparada à utilização da versão sequencial da ferramenta *Proteum*.

Neste cenário serão inicialmente considerados os dados referentes ao tempo total gasto na execução de todo o experimento. O experimento consiste na execução de todos os mutantes com os conjuntos de casos de teste relacionados no Quadro 2. O tempo de execução coletado consiste no tempo gasto para a execução completa do experimento, em outras palavras, o tempo gasto para a execução de todos os programas.

Inicialmente é apresentada uma análise do resultado considerando todos os 39 programas e em seguida, são detalhados de maneira individualmente os dados referentes à execução de cada programa utilizado como objeto de estudo do experimento. O tempo de execução coletado nesta etapa corresponde ao tempo gasto para a execução de cada programa de maneira individual. Esta

discussão é relevante para que seja possível descrever como cada uma das abordagens utilizadas se comporta em relação a cada programa utilizado na execução do experimento.

Como a finalidade deste experimento é medir o quanto a eficiência da execução do Teste de Mutação pode ser melhorada, foi adotada a sessão de teste da ferramenta *Proteum* em modo *research* (Seção 2.4). Esta abordagem obriga que todos os casos de teste existentes na sessão sejam executados com todo o conjunto de mutantes, independentemente do estado do mutante no momento da execução do caso de teste. Em uma sessão de teste convencional, a ferramenta não executaria os demais casos de teste de uma sessão contra um mutante, caso o mesmo tenha sido anteriormente morto por um caso de teste.

5.3.2.1 Teste de Mutação sequencial x Teste de Mutação em paralelo

A Figura 11 mostra que independente do algoritmo de balanceamento de carga utilizado, mesmo para o cenário mais simples onde foram utilizados apenas quatro executores para processamento dos mutantes, a execução do Teste de Mutação em paralelo apresenta um grande *speedup* se comparada à execução do Teste de Mutação em uma execução sequencial convencional. O cenário que apresentou o menor *speedup* foi a utilização do algoritmo GTCOD, contudo, o mesmo ainda conseguiu melhorar a eficiência em 70,51% no experimento com relação à execução sequencial.

Na Tabela 2 é possível comparar o ganho percentual (% Dif) de eficiência para cada algoritmo. Os dados apresentados fazem um comparativo entre o tempo total gasto para a execução do experimento utilizando cada algoritmo de balanceamento utilizando quatro executores e o tempo gasto para a execução do experimento em uma abordagem sequencial. O algoritmo PEDRO apresentou o melhor resultado entre as abordagens garantindo uma eficiência de 78,22% em relação à execução do Teste de Mutação sequencial.

Figura 11 – Comparativo da média do tempo de execução de 4 executores em relação à execução sequencial do Teste de Mutação.

Tabela 2 – Média do tempo de execução para o experimento completo utilizando 4 executores.

Abordagem	4 Executores	Sequencial	% Dif
PEDRO	8865,40s	40695,49s	78,22%
GTCOD	11999,66s		70,51%
GMOD	9617,33s		76,37%
DTC	11889,48s		70,78%
DMBO	10774,08s		73,53%

Ao analisar as Figuras 12, 13, 14 e 15 é possível constatar que em todos os cenários a aplicação do Teste de Mutação em paralelo proporciona um grande aumento de eficiência em relação à execução sequencial. As figuras mostram gráficos de barra que fazem uma comparação do tempo gasto para a execução do Teste de Mutação utilizando a abordagem sequencial comparando à aplicação do Teste de Mutação em paralelo (Seção 4.2), porém, desta vez utilizando respectivamente: 8, 12, 16 e 32 executores para realizar o processamento dos mutantes.

Figura 12 – Comparativo da média do tempo de execução de 8 executores em relação à execução sequencial do Teste de Mutação.

Tabela 3 – Média do tempo de execução para o experimento completo utilizando 8 executores.

Abordagem	8 executores	Sequencial	% Dif
PEDRO	4712,47s	40695,49s	88,42%
GTCOD	7356,43s		81,92%
GMOD	5144,82s		87,36%
DTC	4957,37s		87,82%
DMBO	6759,50s		83,39%

As Tabelas 3, 4, 5 e 6 fazem o mesmo comparativo de cenários da aplicação do Teste de Mutação sequencial com a aplicação do Teste de Mutação em paralelo, porém, desta vez considerando a utilização de um maior número de executores na aplicação do teste em paralelo.

O cenário que apresentou os melhores resultados foi durante a aplicação do Teste de Mutação em paralelo utilizando o algoritmo de balanceamento de carga PEDRO que conseguiu reduzir em 95,66% o tempo de execução total do experimento ao utilizar 32 executores para processamento dos mutantes. O algoritmo GMOD apresentou resultados próximos ao algoritmo PEDRO e conseguiu reduzir o tempo de execução do experimento em 95,26%, seguido do algoritmo DMBO com 94,07%. Os algoritmos DTC e GTCOD apresentaram resultados inferiores, contudo, neste cenário ainda proporcionaram um grande *speedup*, registrando valores acima de 90% e 88% respectivamente.

Figura 13 – Comparativo da média do tempo de execução de 12 executores com relação à execução sequencial do Teste de Mutação.

Tabela 4 – Média do tempo de execução para o experimento completo utilizando 12 executores.

Abordagem	12 executores	Sequencial	% Dif
PEDRO	3444,75s	40695,49s	91,54%
GTCOD	6174,15s		84,83%
GMOD	3680,07s		90,96%
DTC	4101,37s		89,92%
DMBO	4898,29s		87,96%

Tabela 5 – Média do tempo de execução para o experimento completo utilizando 16 executores.

Abordagem	16 executores	Sequencial	% Dif
PEDRO	2711,74s	40695,49s	93,34%
GTCOD	5413,47s		86,70%
GMOD	2822,87s		93,06%
DTC	4219,26s		89,63%
DMBO	4395,83s		89,20%

Figura 14 – Comparativo da média do tempo de execução de 16 executores em relação à execução sequencial do Teste de Mutação.

Figura 15 – Comparativo da média do tempo de execução de 32 executores com relação à execução sequencial do Teste de Mutação.

Tabela 6 – Média do tempo de execução para o experimento completo utilizando 32 executores.

Abordagem	32 executores	Sequencial	% Dif
PEDRO	1764,90s	40695,49s	95,66%
GTCOD	4561,32s		88,79%
GMOD	1930,95s		95,26%
DTC	3739,07s		90,81%
DMBO	2412,50s		94,07%

É possível notar um comportamento padrão nos algoritmos utilizados, tal comportamento mostra que na medida em que o número de máquinas utilizadas para processamento foi

aumentado durante a execução do experimento, o tempo total, ou seja, o tempo necessário para executar todos os programas do experimento foi reduzindo. O gráfico de linhas apresentado na Figura 16 esclarece este comportamento. É possível notar que apenas em um único cenário houve um pequeno acréscimo ao tempo total de execução do experimento quando foi acrescentado o número de máquinas. O algoritmo DTC apresentou este comportamento, obtendo um melhor desempenho quando foi avaliado utilizando 12 executores. Ao ser avaliado com 16 executores a sua eficiência foi reduzida, por fim, ao ser executado com 32 executores o algoritmo voltou a ganhar desempenho atingindo seu melhor resultado.

Figura 16 – Redução do tempo de resposta à medida que o número de executores aumenta.

É importante lembrar que o comportamento demonstrado nesta subseção já era previsto, pois conforme mostrado anteriormente (Capítulo 3), espera-se que a utilização de arquiteturas complexas compostas de vários processadores possam melhorar o aproveitamento dos recursos computacionais disponíveis possibilitando um ganho extra na capacidade de processamento em troca de um gasto maior em infraestrutura. Contudo, a análise feita até aqui se fez necessária com intuito de validar a arquitetura e os algoritmos implementados e com a finalidade de testar a viabilidade da sua aplicação prática em um contexto real.

5.3.2.2 Avaliando o desempenho dos algoritmos de balanceamento de carga

O tempo de execução para cada um dos programas utilizados no experimento é apresentado para cada um dos algoritmos (DMBO, DTC, GMOD, GTCOD e PEDRO) nas Tabelas 7, 8,

Tabela 7 – Resultados individuais do experimento para o algoritmo DMBO

Programas	4 executores	8 executores	12 executores	16 executores	32 executores
boundedQueue	662,86s	649,62s	474,21s	605,22s	280,42s
cal	22,33s	18,78s	16,29s	16,66s	11,85s
Calculation	144,19s	82,39s	67,48s	49,66s	30,70s
checkIt	15,38s	22,65s	17,49s	19,31s	14,70s
CheckPalindrome	14,46s	32,78s	29,50s	27,70s	18,70s
countPositive	12,62s	12,73s	9,84s	10,50s	8,47s
date-plus	283,25s	186,19s	113,18s	108,04s	70,09s
DigitReverser	43,16s	36,66s	32,85s	27,67s	14,32s
findLast	6,90s	6,20s	7,37s	7,21s	5,55s
findVal	33,22s	34,93s	42,54s	24,32s	20,16s
Gaussian	208,39s	114,19s	82,57s	70,03s	40,05s
Heap	67,21s	51,36s	38,55s	35,52s	24,83s
InversePermutation	15,68s	13,11s	12,97s	11,33s	10,15s
jday-jdate	102,55s	82,13s	112,76s	74,82s	60,28s
lastZero	153,34s	112,98s	144,81s	97,35s	86,56s
LRS	64,49s	42,69s	38,25s	34,21s	18,99s
MergeSort	53,42s	33,27s	31,35s	24,45s	13,61s
numZero	154,34s	151,19s	85,26s	117,01s	65,93s
oddOrPos	12,82s	11,54s	10,94s	9,02s	10,26s
pcal	848,03s	478,82s	311,96s	270,49s	128,91s
power	39,00s	31,06s	22,50s	23,61s	13,67s
print_tokens	543,26s	320,04s	214,96s	180,00s	90,34s
print_tokens2	446,78s	301,75s	196,01s	237,71s	110,17s
printPrimes	88,83s	57,02s	39,19s	34,59s	20,49s
Queue	12,24s	8,40s	8,66s	7,85s	6,84s
quicksort	207,59s	119,12s	83,33s	65,46s	34,52s
RecursiveSelectionSort	39,56s	33,50s	22,73s	27,17s	15,23s
replace	4218,55s	2276,19s	1568,02s	1257,62s	604,35s
schedule	616,04s	339,83s	256,86s	184,89s	103,40s
schedule2	388,17s	211,51s	165,58s	128,19s	73,35s
Stack	51,44s	38,83s	28,71s	25,34s	22,90s
stats	48,94s	38,27s	44,18s	90,23s	75,08s
sum	39,37s	31,59s	26,90s	24,54s	17,07s
tcas	44,96s	29,60s	22,04s	21,30s	13,88s
testPad	103,60s	126,09s	97,25s	80,07s	57,99s
totinfo	699,78s	438,37s	286,35s	239,39s	134,80s
trashAndTakeOut	8,53s	8,94s	8,58s	7,79s	7,56s
twoPred	108,41s	93,27s	61,94s	71,40s	48,47s
UnixCal	150,40s	81,93s	64,32s	48,16s	27,88s

9, 10 e 11. O Quadro 3 mostra o algoritmo que obteve os melhores resultados para cada programa utilizado no experimento. Para o cenário onde foram utilizados 4 executores para processamento dos mutantes, o algoritmo PEDRO obteve os melhores resultados, apresentando superioridade em 13 dos 39 programas avaliados, seguido dos algoritmos DTC e GMOD que mostraram ambos serem a melhor abordagem para 9 programas do experimento. O algoritmo DMBO mostrou ser a melhor abordagem em 5 programas, seguido, por fim, do algoritmo GTCOD que mostrou ser a melhor alternativa para 3 dos programas analisados com esta configuração de máquinas.

Quando o número de executores foi aumentado para 8, o algoritmo PEDRO obteve novamente os melhores resultados, dessa vez para 18 programas analisados do experimento,

Tabela 8 – Resultados individuais do experimento para o algoritmo DTC

Programass	4 executores	8 executores	12 executores	16 executores	32 executores
boundedQueue	1442,85s	295,25s	304,01s	316,93s	385,95s
cal	18,08s	22,47s	24,76s	25,47s	24,07s
Calculation	134,97s	86,34s	85,76s	83,68s	85,22s
checkIt	24,10s	19,67s	20,84s	20,38s	19,74s
CheckPalindrome	42,69s	117,63s	119,37s	120,68s	115,17s
countPositive	29,38s	14,57s	14,33s	15,49s	16,04s
date-plus	344,30s	86,13s	86,83s	89,82s	103,52s
DigitReverser	90,76s	54,68s	55,32s	59,27s	59,80s
findLast	34,59s	6,79s	7,33s	8,29s	8,12s
findVal	29,92s	78,23s	78,84s	84,93s	86,51s
Gaussian	230,13s	66,45s	67,00s	70,41s	80,64s
Heap	31,59s	35,01s	44,52s	43,45s	38,82s
InversePermutation	27,42s	11,39s	12,20s	12,83s	12,25s
jday-jdate	122,12s	39,57s	38,74s	43,69s	54,22s
lastZero	42,96s	60,47s	59,97s	61,23s	61,12s
LRS	51,55s	42,34s	74,90s	76,19s	82,65s
MergeSort	37,82s	35,66s	35,62s	36,98s	38,72s
numZero	190,98s	180,67s	179,29s	184,30s	178,46s
oddOrPos	40,06s	9,98s	10,05s	11,07s	10,18s
pcal	870,10s	179,53s	185,02s	190,11s	190,38s
power	27,76s	23,73s	23,46s	26,31s	29,03s
print_tokens	535,25s	127,72s	141,46s	135,74s	161,64s
print_tokens2	581,69s	178,85s	184,51s	183,30s	208,35s
printPrimes	66,72s	86,96s	86,83s	86,30s	84,85s
Queue	30,96s	7,44s	8,36s	9,25s	9,15s
quicksort	131,62s	123,25s	122,54s	123,82s	118,92s
RecursiveSelectionSort	62,68s	45,11s	46,00s	46,13s	46,46s
replace	4376,31s	2314,39s	1315,62s	1326,24s	566,62s
schedule	571,10s	79,52s	81,45s	83,56s	84,11s
schedule2	428,38s	53,37s	52,83s	56,65s	68,80s
Stack	65,48s	97,66s	98,27s	134,31s	129,03s
stats	36,43s	43,85s	50,04s	61,01s	131,92s
sum	54,15s	37,06s	37,56s	37,82s	38,15s
tcas	24,36s	11,82s	12,68s	14,05s	14,55s
testPad	83,61s	67,23s	63,64s	65,53s	103,11s
totinfo	717,57s	138,61s	192,02s	183,97s	203,78s
trashAndTakeOut	9,18s	7,64s	8,28s	9,17s	9,38s
twoPred	18,96s	5,98s	5,96s	14,43s	13,67s
UnixCal	230,91s	64,37s	65,16s	66,47s	65,98s

seguido do algoritmo DTC com desempenho superior em 14 programas. O algoritmo GMOD mostrou ser a melhor abordagem em 4 programas utilizados no experimento e, por fim, o algoritmo DMBO apresentou ser a melhor abordagem para 3 programas.

Para o cenário onde foram utilizados 12 executores para processamento dos mutantes, o algoritmo PEDRO se mostrou como melhor alternativa, obtendo melhor resultado em 23 dos 39 programas analisados, seguido do algoritmo DTC que conseguiu ser a melhor abordagem em 10 programas, por fim, os algoritmos DMBO e GMOD apresentaram ser a melhor abordagem a utilizada em 3 dos programas utilizados no experimento. Assim como na configuração anterior, os resultados positivos do algoritmo DTC tendem a ser justificados por conta do modelo de

Tabela 9 – Resultados individuais do experimento para o algoritmo GMOD

Programass	4 executores	8 executores	12 executores	16 executores	32 executores
boundedQueue	62,25s	52,41s	47,21s	51,63s	43,79s
cal	29,53s	20,32s	16,53s	12,78s	12,83s
Calculation	150,40s	79,28s	58,03s	43,60s	30,20s
checkIt	9,17s	8,81s	8,69s	8,36s	9,89s
CheckPalindrome	11,38s	12,92s	11,74s	10,02s	12,64s
countPositive	12,45s	11,88s	11,08s	9,70s	10,19s
date-plus	255,31s	150,36s	100,07s	68,89s	56,30s
DigitReverser	43,84s	26,84s	21,72s	18,50s	15,10s
findLast	11,43s	9,16s	9,32s	9,37s	9,69s
findVal	17,06s	16,03s	15,21s	14,24s	14,49s
Gaussian	202,92s	107,72s	77,55s	58,91s	38,37s
Heap	71,14s	42,27s	33,30s	26,55s	20,51s
InversePermutation	25,31s	18,35s	15,89s	14,36s	12,86s
jday-jdate	69,56s	43,77s	35,75s	31,29s	26,60s
lastZero	26,58s	24,97s	25,05s	23,92s	28,67s
LRS	72,62s	41,95s	31,90s	27,33s	19,84s
MergeSort	66,97s	37,38s	28,43s	22,66s	16,54s
numZero	21,70s	19,86s	20,90s	18,41s	18,43s
oddOrPos	18,32s	14,12s	12,80s	11,35s	11,36s
pcal	895,28s	459,64s	314,94s	238,81s	131,86s
power	34,64s	21,61s	16,96s	14,50s	10,88s
print_tokens	489,84s	254,56s	180,12s	142,10s	83,56s
print_tokens2	476,34s	241,60s	175,37s	133,21s	80,08s
printPrimes	91,23s	50,12s	36,79s	29,51s	19,47s
Queue	20,87s	13,54s	11,23s	10,52s	9,30s
quicksort	218,68s	114,40s	79,52s	61,04s	35,37s
RecursiveSelectionSort	31,34s	20,19s	16,30s	14,39s	12,88s
replace	4055,53s	2046,87s	1380,89s	1034,71s	542,48s
schedule	579,60s	300,99s	209,22s	158,95s	97,67s
schedule2	398,16s	207,23s	144,39s	107,20s	66,04s
Stack	35,43s	23,49s	19,03s	15,18s	13,27s
stats	55,84s	58,91s	83,97s	46,23s	180,55s
sum	11,90s	10,05s	15,33s	12,55s	15,57s
tcas	79,77s	43,88s	32,43s	25,18s	17,44s
testPad	37,18s	27,87s	27,94s	18,89s	22,79s
totinfo	682,37s	350,38s	241,96s	184,42s	108,27s
trashAndTakeOut	17,61s	13,19s	11,46s	10,70s	9,36s
twoPred	15,94s	37,01s	19,41s	21,52s	24,73s
UnixCal	211,84s	110,92s	81,67s	61,41s	41,09s

sessão de testes utilizado no experimento. Esta configuração com o número de 12 executores se mostrou a configuração ideal para utilização do algoritmo DTC, pois foi a configuração na qual o algoritmo atingiu o seu melhor desempenho.

Ao aumentar o número de executores para 16, o algoritmo PEDRO volta a ser o mais eficiente e mostra ser a abordagem com melhor desempenho em 24 programas do experimento, seguido pelo algoritmo DTC com 7 resultados, o algoritmo DMBO com melhor resultado em 6 programas e por fim os algoritmos GMOD e GTCOD se mostrando ser superior em 1 caso cada um.

Por fim, utilizando 32 executores, a maior configuração utilizada para processamento

Tabela 10 – Resultados individuais do experimento para o algoritmo GTCOD.

Programas	4 executores	8 executores	12 executores	16 executores	32 executores
boundedQueue	288,90s	298,39s	378,98s	324,79s	305,77s
cal	35,46s	26,68s	28,99s	28,79s	29,45s
Calculation	228,50s	94,18s	97,19s	94,44s	110,93s
checkIt	79,61s	21,14s	22,38s	21,76s	72,38s
CheckPalindrome	98,26s	103,81s	171,72s	123,52s	513,18s
countPositive	16,16s	17,36s	19,17s	19,08s	14,47s
date-plus	280,83s	228,92s	140,35s	178,88s	104,84s
DigitReverser	337,00s	49,83s	52,42s	59,88s	162,04s
findLast	55,36s	9,22s	9,49s	9,38s	57,90s
findVal	127,04s	58,53s	81,19s	89,82s	114,11s
Gaussian	188,66s	138,11s	87,76s	106,76s	92,02s
Heap	66,16s	43,36s	44,32s	43,05s	69,22s
InversePermutation	22,33s	15,85s	16,05s	16,10s	18,67s
jday-jdate	107,25s	103,20s	96,43s	83,78s	59,71s
lastZero	69,39s	82,43s	74,70s	97,43s	70,35s
LRS	105,83s	76,64s	82,94s	83,47s	76,87s
MergeSort	288,80s	51,48s	41,95s	42,44s	103,47s
numZero	193,59s	175,94s	194,20s	173,99s	180,73s
oddOrPos	16,63s	13,18s	13,55s	13,13s	15,40s
pcal	682,18s	569,19s	450,07s	366,23s	160,54s
power	34,61s	28,27s	29,20s	30,70s	21,51s
print_tokens	352,35s	385,86s	294,27s	259,74s	144,97s
print_tokens2	396,43s	385,91s	289,17s	280,31s	115,35s
printPrimes	92,95s	87,42s	89,49s	85,71s	47,80s
Queue	34,75s	13,76s	10,40s	10,59s	24,12s
quicksort	559,32s	153,90s	170,45s	125,35s	162,64s
RecursiveSelectionSort	95,84s	49,02s	49,86s	48,30s	70,82s
replace	4490,86s	2337,77s	1741,46s	1340,29s	387,26s
schedule	384,63s	348,25s	254,71s	225,52s	109,02s
schedule2	277,85s	255,34s	187,47s	149,25s	53,36s
Stack	133,51s	121,40s	114,65s	119,54s	113,54s
stats	186,58s	47,00s	61,06s	62,66s	244,35s
sum	245,60s	50,81s	49,90s	59,90s	164,27s
tcas	351,07s	61,98s	43,58s	36,31s	84,29s
testPad	101,80s	89,09s	88,88s	98,51s	95,13s
totinfo	531,53s	558,96s	428,93s	338,62s	148,86s
trashAndTakeOut	67,31s	12,62s	11,65s	11,32s	48,74s
twoPred	158,08s	33,06s	36,75s	43,50s	66,56s
UnixCal	216,63s	158,56s	118,41s	110,65s	126,69s

dos mutantes no experimento, o algoritmo PEDRO conseguiu novamente o melhor desempenho, apresentando os melhores resultados em 29 dos 39 programas avaliados, o algoritmo DMBO obteve 5 resultados entre os programas analisados, o algoritmo DTC teve 4 resultados e, por fim, o algoritmo GMOD, obteve melhor resultado em 1 dos programas.

Analisando os resultados apresentados até o momento, é possível constatar que PEDRO, GMOD e DMBO foram os algoritmos que obtiveram os melhores resultados de maneira geral. Isto confirma as afirmações realizadas anteriormente (Seção 4.1) e conclui que de fato estes algoritmos possuem características bastante similares.

Ao analisar os resultados, percebe-se que quanto menor o número de máquinas disponível

Tabela 11 – Resultados individuais do experimento para o algoritmo PEDRO.

Programas	4 executores	8 executores	12 executores	16 executores	32 executores
boundedQueue	43,97s	54,57s	57,26s	42,14s	42,53s
cal	31,19s	12,78s	13,10s	12,20s	11,10s
Calculation	150,96s	73,33s	54,94s	43,36s	27,07s
checkIt	5,09s	6,26s	5,99s	7,25s	7,70s
CheckPalindrome	9,94s	8,71s	9,85s	9,94s	10,01s
countPositive	14,99s	9,08s	8,81s	7,92s	8,31s
date-plus	240,41s	146,72s	86,86s	75,85s	53,66s
DigitReverser	36,29s	22,69s	20,55s	18,08s	13,06s
findLast	7,56s	6,48s	7,49s	7,45s	6,84s
findVal	12,18s	11,84s	12,77s	12,14s	11,86s
Gaussian	196,00s	98,01s	72,65s	58,41s	34,71s
Heap	59,99s	33,85s	29,76s	24,07s	19,23s
InversePermutation	16,65s	12,73s	13,09s	12,23s	10,92s
jday-jdate	44,77s	28,29s	27,19s	23,31s	24,93s
lastZero	25,21s	21,09s	22,35s	21,16s	22,85s
LRS	57,97s	34,05s	28,02s	24,15s	18,19s
MergeSort	50,95s	28,72s	23,47s	20,72s	14,64s
numZero	28,25s	22,09s	24,71s	17,07s	15,94s
oddOrPos	13,37s	10,21s	10,91s	10,44s	9,37s
pcal	788,96s	404,41s	282,65s	214,50s	125,71s
power	30,17s	18,15s	14,27s	12,13s	9,25s
print_tokens	470,44s	241,97s	196,18s	139,95s	78,75s
print_tokens2	438,85s	212,64s	161,60s	133,52s	77,15s
printPrimes	81,32s	45,11s	33,59s	27,65s	17,72s
Queue	13,03s	9,45s	8,90s	8,41s	6,91s
quicksort	203,42s	106,45s	75,49s	59,20s	34,23s
RecursiveSelectionSort	28,32s	15,44s	14,32s	13,23s	11,11s
replace	3810,92s	1932,14s	1319,99s	993,75s	526,21s
schedule	546,47s	281,18s	200,35s	154,87s	90,65s
schedule2	365,53s	188,35s	133,03s	103,99s	64,04s
Stack	27,95s	19,02s	16,64s	14,80s	11,78s
stats	43,42s	41,24s	49,33s	59,73s	134,22s
sum	28,61s	23,99s	8,88s	11,19s	12,96s
tcas	56,77s	28,12s	24,20s	19,23s	15,86s
testPad	73,95s	47,21s	55,31s	45,33s	20,43s
totinfo	611,86s	308,18s	238,05s	171,88s	97,37s
trashAndTakeOut	10,06s	8,43s	8,82s	8,78s	7,54s
twoPred	41,70s	56,89s	17,49s	23,24s	22,46s
UnixCal	147,92s	82,58s	55,90s	48,46s	37,65s

Quadro 3 – Algoritmo com melhor desempenho para cada objeto de estudo utilizado no experimento.

Programas	4 executores	8 executores	12 executores	16 executores	32 executores
boundedQueue	PEDRO	GMOD	GMOD	PEDRO	PEDRO
cal	DTC	PEDRO	PEDRO	PEDRO	PEDRO
Calculation	DTC	PEDRO	PEDRO	PEDRO	PEDRO
checkIt	PEDRO	PEDRO	PEDRO	PEDRO	PEDRO
CheckPalindrome	PEDRO	PEDRO	PEDRO	PEDRO	PEDRO
countPositive	GMOD	PEDRO	PEDRO	PEDRO	PEDRO
date-plus	PEDRO	DTC	PEDRO	GMOD	PEDRO
DigitReverser	PEDRO	PEDRO	PEDRO	PEDRO	PEDRO
findLast	DMBO	DMBO	DMBO	DMBO	DMBO
findVal	PEDRO	PEDRO	PEDRO	PEDRO	PEDRO
Gaussian	GMOD	DTC	DTC	PEDRO	PEDRO
Heap	DTC	PEDRO	PEDRO	PEDRO	PEDRO
InversePermutation	DMBO	DTC	DMBO	DMBO	DMBO
jday-jdate	PEDRO	PEDRO	PEDRO	PEDRO	PEDRO
lastZero	PEDRO	PEDRO	PEDRO	PEDRO	PEDRO
LRS	DTC	PEDRO	PEDRO	PEDRO	PEDRO
MergeSort	DTC	PEDRO	PEDRO	PEDRO	DMBO
numZero	GMOD	GMOD	GMOD	PEDRO	PEDRO
oddOrPos	DMBO	DTC	PEDRO	DMBO	PEDRO
pcal	GMOD	DTC	DTC	DTC	PEDRO
power	PEDRO	PEDRO	PEDRO	PEDRO	PEDRO
print_tokens	GTCOD	DTC	DTC	DTC	PEDRO
print_tokens2	GTCOD	DTC	PEDRO	GMOD	PEDRO
printPrimes	DTC	PEDRO	PEDRO	PEDRO	PEDRO
Queue	DMBO	DMBO	DTC	DMBO	DMBO
quicksort	DTC	PEDRO	PEDRO	PEDRO	PEDRO
RecursiveSelectionSort	PEDRO	PEDRO	PEDRO	PEDRO	PEDRO
replace	PEDRO	PEDRO	PEDRO	PEDRO	PEDRO
schedule	GMOD	DTC	DTC	DTC	DTC
schedule2	GMOD	DTC	DTC	DTC	GTCOD
Stack	PEDRO	PEDRO	PEDRO	PEDRO	PEDRO
stats	DTC	DMBO	DMBO	GMOD	DMBO
sum	GMOD	GMOD	PEDRO	PEDRO	PEDRO
tcas	DTC	DTC	DTC	DTC	DMBO
testPad	GMOD	GMOD	GMOD	GMOD	PEDRO
totinfo	GTCOD	DTC	DTC	PEDRO	PEDRO
trashAndTakeOut	DMBO	DTC	DTC	DMBO	PEDRO
twoPred	GMOD	DTC	DTC	DTC	DTC
UnixCal	PEDRO	DTC	PEDRO	DMBO	DMBO

para processo, menor a eficiência dos algoritmos. Pode-se justificar este comportamento por conta do tempo que é gasto em cada algoritmo esperando dados do escalonador enquanto os executores estão ocupados processando suas porções de trabalho. Os algoritmos que obtiveram os piores resultados foram DTC e GTCOD, mesmo DTC possuindo alguns bons resultados individuais, conforme mostrado no Quadro 3.

Estes resultados justificam-se em geral por conta do baixo número de casos de teste utilizados no experimento. Neste cenário, os algoritmos DTC e GTCOD são ligeiramente favorecidos por conta do tipo da sessão de testes utilizada na ferramenta *Proteum*. Utilizar a sessão de testes em modo *research*, como foi utilizado durante este experimento, acaba contribuindo para que o desempenho dos algoritmos que distribuem mutantes (DMBO, GMOD, PEDRO) seja afetado, pelo fato de obrigá-los a executar mutantes desnecessariamente.

Conforme mencionado anteriormente, existe ainda o fator de não haver comunicação entre as máquinas responsáveis por processar os mutantes, causando o problema de um mutante ser morto diversas vezes desnecessariamente. Contudo, este não se aplica neste contexto pelo fato de o experimento ter sido rodado com objetivo de estressar ao máximo a sessão de testes, logo todos os mutantes foram obrigatoriamente executados contra todos os casos de teste. Em um cenário convencional, a desconsideração deste fator contribuiria para uma redução no tempo de execução dos mutantes e consequentemente na aplicação de todos os algoritmos.

5.3.2.3 Testando hipóteses

Sabendo que alguns testes estatísticos somente aplicam-se a populações que seguem uma distribuição de dados normal, antes de escolher o teste estatístico a ser aplicado, é necessário que seja examinado se os dados coletados seguem uma linearidade e podem ser considerados como uma distribuição normal ou não. Para tal, são apresentados gráficos *Quantile to Quantile* (Q-Q) (DODGE, 2008) nas Figuras 17,18,19,20,21 que confirmam que os dados coletados seguem uma distribuição normal. Além da representação por meio dos gráficos, a normalidade dos dados foi medida por meio do teste de *Shapiro-Wilk*.

Neste primeiro momento serão considerados os dados referentes ao tempo total gasto na execução do experimento. O experimento consiste na execução de todos os mutantes de cada programa utilizado como objeto de estudo do experimento com seus respectivos conjuntos de casos de teste.

O teste de *Shapiro-Wilk* tem como objetivo analisar se o conjunto de dados é estatisticamente diferente de uma distribuição normal. Como hipótese nula (H_0), é definido que os dados analisados não são estatisticamente diferentes de uma distribuição normal, ou seja, assume-se a normalidade dos dados analisados. Como hipótese alternativa, garante que os dados são estatisticamente diferentes de uma distribuição normal, ou seja, assume-se então a não normalidade dos dados.

Figura 17 – Gráficos de probabilidade normal dos tempos de execução para cada algoritmo utilizando 4 executores.

Figura 18 – Gráficos de probabilidade normal dos tempos de execução para cada algoritmo utilizando 8 executores.

De acordo com o teste, um grau de significância $p < 0.05$ sugere que a população não segue uma distribuição normal, enquanto um valor de $p > 0.05$ indica que há evidência de que os dados seguem uma distribuição normal (JOHNSON, 2009). Os resultados na Tabela 12 sugerem que os dados coletados seguem uma distribuição normal e portanto asseguram a aplicação de testes paramétricos para a avaliação estatística.

O *T-Test* é um teste estatístico utilizado para checar se duas médias são confiantemente diferentes uma da outra. Este tipo de afirmação poderia ser realizada olhando diretamente para o valor da média dos resultados, contudo, apesar de demonstrar uma diferença não é possível inferir se essa diferença é de fato confiável.

Assim, foi realizado o *T-Test* Pareado com objetivo de comparar a diferenças entre duas

Figura 19 – Gráficos de probabilidade normal dos tempos de execução para cada algoritmo utilizando 12 executores.

Figura 20 – Gráficos de probabilidade normal dos tempos de execução para cada algoritmo utilizando 16 executores.

Tabela 12 – Valor de significância *p-value* para o teste de *Shapiro-Wilk*.

Teste de <i>Shapiro-Wilk</i> para normalidade (p-value)						
Experimento	Tratamento					
	DMBO	DTC	GMOD	GTCOD	PEDRO	SERIAL
4 executores	0.407563	0.157805	0.069689	0.212271	0.313525	0.058038
8 executores	0.746409	0.969443	0.605938	0.634636	0.646489	
12 executores	0.965744	0.904120	0.435913	0.656024	0.509029	
16 executores	0.257434	0.161261	0.141310	0.251307	0.909989	
32 executores	0.121804	0.152438	0.363226	0.344847	0.080142	

amostras. Neste caso, foram analisados os resultados para cada um dos cinco algoritmos de balanceamento de carga com a execução sequencial do experimento. A análise do *T-Test* foi

Figura 21 – Gráficos de probabilidade normal dos tempos de execução para cada algoritmo utilizando 32 executores.

estendida para comparar os resultados da execução sequencial com os cinco cenários diferentes explorados durante o experimento, nos quais foram utilizados respectivamente 4, 8, 12, 16 e 32 executores.

Tabela 13 – Resultados do *T-Test* para 4 executores.

4 executores		
Abordagem	<i>t</i>	<i>p</i>
DMBO	24.2003071782	8.00197294613e-13
DTC	22.3330603107	2.39796971181e-12
GMOD	25.0657597006	4.94398351903e-13
GTCOD	22.7664641177	1.84466466257e-12
PEDRO	26.0627678413	2.89459765779e-13

Tabela 14 – Resultados do *T-Test* para 8 executores.

8 executores		
Abordagem	<i>t</i>	<i>p</i>
DMBO	27.4044156278	1.45174928656e-13
DTC	28.8805263639	7.04869292927e-14
GMOD	28.7131007899	7.63674747707e-14
GTCOD	26.917341736	1.85790971006e-13
PEDRO	29.1104821335	6.31871874018e-14

De acordo com uma análise das Tabelas 13, 14, 15, 16 e 17, como o valor absoluto para todos os valores de *p* são menores que a margem de erro (0,01%), que corresponde ao complemento do nível de significância estabelecido de 99,99%. Pode-se estatisticamente, rejeitar a hipótese nula (H_0) para a primeira questão de pesquisa (QP_1), que defendia que a aplicação do Teste de Mutação em um ambiente sequencial é equivalente à utilização das técnicas de

Tabela 15 – Resultados do *T-Test* para 12 executores.

12 executores		
Abordagem	<i>t</i>	<i>p</i>
DMBO	28.9434130052	6.84050117246e-14
DTC	29.5264090261	5.19598614833e-14
GMOD	29.9439306739	4.28096138773e-14
GTCOD	28.232828094	9.63464684604e-14
PEDRO	30.2257519392	3.76182430045e-14

Tabela 16 – Resultados do *T-Test* para 16 executores.

16 executores		
Abordagem	<i>t</i>	<i>p</i>
DMBO	29.3801565686	5.56429002848e-14
DTC	29.4774739097	5.31622612136e-14
GMOD	30.5333283051	3.27116550982e-14
GTCOD	28.6015551963	8.05754522442e-14
PEDRO	30.734461427	2.98766755035e-14

Tabela 17 – Resultados do *T-Test* para 32 executores.

32 executores		
Abordagem	<i>t</i>	<i>p</i>
DMBO	30.2792467055	3.6711239437e-14
DTC	29.9166882487	4.33506453386e-14
GMOD	31.3481858539	2.27369102401e-14
GTCOD	29.7268733587	4.73296363912e-14
PEDRO	31.4817641031	2.14396373338e-14

paralelismo implementadas. Como todo valor *t* é positivo, os indícios sobre a hipótese alternativa (H_1) são reforçados, ou seja, há uma diferença significativa no tempo de execução gasto na aplicação do Teste de Mutação utilizando uma ferramenta que possibilite a execução do Teste de Mutação em paralelo em relação à versão sequencial.

Para responder a primeira questão de pesquisa (QP_1) foi necessário realizar uma análise entre pares de dois grupos entre a abordagem sequencial e cada um dos algoritmos avaliados para cada uma das configurações exploradas no experimento. Para verificar a segunda questão de pesquisa definida (QP_2), é necessário realizar um comparativo entre mais de dois grupos, neste caso cinco grupos, referentes a cada um dos algoritmos paralelos implementados. Para realizar esta análise não é possível a utilização do *T-Test*, contudo, é possível utilizar outro teste estatístico que considere a avaliação de mais de dois grupos. Nesta situação, para dados que segue uma distribuição normal, o teste estatístico mais adequado a ser utilizado é o *Analysis of Variance ANOVA-Test*.

A hipótese nula (H_0) para o *ANOVA-Test* defende que as médias para as amostras analisadas são iguais. Então, caso o resultado do *one-way ANOVA* produzir resultados significativos, ou

seja, rejeitar H_0 , é possível ter indícios a respeito da hipótese alternativa, (H_1), que defende que os dados analisados não são iguais. Então, para determinar se existe uma diferença estatística entre as médias do tempo de execução de cada algoritmo para cada um dos cenários explorados foi aplicado o *ANOVA-Test* e os resultados são apresentados na Tabela 18 abaixo.

Tabela 18 – Resultados do teste estatístico *ANOVA-Test*.

<i>ANOVA-Test</i>		
Arquitetura	<i>f</i>	<i>p</i>
4 executores	348.402628352	2.10941538406e-45
8 executores	11164.5412797	2.31065291928e-97
12 executores	2018.40153744	1.78855633482e-71
16 executores	12450.0944095	5.12445691704e-99
32 executores	812.700820959	7.6248427166e-58

De acordo com a análise da Tabela 18, como o valor absoluto para todos os valores de p são menores que a margem de erro (0,01%), que corresponde ao complemento do nível de significância estabelecido de 99,99%, estatisticamente, pode-se rejeitar a hipótese nula (H_0) para a segunda questão de pesquisa (Q_2), que defendia que tempo de execução gasto por cada algoritmo é bastante similar, justificando a utilização de qualquer um dos algoritmo. Assim, temos indícios a respeito da hipótese alternativa (H_1) que defende que o tempo de execução varia entre cada algoritmo utilizado, portanto, o algoritmo com os melhores resultados deve ser utilizado.

Com base nos resultados apresentados durante a análise descritiva (subseção 5.3.2), é possível observar que os algoritmos que apresentaram melhor desempenho no experimento global foram os algoritmos PEDRO e GMOD com resultado bem próximos, contudo, conforme o teste de hipótese para a segunda questão de pesquisa (QP_2) evidenciou, o tempo de execução varia para cada algoritmo utilizado, então se julga necessário determinar qual o algoritmo que melhor se adequa à utilização. Portanto, como a hipótese nula para a segunda questão de pesquisa foi rejeitada, faz-se necessário analisar o quão diferente são os grupos analisados. O *ANOVA-test* informa se há uma diferença global significativa entre os grupos analisados, porém, não é capaz de informar como os grupos diferem entre si, contudo, para obter essas informações é possível a aplicação de *Post Hoc Tests*.

Post Hoc Tests são executados com objetivo de confirmar onde as diferenças ocorreram entre os grupos. Assim, sua utilização só é recomendada quando as amostras analisadas mostram uma diferença global significativa. Ou seja, quando há um resultado que rejeita a hipótese nula nos resultados do *ANOVA-test*. Assim, foi adotado o *TUKEY HSD-Test* (DODGE, 2008) para indicar quais pares do tratamento são estatisticamente diferentes entre si.

As Tabelas 19, 20, 21 mostram os resultados para a aplicação do *TUKEY HSD-Test*, adotando um intervalo de confiança de 95%, para as configurações do experimento com 4, 8, 12, 16 e 32 executores respectivamente. Por meio das tabelas é possível verificar a diferença entre os

grupos analisados. Ao realizar uma análise de pares é então possível verificar a diferença entre os grupos.

Tabela 19 – Resultados para teste *TUKEY HSD-Test* para configurações com 4 e 8 executores.

DMBO subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
DTC	1115.4	82253345	14082666	<0.0001
GMOD	-11567493	-14496159	-86388278	<0.0001
GTCOD	1225576	93270945	15184426	<0.0001
PEDRO	-19086827	-22015492	-16158161	<0.0001
DTC subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
GMOD	-22721493	-25650159	-19792828	<0.0001
GTCOD	110176	-18269055	40304255	0.8294
PEDRO	-30240827	-33169492	-27312161	<0.0001
GMOD subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
GTCOD	23823253	20894588	26751919	<0.0001
PEDRO	-75193333	-10447999	-45906678	<0.0001
GTCOD subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
PEDRO	-31342587	-34271252	-28413921	<0.0001

DMBO subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
DTC	-18021227	-18466317	-17576137	<0.0001
GMOD	-16146767	-16591857	-15701677	<0.0001
GTCOD	596932	552423	641441	<0.0001
PEDRO	-2036492	-2081001	-1991983	<0.0001
DTC subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
GMOD	187446	142937	231955	<0.0001
GTCOD	23990547	23545457	24435637	<0.0001
PEDRO	-23436933	-27887833	-18986034	<0.0001
GMOD subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
GTCOD	22116087	21670997	22561177	<0.0001
PEDRO	-42181533	-46632433	-37730634	<0.0001
GTCOD subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
PEDRO	-2633424	-2677933	-2588915	<0.0001

Tabela 20 – Resultados para teste *TUKEY HSD-Test* para configurações com 12 e 16 executores.

DMBO subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
DTC	-796918	-89445619	-69937981	<0.0001
GMOD	-1218214	-13157522	-11206758	<0.0001
GTCOD	12758653	11783271	13734035	<0.0001
PEDRO	-14535393	-15510775	-13560011	<0.0001
DTC subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
GMOD	-421296	-51883419	-32375781	<0.0001
GTCOD	20727833	19752451	21703215	<0.0001
PEDRO	-65662133	-75415953	-55908314	<0.0001
GMOD subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
GTCOD	24940793	23965411	25916175	<0.0001
PEDRO	-23532533	-33286353	-13778714	<0.0001
GTCOD subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
PEDRO	-27294047	-28269429	-26318665	<0.0001

DMBO subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
DTC	-176568	-21722694	-13590906	<0.0001
GMOD	-1572954	-16136129	-15322951	<0.0001
GTCOD	1030.98	99032106	10716389	<0.0001
PEDRO	-1684086	-17247449	-16434271	<0.0001
DTC subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
GMOD	-1396386	-14370449	-13557271	<0.0001
GTCOD	1207548	11668891	12482069	<0.0001
PEDRO	-1507518	-15481769	-14668591	<0.0001
GMOD subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
GTCOD	2603934	25632751	26445929	<0.0001
PEDRO	-111132	-15179094	-70473063	<0.0001
GTCOD subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
PEDRO	-2715066	-27557249	-26744071	<0.0001

Para todos os valores *p-value* inferiores à margem de erro (0,5%), correspondente ao complemento do nível de significância estabelecido (95%) é possível verificar a diferença entre os grupos. Ao analisar a Tabela 19, correspondente ao resultados para a execução do experimento utilizando 4 executores é possível constatar com segurança que o algoritmo PEDRO obteve melhores resultados comparado ao demais algoritmos analisados (DMBO, DTC, GMOD, GTCOD).

A mesma análise pode ser utilizada com objetivo de verificar o desempenho dos demais algoritmos entre si. Por exemplo, ao analisar o algoritmo DMBO é possível verificar por meio da diferença que o mesmo obteve melhores resultados em relação aos algoritmos DTC e GTCOD e em contrapartida obteve resultados piores aos algoritmos GMOD e PEDRO.

Tabela 21 – Resultados para teste *TUKEY HSD-Test* para configuração com 32 executores.

DMBO subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
DTC	13265673	11551991	14979356	<0.0001
GMOD	-563034	-73440227	-39166573	<0.0001
GTCOD	2148818	19774497	23201863	<0.0001
PEDRO	-64759733	-81896561	-47622906	<0.0001
DTC subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
GMOD	-18896013	-20609696	-17182331	<0.0001
GTCOD	82225067	65088239	99361894	<0.0001
PEDRO	-19741647	-21455329	-18027964	<0.0001
GMOD subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
GTCOD	2711852	25404837	28832203	<0.0001
PEDRO	-84563333	-25593161	86804942	0.6412
GTCOD subtraído de				
	Diferença	Inferior	Superior	<i>P-value</i>
PEDRO	-27964153	-29677836	-26250471	<0.0001

Figura 22 – *Boxplot* para comparação da diferença entre grupos (I)

Para facilitar a visualização dessas informações foram plotados gráficos *boxplot*. As figuras 22, 23, 24 apresentam os gráficos *boxplot* em que é possível observar de maneira clara os resultados referentes às análises realizadas por meio do *TUKEY HSD-Test*.

De posse dessas informações também é possível observar os resultados em que não se pode ser totalmente conclusivo a respeito de qual abordagem é a mais eficiente. Por exemplo, ao observar a tabela 19, para o algoritmo DTC, ao ser comparado com o algoritmo GTCOD o valor de *p-value* (0.8294) obteve resultado superior ao à margem de erro (0,5%). Neste cenário não é possível determinar qual a abordagem mais eficiente, já que não há uma diferença significativa entre o algoritmo DTC em relação ao algoritmo GTCOD.

Figura 23 – *Boxplot* para comparação da diferença entre grupos (II)

Figura 24 – *Boxplot* para comparação da diferença entre grupos utilizando 32 executores.

Essa relação também pode ser observada ao visualizar os resultados da Figura 22, é possível observar por meio do *boxplot* que há uma intercessão nos valores entre os algoritmos DTC e GTCOD. Contudo, vale ressaltar que fazer a análise dessas informações por si não garante que não haja diferença significativa entre os valores, ela apenas comprova que não foi possível detectar tal diferença ao realizar o *Tukey HSD-Test* com o grau de significância adotado.

5.3.3 Ameaças à validade

A falta de representatividade dos programas utilizados no experimento pode ser uma ameaça à sua validade. Infelizmente, este é um problema que sempre afeta pesquisas

relacionadas à área de engenharia de software devido ao fato de não existir teoria suficientemente embasada que possa garantir que um determinado conjunto de programas seja considerado uma amostra representativa para experimentação. A fim de mitigar este risco, procurou utilizar-se de um conjunto de programas que possibilitassem cobrir o maior número de aplicações possíveis. Além de ter tentado estabelecer uma visão considerando não apenas os programas utilizados no experimento de maneira individual, mas também considerando todos os programas em uma análise geral, a fim de tentar simular a condição de execução de um software de grandes proporções em termos de linhas de código.

Com relação às ameaças à validade por construção, pode-se destacar possíveis enganos na implementação tanto da arquitetura proposta como dos algoritmos analisados. Para mitigar esta ameaça foram conduzidos diversos testes entre os módulos durante o período de construção a fim de tentar garantir que durante a execução do experimento a implementação em paralelo fluísse de maneira mais correta possível. Com relação à ferramenta *Proteum*, a mesma tem sido amplamente utilizada nos meios acadêmicos durante anos, assim pode-se descartar esta ameaça.

As ameaças à validade interna definem o grau de confiança entre os resultados esperados e os resultados obtidos. Todas as variáveis nos experimentos foram controladas para minimizar as ameaças. Para aumentar a confiança os dados foram analisados não apenas por meio de tabelas e gráficos, mas também com auxílio de testes estatísticos para garantir que os resultados obtidos sejam de fato interpretados de maneira correta e para assegurar que as relações apresentadas por meio de tabelas e gráficos sejam realmente significativas e não simplesmente geradas ao acaso.

5.4 Comparando resultados

A terceira questão de pesquisa (QP_3) levanta um questionamento sobre como os resultados obtidos durante a execução do experimento realizado comportam-se em comparação aos trabalhos similares existentes na literatura. Para que seja possível realizar esta comparação, são apresentados de maneira resumida os resultados obtidos por [Mateo e Usaola \(2013\)](#).

Em linhas gerais o trabalho desenvolvido por [Mateo e Usaola \(2013\)](#) traz a implementação dos algoritmos de carga apresentados nesta dissertação, porém, aplicado ao contexto do Teste de Mutação para programas em Java. Devido ao fato dos dados brutos sobre os resultados obtidos não estarem disponíveis, a discussão dos resultados foi feita utilizando como base um breve resumo da análise descritiva apresentada no artigo supracitado. O Quadro 4 mostra os objetos experimentais utilizados durante a avaliação experimental realizada neste trabalho.

O experimento foi conduzido em duas etapas, durante a primeira etapa o objetivo era realizar, de maneira geral, um comparativo entre os cinco algoritmos apresentados. Assim, os algoritmos foram avaliados utilizando três dos quatro programas descritos no Quadro 4. Durante a avaliação foi medida a eficiência da aplicação de cada algoritmo (tempo de resposta) para o respectivo conjunto de casos de testes para cada programa. O resultado da execução para cada

Quadro 4 – Programas utilizados no experimento por [Mateo e Usaola \(2013\)](#)

Programas	Metodos	LC	Mutantes	Casos de Testes
Monopoly	135	654	2040	108
Cinema	106	678	1250	140
Chess	108	2288	5737	137
Analysis	195	2398	8036	152

algoritmo é apresentado de maneira resumida nas Tabelas 22, 23 e 24.

O trabalho também faz uma análise sobre o efeito causado ao utilizar ambientes de execução contendo máquinas com um único sistema operacional (ambiente homogêneo) e ambientes de execução contendo máquinas com dois sistemas operacionais (ambientes heterogêneos). Esta análise é descartada neste, pelo fato desta avaliação não ser levada em consideração durante a execução do experimento executado neste trabalho.

Tabela 22 – Resultados para execução com 8 e 19 máquinas ([Mateo e Usaola \(2013\)](#))

8 máquinas executoras (Heterogêneo)				19 máquinas executoras (Homogêneo)			
Algoritmos	Cine	Monopoly	Chess	Algoritmos	Cine	Monopoly	Chess
PEDRO	0,67s	3,17s	0,74s	PEDRO	28,79s	23,75s	103,06s
GMOD	0,40s	2,85s	1,81s	GMOD	28,07s	25,42s	110,40s
DMBO	132,02s	151,12s	732,92s	DMBO	48,68s	59,39s	125,14s
DTC	96,65s	251,77s	347,30s	DTC	56,51s	48,75s	204,53s
GTCOD	39,64s	10,85s	85,05s	GTCOD	105,07s	69,13s	255,66s

Tabela 23 – Resultados para execução com 40 máquinas ([Mateo e Usaola \(2013\)](#))

40 máquinas executoras (Homogêneo)				40 máquinas executoras (Heterogêneo)			
Algoritmos	Cine	Monopoly	Chess	Algoritmos	Cine	Monopoly	Chess
PEDRO	22,69s	16,52s	74,44s	PEDRO	21,28s	16,23s	64,12s
GMOD	21,94s	16,24s	75,33s	GMOD	21,18s	15,84s	64,63s
DMBO	27,18s	22,65s	82,05s	DMBO	26,41s	21,99s	81,36s
DTC	54,25s	34,87s	129,77s	DTC	54,18s	39,15s	118,34s
GTCOD	66,90s	30,32s	187,20s	GTCOD	56,76s	28,80s	128,16s

Tabela 24 – Resultados para execução com 80 máquinas ([Mateo e Usaola \(2013\)](#))

80 máquinas executoras (Homogêneo)				80 máquinas executoras (Heterogêneo)			
Algoritmos	Cine	Monopoly	Chess	Algoritmos	Cine	Monopoly	Chess
PEDRO	14,40s	11,48s	36,32s	PEDRO	13,82s	12,06s	35,64s
GMOD	13,70s	11,66s	34,97s	GMOD	14,78s	12,27s	36,23s
DMBO	17,62s	18,10s	42,98s	DMBO	17,03s	21,00s	43,08s
DTC	278,07s	30,86s	150,31s	DTC	28,54s	34,91s	117,59s
GTCOD	67,44s	26,78s	176,87s	GTCOD	52,20s	30,09s	236,65s

O tempo de execução total mostrado nas tabelas 22, 23, 24 mostra os algoritmos PEDRO e GMOD como os que obtiveram os melhores resultados. Seus tempos de execução em geral

são bem próximos. O terceiro melhor algoritmo nesta linha foi o algoritmo DMBO que teve resultados um pouco piores que os algoritmos PEDRO e GMOD, um fato curioso é que para este algoritmo, conforme o número de máquinas é aumentado, sua eficiência melhora em comparação aos algoritmos PEDRO e GMOD. Os algoritmos DTC e GTCOD são os algoritmos que apresentaram os piores resultados variando um pouco seu desempenho de acordo com o programa analisado.

Devido à grande semelhança dos resultados obtidos na análise dos algoritmos PEDRO e GMOD, foi conduzida uma segunda etapa na execução do experimento utilizando o programa *Analysis* descrito no Quadro 4 para atestar de fato qual o algoritmo com melhor resultado. Optou-se pela utilização de um programa de maior complexidade pelo fato dos programas previamente analisados serem exemplos relativamente simples e, por conta disso, possuírem pouca representatividade.

Tabela 25 – Segunda etapa - Avaliação dos algoritmos PEDRO e GMOD (*Analysis*)

Configuração	PEDRO	GMOD
80 Heterogêneo	71,38s	73,01s
80 Homogêneo	71,60s	75,04s
40 Heterogêneo	119,07s	121,49s
40 Homogêneo	120,79s	120,89s
19 Homogêneo	171,56s	175,40s
8 Heterogêneo	825,42s	875,96s

A Tabela 25 mostra o tempo total obtido para cada uma das configurações analisadas para o objeto de estudo *Analysis* utilizando os algoritmos PEDRO e GMOD. É possível observar que o algoritmo PEDRO obteve melhores resultados que o algoritmo GMOD para todas as configurações, contudo pode-se observar que a diferença entre os algoritmos é reduzida conforme o número de máquinas utilizadas para processamento é aumentado. Estes resultados confirmaram o algoritmo PEDRO como a melhor abordagem como algoritmo de balanceamento de carga a ser utilizada.

Ao comparar os resultados apresentados por Mateo e Usaola (2013) com os resultados obtidos durante a execução dos experimentos desta dissertação, é claramente possível notar uma semelhança entre os resultados. Assim como os experimentos realizados no artigo analisado, os experimentos realizados para este trabalho também evidenciaram o algoritmo PEDRO como sendo o algoritmo com os melhores resultados, seguido do algoritmo GMOD e do algoritmo DMBO. Contudo, ao analisar os programas utilizados neste experimento de maneira individual (Quadro 3), foi possível observar que houve uma variação maior entre os algoritmos que apresentaram melhor resultado individualmente para cada programa avaliado do experimento.

Apesar do algoritmo PEDRO ter mostrando-se como melhor alternativa para a maioria dos programas, além do algoritmo GMOD, os algoritmos DTC e DMBO também demonstraram ser a melhor abordagem a ser utilizada para alguns dos programas utilizados na execução do

experimento. Este comportamento pode ser possivelmente justificado por conta da sessão de testes utilizada durante a execução do experimento, conforme mencionado anteriormente. Outro ponto que também pode ser questionado é o baixo número de programas utilizados durante a execução do experimento de Mateo e Usaola. Conforme os mesmos comentam nas ameaças à validade do experimento, a natureza e o tamanho dos programas utilizados para realização do experimento limitam a generalização dos resultados obtidos, sendo necessário a avaliação de mais tipos de programas para que os resultados possam de fato ser generalizados e conclusivos.

5.5 Considerações Finais

Conforme discutido no Capítulo 2, performance ainda é um grande desafio para a aplicação do Teste de Mutação, o que torna a exploração de estudo nesse sentido sempre temas de pesquisa interessantes. A arquitetura avaliada neste capítulo utiliza como premissa a possibilidade de utilização de estruturas paralelas para melhorar a eficiência na aplicação do Teste de Mutação.

De acordo com os resultados obtidos pela avaliação experimental conduzida, pode-se concluir que a aplicação do Teste de Mutação em paralelo pode de fato beneficiar a aplicabilidade da técnica. Os estudos comprovaram que a utilização de um ambiente em paralelo com um pequeno número de máquinas para melhorar bastante o processamento e possibilitar um grande *speedup*.

Os resultados dos experimentos comprovaram que em uma simples configuração, utilizando quatro máquinas para processamento de mutantes, pode acarretar em uma melhora de até 78,22% em comparação com a aplicação sequencial do Teste de Mutação. Porém, é necessário justificar que a aplicabilidade da técnica tem como objetivo superar o gargalo de processamento quando a técnica é aplicada a um grande conjunto de mutantes, não justificando a sua aplicabilidade em programas relativamente pequenos.

Procurou-se realizar testes estatísticos com objetivo de dar um maior grau de confiança sobre os resultados obtidos a fim de tornar possível a generalização dos resultados. Por fim, foi realizada uma análise comparativa com os resultados de um trabalho similar na literatura que motivou o desenvolvimento deste trabalho. Como resultado desta comparação, pode-se observar que apesar de serem aplicados a contextos diferentes, ambos os trabalhos apresentaram resultados bem semelhantes.

Como trabalhos futuros, pretende-se aperfeiçoar a camada de comunicação entre as máquinas envolvidas no processo. Esta etapa tem como objetivo proporcionar maior liberdade entre as máquinas envolvidas no processo, possibilitando a execução assíncrona e solucionar problemas relacionados à tolerância a falhas durante a execução por parte das máquinas executoras. Além disso, pretende-se possibilitar a incorporação de técnicas de geração de dados de teste ao processo de execução do Teste de Mutação em paralelo. Esta etapa também serviria como outra justificativa para adoção e aplicação da abordagem. Por fim, submeter a abordagem

à avaliação de um conjunto extra de programas programas com objetivo de avaliar e corroborar com os resultados aqui apresentados.

CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo faz uma revisão sobre as questões de pesquisa que nortearam o desenvolvimento deste trabalho (Seção 6.1). Com base nos resultados obtidos durante o desenvolvimento deste trabalho, na Seção 6.2 são discutidas vantagens e desvantagens sobre a utilização de computação paralela na aplicação do Teste de Mutação. Este capítulo também ressalta as contribuições e limitações (Seções 6.3, 6.4) da abordagem proposta e, por fim, se encerra com uma discussão da relevância desta pesquisa, além de destacar os desafios que foram encontrados durante o desenvolvimento da pesquisa e seus possíveis desdobramentos futuros.

6.1 Problema de Pesquisa e Questões de Pesquisa

Conforme abordado no Capítulo 3, o avanço da capacidade de processamento dos computadores abriu um grande leque para o desenvolvimento de estruturas complexas para dar suporte ao processamento de uma crescente demanda de dados. Uma importante técnica para redução de custos é a computação paralela que já foi amplamente utilizada com sucesso em diversas áreas. Em geral, os estudos voltados ao Teste de Mutação que procuraram aplicar conceitos relacionados a programação paralela datam do fim da década de 1980 e meados da década de 1990, voltando ao interesse da comunidade acadêmica em 2012, por conta dos resultados apresentados em (MATEO; USAOLA, 2013).

O Capítulo 2 procurou resumir o conjunto de trabalhos na literatura que tratam sobre técnicas que tem como objetivo a redução do custo computacional do Teste de Mutação. Dentre as técnicas, é possível destacar que existe uma grande gama de estudos relacionados a abordagens *do fewer* e *do smarter*. Contudo, estudos relacionados a abordagem *do faster* não acompanharam o mesmo avanço. Graças ao desenvolvimento dos computadores durante o período de hiato nesta linha de pesquisa, e ao recente retorno ao interesse de pesquisas nesta área, justifica-se o desenvolvimento desta pesquisa a fim de validar resultados e propor novos desdobramentos ao estado da arte.

Neste contexto, esta dissertação adotou como problema de pesquisa como ambientes de execução distribuídos podem ser utilizados para melhorar a aplicabilidade do Teste de Mutação. E com base neste problema de pesquisa, foram derivadas questões de pesquisa que nortearam o desenvolvimento deste trabalho. Com base nos resultados obtidos, as questões de pesquisa obtiveram as seguintes conclusões:

- **QP₁** : É possível melhorar a eficiência computacional na aplicabilidade do Teste de Mutação utilizando técnicas de computação paralela?
 - Com objetivo de responder à primeira questão de pesquisa, foi desenvolvida uma arquitetura que suportasse a aplicação dos algoritmos de balanceamento de carga existentes na literatura. À medida que os resultados para responder esta questão de pesquisa foram obtidos, foram realizados novos experimentos com objetivo de medir o grau de melhoria na eficiência da aplicação do Teste de Mutação em paralelo, comparado ao cenário convencional de aplicação do Teste de Mutação em ambientes sequenciais. Os resultados dos experimentos foram descritos no Capítulo 5 e comprovaram que a adoção de uma estrutura relativamente simples é capaz de reduzir substancialmente o gargalo de processamento do Teste de Mutação melhorando a eficiência na aplicação da técnica.

Ao observar os resultados obtidos durante a primeira etapa da avaliação experimental, decidiu-se então explorar a diferença entre os algoritmos de balanceamento de carga utilizados durante a avaliação:

- **QP₂** : Existe diferença significativa entre o tempo de resposta dos algoritmos de balanceamento de carga implementados na arquitetura proposta?
 - De posse dos resultados da primeira avaliação, que comprovaram a superioridade da eficiência da aplicação do Teste de Mutação em paralelo comparado a sua aplicação sequencial convencional, decidiu-se, então, verificar qual o relacionamento existente entre os algoritmos de balanceamento de carga abordados. Os resultados obtidos mostraram que os algoritmos apresentam de fato resultados diferentes.
Este resultado levou à uma análise mais extensa dos resultados, assim foi feita uma análise dos grupos para que fosse possível atestar qual o algoritmo de balanceamento de carga mais indicado para utilização. Assim, por meio de uma análise estatística extra, foi possível comprovar que o algoritmo PEDRO apresentou resultados mais eficientes em todas as configurações utilizadas no experimento.

De posse de todos estes resultados obtidos nas questões de pesquisa um e dois, restava, então, responder à questão de pesquisa três que tem como finalidade analisar os resultados obtidos nesta pesquisa e comparar com os resultados do trabalho que deu origem a essa investigação:

- **QP₃** : Como os resultados obtidos se comportam em relação a trabalhos similares existentes na literatura?
 - Foi realizada uma análise descritiva dos resultados de ambos os trabalhos e chegou-se à conclusão que ambos os trabalhos possuem resultados bem semelhantes, resguardando alguns resultados específicos obtidos nesta pesquisa que destoam dos resultados apresentados por [Mateo e Usaola \(2013\)](#).

6.2 Teste de Mutação em Paralelo: Vantagens na Adoção

Com base nos resultados desta pesquisa e nos trabalhos relacionados, é possível concluir que a aplicação do Teste de Mutação pode ser significativamente beneficiada por meio da utilização de técnicas de computação paralela. Durante a execução dos algoritmos de balanceamento e distribuição procurou-se utilizar uma grande variedade de configurações para execução e os resultados evidenciaram que é plenamente possível atingir bons resultados, mesmo sem a utilização da configuração mais robusta explorada.

Conclui-se, portanto, que esta abordagem pode viabilizar a aplicação do Teste de Mutação em sistemas mais complexos, além de possibilitar a verificação intensiva de sistemas possibilitando a adoção de técnicas como geração automática de casos de teste que pode ser plenamente incorporada ao processo. Estudos experimentais sobre o Teste de Mutação, que, em geral, requerem um número muito grande de execuções de mutantes em diversos objetos de estudos (programas) também são beneficiados com a adoção deste tipo de abordagem.

6.3 Contribuições

Foi realizada uma profunda revisão da literatura sobre o tema, durante o desenvolvimento deste trabalho. Grande parte dos trabalhos relacionados à redução de custos do Teste de Mutação tratam sobre como selecionar o melhor conjunto de mutantes a ser executado, ou sobre técnicas de otimização.

Dentre os trabalhos que tratam de arquiteturas especializadas para execução de mutantes, em geral, datam de décadas atrás. [Mateo e Usaola \(2013\)](#) foram pioneiros em voltar a explorar este campo fazendo a análise de algoritmos de balanceamento de carga adaptados ao Teste de Mutação aplicados no contexto de programas escritos na linguagem de programação *Java*. [Durelli, Offutt e Delamaro \(2012\)](#) também trabalharam na linha de computação concorrente, propondo uma abordagem que possibilita a aplicação de *Weak Mutation* integrando o processo do Teste de Mutação à máquina virtual responsável pela execução do código *Java* utilizando de *threads* para possibilitar a paralelização da execução dos mutantes.

Este trabalho teve como objetivo propor e avaliar uma arquitetura para aplicação do Teste de Mutação em paralelo no contexto de programas desenvolvidos para a linguagem de programação C. Julga-se necessária esta avaliação para possibilitar a comparação e questionamento dos resultados obtidos com resultados já existente na literatura afim de gerar novos desdobramentos e possibilitar novas pesquisas na área.

Os resultados mostraram uma grande melhora de performance, levando em consideração o custo computacional. Observou-se que nos piores cenários foi possível melhorar a eficiência da técnica em até 70% em relação à aplicação sequencial. Ao comparar os resultados obtidos com os resultados apresentados por [Mateo e Usaola \(2013\)](#), observou-se uma similaridade nos resultados quanto ao melhor algoritmo de balanceamento de carga a ser utilizado. Este fator contribui com indícios para uma possível generalização dos resultados evidenciando que os resultados alcançados pelos algoritmos indeferem quanto à plataforma utilizada.

6.4 Limitações

A prova de conceito descrita neste trabalho reforça a hipótese que a utilização de sistemas de processamento paralelo pode beneficiar a aplicação do Teste de Mutação, contudo, existem algumas limitações:

- *Trade-of* da infraestrutura:
 - À medida em que a utilização de uma infraestrutura de processamento mais robusta pode beneficiar a aplicação do Teste de Mutação, há de se alertar que existe um contraponto nesta abordagem. Ao se utilizar de uma infraestrutura complexa, ganha-se em termos de tempo de resposta, contudo, há o aumento relacionado ao maior número de máquinas a ser gerenciadas e todo o gasto existente na operacionalização de cada uma delas.
- Limitações inerentes à ferramenta *Proteum*:
 - Como a arquitetura apresentada utiliza das funcionalidades propostas por [Delamaro e Maldonado \(1996b\)](#), todas as limitações existentes na ferramenta são automaticamente propagadas para a utilização desta abordagem. Além disso, como a arquitetura proposta foi concebida para ser utilizada em conjunto com a ferramenta *Proteum*, a extensão desta abordagem para aplicação do Teste de Mutação com alguma ferramenta diferente inclui a criação de uma interface entre a ferramenta e a abordagem de forma a respeitar as regras impostas durante o seu desenvolvimento.

6.5 Trabalhos Futuros

Como trabalhos futuros, pretende-se aperfeiçoar a camada de comunicação existente entre as máquinas envolvidas na arquitetura, esta etapa tem como objetivo proporcionar maior liberdade entre as máquinas envolvidas no processo, possibilitando a execução assíncrona e solucionar problemas relacionados à tolerância a falhas durante a execução por parte das máquinas executoras. Além disso, pretende-se possibilitar a incorporação de técnicas de geração de dados de teste ao processo de execução do Teste de Mutação em paralelo. Esta etapa também serviria como outra justificativa para adoção e aplicação da abordagem.

Pretende-se estender os experimentos realizados para um maior número de máquinas executoras a fim de identificar um número ideal de máquinas que garanta o melhor desempenho possível na utilização da arquitetura, visto que a capacidade de processamento de estruturas paralelas não aumenta linearmente e fatalmente a partir de um determinado instante aumentar o número de máquinas na infraestrutura contribuirá não mais para aumento de eficiência da abordagem, mas para a sua redução. Não foi possível identificar este comportamento com o número de máquinas utilizadas no experimento realizado nesta pesquisa, então, espera-se que estender o experimento com número maior de máquinas possibilite a obtenção desta informação. Por fim, pretende-se submeter a abordagem à avaliação de um conjunto novo de programas com objetivo de avaliar e corroborar para a generalização dos resultados aqui apresentados.

6.6 Conclusões

Esta pesquisa contribuiu para levantar indícios sobre a aplicabilidade do Teste de Mutação utilizando estruturas complexas para processamento de mutantes. Para atingir esse objetivo, foi desenvolvida uma arquitetura que possibilita a aplicação do Teste de Mutação utilizando técnicas de programação paralela.

Foi analisado um conjunto de algoritmos de balanceamento de carga que possuem como finalidade dividir e coordenar a distribuição dos dados durante a execução do Teste de Mutação. Os experimentos conduzidos (Capítulo 5) serviram para responder as questões de pesquisa que nortearam o desenvolvimento deste trabalho, bem como para refutar as hipóteses analisadas ao longo do trabalho. Estes resultados foram de extrema importância para validar a abordagem desenvolvida, comprovando a eficiência da utilização de computação paralela com suporte à execução do Teste de Mutação, assim como por possibilitar a comparação com outras abordagens existentes na literatura.

Conforme foi especificado durante esta dissertação, o Teste de Mutação possui uma grande eficácia em revelar a presença de defeitos em um produto de software. A principal contribuição desta pesquisa concentra-se então em reforçar os indícios de que a utilização de técnicas de programação paralela pode ajudar na aplicação do Teste de Mutação superando

gargalos referentes à questão de processamento. Tal fato possibilita a aplicação do Teste de Mutação em ambientes reais, possibilita a utilização de técnicas de geração de dados de teste em conjunto à aplicação do Teste de Mutação, além de facilitar a execução de experimentos, que, normalmente requerem um número muito grande de execuções, com um número grande de programas. Todos esses fatores contribuem diretamente para desdobramentos de novas pesquisas na área.

REFERÊNCIAS

ACREE, A. T. **On Mutation**. Tese (phdthesis) — Georgia Institute of Technology, Atlanta, Georgia, 1980. Citado na página 41.

AGRAWAL, H.; DEMILLO, R. A.; HATHAWAY, B.; HSU, W.; HSU, W.; KRAUSER, E. W.; MARTIN, R. J.; MATHUR, A. P.; SPAFFORD, E. **Design of Mutant Operators for the C Programming Language**. West Lafayette, Indiana, 1989. Citado 3 vezes nas páginas 37, 38 e 79.

AICHERNIG, B. K.; JIFENG, H. Mutation testing in utp. **Formal Aspects of Computing**, v. 21, n. 1-2, p. 33–64, February, 2009 2009. Citado na página 27.

AMMANN, P.; DELAMARO, M. E.; OFFUTT, J. Establishing theoretical minimal sets of mutants. In: **Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA**. [s.n.], 2014. p. 21–30. Disponível em: <<http://dx.doi.org/10.1109/ICST.2014.13>>. Citado na página 79.

ANDREWS, G. **Multithreaded, Parallel, and Distributed Programming**. [S.l.]: Addison-Wesley, 2000. 1 – 34 p. ISBN 9780201357523. Citado na página 52.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? In: **Proceedings of the 27th International Conference on Software Engineering (ICSE'05)**. St Louis, Missouri: [s.n.], 2005. p. 402 – 411. Citado na página 27.

BARBOSA, E. F.; CHAIM, M. L.; VICENZI, A. M. R.; DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. Teste Estrutural. In: DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. (Ed.). **Introdução ao Teste de Software**. [S.l.]: Elsevier, 2007. p. 47 – 76. Citado na página 35.

BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R. Toward the determination of sufficient mutant operators for c. **Software Testing, Verification and Reliability**, v. 11, n. 2, p. 113–136, May 2001. Citado na página 42.

BASIL, V.; WEISS, D. A methodology for collecting valid software engineering data. **Software Engineering, IEEE Transactions on**, SE-10, n. 6, p. 728–738, Nov 1984. ISSN 0098-5589. Citado na página 76.

BOOTH, W.; COLOMB, G.; WILLIAMS, J. **The Craft of Research, Third Edition**. [S.l.]: University of Chicago Press, 2009. ISBN 9780226062648. Citado 2 vezes nas páginas 27 e 28.

BRADBURY, J. S.; CORDY, J. R.; DINGEL, J. Exman: A generic and customizable framework for experimental mutation analysis. In: **Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)**. Raleigh, North Carolina: [s.n.], 2006. p. 57–62. Citado na página 46.

BUDD, T. A. **Mutation Analysis of Program Test Data**. Tese (phdthesis) — Yale University, New Haven, Connecticut, 1980. Citado na página 41.

- BUDD, T. A.; DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. The design of a prototype mutation system for program testing. In: **Proceedings of the AFIPS National Computer Conference**. Anaheim, New Jersey: [s.n.], 1978. p. 623–627. Citado na página 46.
- CHAIM, M. L. **PokeTool – Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados**. Dissertação (Mestrado) — DCA/FEEC/UNICAMP, Campinas, SP, April 1991. Citado na página 48.
- CHEVALLEY, P.; THÉVENOD-FOSSE, P. A mutation analysis tool for java programs. **International Journal on Software Tools for Technology Transfer**, v. 5, n. 1, p. 90–103, November 2002. Citado na página 46.
- CHOI, B.; MATHUR, A. P. High-performance mutation testing. **Journal of Systems and Software**, v. 20, n. 2, p. 135–152, February 1993. Citado 2 vezes nas páginas 45 e 64.
- COLES, H. **Pitest - Real world mutation testing**. [S.l.], 2015. <<http://pitest.org/>>. Disponível em: <<http://pitest.org/>>. Citado na página 46.
- DELAMARE, R.; BAUDRY, B.; TRAON, Y. L. Ajmutator: A tool for the mutation analysis of aspectj pointcut descriptors. In: **Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09)**. Denver, Colorado: [s.n.], 2009. p. 200–204. Citado na página 46.
- DELAMARO, M.; DENG, L.; DURELLI, V. S.; LI, N.; OFFUTT, J. Experimental evaluation of sdl and one-op mutation for c. In: **Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on**. [S.l.: s.n.], 2014. p. 203–212. Citado na página 79.
- DELAMARO, M. E. **Proteum - A Mutation Analysis Based Testing Environmen**. Tese (phdthesis) — University of São Paulo, Sao Paulo, Brazil, 1993. Citado na página 43.
- DELAMARO, M. E.; BARBOSA, E. F.; VICENZI, A. M. R.; MALDONADO, J. C. Teste de Mutação. In: DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. (Ed.). **Introdução ao Teste de Software**. [S.l.]: Elsevier, 2007. p. 77 – 117. Citado 2 vezes nas páginas 36 e 46.
- DELAMARO, M. E.; MALDONADO, J. C. Proteum-a tool for the assessment of test adequacy for c programs. In: **Proceedings of the Conference on Performability in Computing Systems (PCS'96)**. New Brunswick, New Jersey: [s.n.], 1996. p. 79–95. Citado na página 43.
- _____. Proteum-A tool for the assessment of test adequacy for C programs. In: **Proceedings of the Conference on Performability in Computing Systems (PCS 96)**. New Brunswick, NJ: [s.n.], 1996. p. 79–95. Citado 4 vezes nas páginas 46, 48, 76 e 112.
- _____. Interface mutation: Assessing testing quality at interprocedural level. In: **Proceedings of the 19th International Conference of the Chilean Computer Science Society (SCCC'99)**. Talca, Chile: [s.n.], 1999. p. 78–86. Citado na página 27.
- DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. Introdução ao teste de software. In: DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. (Ed.). **Introdução ao Teste de Software**. [S.l.]: Elsevier, 2007. p. 1 – 7. Citado 4 vezes nas páginas 25, 26, 32 e 33.
- DELAMARO, M. E.; MALDONADO, J. C.; MATHUR, A. P. Interface mutation: An approach for integration testing. **IEEE Transactions on Software Engineering**, v. 27, n. 3, p. 228–247, May 2001. Citado 2 vezes nas páginas 37 e 38.

DELAMARO, M. E.; MALDONADO, J. C.; VINCENZI, A. Proteum/im 2.0: An integrated mutation testing environment. In: **Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)**. San Jose, California: [s.n.], 2001. p. 91–101. Citado na página 47.

DELAMARO, M. E.; OFFUTT, J.; AMMANN, P. Designing deletion mutation operators. In: **Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation**. Washington, DC, USA: IEEE Computer Society, 2014. (ICST '14), p. 11–20. ISBN 978-1-4799-2255-0. Disponível em: <http://dx.doi.org/10.1109/ICST.2014.12>. Citado 2 vezes nas páginas 79 e 80.

DEMILLO, R. **Mutation Analysis as a Tool for Software Quality Assurance**. [S.l.]: School of Information and Computer Science, Georgia Institute of Technology, 1980. Citado 2 vezes nas páginas 27 e 36.

DEMILLO, R. A.; GUINDI, D. S.; KING, K. N.; MCCRACKEN, W. M.; OFFUTT, A. J. An extended overview of the mothra software testing environment. In: **Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88)**. Banff Alberta, Canada: [s.n.], 1988. p. 142–151. Citado na página 46.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. **Computer**, v. 11, n. 4, p. 34–41, April 1978. Citado 2 vezes nas páginas 36 e 42.

DEMILLO, R. A.; OFFUTT, A. J. Constraint-based automatic test data generation. **IEEE Transactions on Software Engineering**, v. 17, n. 9, p. 900–910, September 1991. Citado na página 37.

DEREZIŃSKA, A.; SZUSTEK, A. Tool-supported advanced mutation approach for verification of c# programs. In: **Proceedings of the 3th International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX'08)**. Szklarska Poręba, Poland: [s.n.], 2008. p. 261–268. Citado na página 46.

DODGE, Y. **The Concise Encyclopedia of Statistics**. [S.l.]: Springer, 2008. ISBN 9780387317427. Citado 2 vezes nas páginas 94 e 99.

DURELLI, V.; OFFUTT, J.; DELAMARO, M. Toward harnessing high-level language virtual machines for further speeding up weak mutation testing. In: **Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on**. [S.l.: s.n.], 2012. p. 681–690. Citado 2 vezes nas páginas 43 e 111.

FABBRI, S. C. P. F.; MALDONADO, J. C.; MASIERO, P. C.; DELAMARO, M. E.; WONG, W. E. Mutation testing applied to validate specifications based on petri nets. In: **Proceedings of the IFIP TC6 8th International Conference on Formal Description Techniques VIII**. [S.l.: s.n.], 1995. p. 329–337. Citado na página 27.

FABBRI, S. C. P. F.; VINCENZI, A. M. R.; MALDONADO, J. C. Teste funcional. In: DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. (Ed.). **Introdução ao Teste de Software**. [S.l.]: Elsevier, 2007. p. 9 – 24. Citado na página 34.

FERRARI, F. C.; NAKAGAWA, E. Y.; MALDONADO, J. C.; RASHID, A. Proteum/aj: a mutation system for aspectj programs. In: BORBA, P.; CHIBA, S. (Ed.). **AOSD (Companion)**. [S.l.]: ACM, 2011. p. 73–74. ISBN 978-1-4503-0606-5. Citado na página 46.

FICHTER, S. **Parallelizing Mutation on a Hypercube**. Dissertação (mastersthesis) — Clemson University, Clemson, SC, 1991. Citado na página 61.

FLYNN, M. Very high-speed computing systems. **Proceedings of the IEEE**, v. 54, n. 12, p. 1901–1909, 1966. ISSN 0018-9219. Citado 4 vezes nas páginas 51, 52, 54 e 58.

GLIGORIC, M.; BADAME, S.; JOHNSON, R. SMutant: a tool for type-sensitive mutation testing in a dynamic language. In: GYIMÓTHY, T.; ZELLER, A. (Ed.). **Proceedings of the 19th Symposium on the Foundations of Software Engineering and the 13rd European Software Engineering Conference**. [S.l.]: ACM, 2011. p. 424–427. ISBN 978-1-4503-0443-6. Citado na página 46.

GLIGORIC, M.; JAGANNATH, V.; LUO, Q.; MARINOV, D. Efficient mutation testing of multithreaded code. **Software Testing, Verification and Reliability**, v. 23, n. 5, p. 375–403, 2013. ISSN 1099-1689. Citado na página 46.

GRAMA, A. **Introduction to Parallel Computing**. [S.l.]: Addison-Wesley, 2003. ISBN 9780201648652. Citado 4 vezes nas páginas 52, 55, 57 e 62.

HARROLD, M. J. Testing: A roadmap. In: **Proceedings of the Conference on The Future of Software Engineering**. New York, NY, USA: ACM, 2000. (ICSE '00), p. 61–72. ISBN 1-58113-253-0. Citado na página 40.

HAŁAS, K. **MutPy - A Mutation Testing Tool for Python 3.x**. [S.l.], 2014. <<https://bitbucket.org/khalas/mutpy>>. Disponível em: <<https://bitbucket.org/khalas/mutpy>>. Citado na página 46.

HOWDEN, W. E. Weak mutation testing and completeness of test sets. **IEEE Transactions on Software Engineering**, v. 8, n. 4, p. 371–379, July 1982. Citado na página 43.

HUMMEL, S. F.; SCHONBERG, E.; FLYNN, L. E. Factoring: A method for scheduling parallel loops. **Commun. ACM**, ACM, New York, NY, USA, v. 35, n. 8, p. 90–101, ago. 1992. ISSN 0001-0782. Citado na página 66.

HUSSAIN, S. **Mutation Clustering**. Tese (phdthesis) — King's College London, UK, 2008. Citado na página 41.

IRVINE, S. A.; PAVLINIC, T.; TRIGG, L.; CLEARY, J. G.; INGLIS, S. J.; UTTING, M. Jumble java byte code to measure the effectiveness of unit tests. In: **Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)**. Windsor, UK: [s.n.], 2007. p. 169–175. Citado na página 46.

JACKSON, D.; WOODWARD, M. R. Parallel firm mutation of java programs. In: **Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)**. San Jose, California: [s.n.], 2001. p. 55–61. Citado na página 43.

JAIN, R. **The Art Of Computer Systems Performance Analysis**. [S.l.]: Wiley India Pvt. Limited, 1996. ISBN 9788126519057. Citado 2 vezes nas páginas 57 e 58.

JI, C.; CHEN, Z.; XU, B.; ZHAO, Z. A novel method of mutation clustering based on domain analysis. In: **Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE'09)**. Boston, Massachusetts: [s.n.], 2009. Citado na página 41.

JIA, Y.; HARMAN, M. Constructing subtle faults using higher order mutation testing. In: **Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)**. Beijing, China: [s.n.], 2008. p. 249–258. Citado na página 42.

_____. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In: **Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08)**. Windsor, UK: [s.n.], 2008. p. 94–98. Citado na página 46.

_____. An analysis and survey of the development of mutation testing. **IEEE Transactions of Software Engineering**, v. 37, n. 5, p. 649–678, 2011. Citado 5 vezes nas páginas 15, 28, 40, 41 e 46.

JOHNSON, R. **Statistics: Principles and Methods, 6th Edition**. [S.l.]: Wiley Global Education, 2009. ISBN 9781118049426. Citado na página 95.

JURISTO, N.; GÓMEZ, O. S. Empirical software engineering and verification. In: MEYER, B.; NORDIO, M. (Ed.). **Empirical Software Engineering and Verification**. Berlin, Heidelberg: Springer-Verlag, 2012. cap. Replication of Software Engineering Experiments, p. 60–88. ISBN 978-3-642-25230-3. Disponível em: <<http://dl.acm.org/citation.cfm?id=2184075.2184077>>. Citado na página 29.

KAPFHAMMER, G. M. Automatically and transparently distributing the execution of regression test suites. In: **Proceedings of the 18th International Conference on Testing Computer Software**. [S.l.: s.n.], 2001. Citado 2 vezes nas páginas 63 e 65.

KIM, S.; CLARK, J. A.; MCDERMID, J. A. Assessing test set adequacy for object oriented programs using class mutation. In: **Proceedings of the 3rd Symposium on Software Technology (SoST'99)**. Buenos Aires, Argentina: [s.n.], 1999. Citado na página 38.

KING, K. N.; OFFUTT, A. J. A fortran language system for mutation-based software testing. **Software: Practice and Experience**, v. 21, n. 7, p. 685–718, October 1991. Citado na página 43.

KOSCIANSKI, A.; SOARES, M. dos S. **Qualidade de software: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software**. [S.l.]: Novatec Editora, 2007. ISBN 9788575221129. Citado na página 35.

KRAUSER, E. W.; MATHUR, A. P.; REGO, V. J. High performance software testing on simd machines. **IEEE Transactions on Software Engineering**, v. 17, n. 5, p. 403–423, May 1991. Citado na página 44.

MA, Y.-S.; OFFUTT, A. J.; KWON, Y.-R. Mujava: An automated class mutation system. **Software Testing, Verification & Reliability**, v. 15, n. 2, p. 97–133, June 2005. Citado 3 vezes nas páginas 38, 44 e 46.

MACHADO, F.; MAIA, L. **Arquitetura de sistemas operacionais**. [S.l.]: Livros Tecnicos e Cientificos, 1996. ISBN 9788521610977. Citado 2 vezes nas páginas 53 e 54.

MADEYSKI, L.; RADYK, N. Judy – a mutation testing tool for java. **IET Software**, v. 4, n. 1, p. 32–42, February 2010. Citado na página 46.

MADIRAJU, P.; NAMIN, A. Para μ – a partial and higher-order mutation tool with concurrency operators. In: **Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on**. [S.l.: s.n.], 2011. p. 351–356. Citado na página 46.

MALDONADO, J. C. **Crterios Potenciais Usos: Uma Contribuio ao Teste Estrutural de Software**. Tese (Doutorado) — DCA/FEEC/UNICAMP, Campinas, SP, jul. 1991. Citado na pgina 35.

MATEO, P. R.; USAOLA, M. P. Parallel mutation testing. **Software Testing, Verification and Reliability**, v. 23, n. 4, p. 315–350, 2013. ISSN 1099-1689. Citado 16 vezes nas pginas 17, 21, 28, 29, 45, 62, 66, 75, 76, 79, 103, 104, 105, 109, 111 e 112.

MATEO, P. R.; USAOLA, M. P.; OFFUTT, J. Mutation at the multi-class and system levels. **Sci. Comput. Program.**, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 78, n. 4, p. 364–387, abr. 2013. ISSN 0167-6423. Citado na pgina 46.

MATHUR, A. P. Performance, effectiveness, and reliability issues in software testing. In: **Proceedings of the 5th International Computer Software and Applications Conference (COMP-SAC'79)**. Tokyo, Japan: [s.n.], 1991. p. 604–605. Citado na pgina 42.

MATHUR, A. P.; KRAUSER, E. W. Modeling mutation on a vector processor. In: **Proceedings of the 10th International Conference on Software Engineering**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988. (ICSE '88), p. 154–161. ISBN 0-89791-258-6. Disponvel em: <<http://dl.acm.org/citation.cfm?id=55823.55840>>. Citado na pgina 44.

_____. **Mutant Unification for Improved Vectorization**. West Lafayette, Indiana, 1988. Citado na pgina 61.

MCCABE, T. J. A complexity measure. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 2, n. 4, p. 308–320, jul. 1976. ISSN 0098-5589. Citado na pgina 35.

MCGREGOR, J.; SYKES, D. **A Practical Guide to Testing Object-oriented Software**. [S.l.]: Addison-Wesley, 2001. ISBN 9780201325645. Citado na pgina 32.

MOORE, I. **Jester - The Junit Test Tester**. [S.l.], 2005. <<http://jester.sourceforge.net>>. Disponvel em: <<http://jester.sourceforge.net>>. Citado na pgina 46.

MYERS, G.; SANDLER, C.; BADGETT, T.; THOMAS, T. **The Art of Software Testing**. [S.l.]: Wiley, 2004. ISBN 9780471678359. Citado 4 vezes nas pginas 25, 26, 32 e 34.

NAMIN, A. S.; ANDREWS, J. H. Finding sufficient mutation operators via variable reduction. In: **Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)**. Raleigh, North Carolina: [s.n.], 2006. p. 5. Citado na pgina 42.

OFFUTT, A. J.; PARGAS, R. P.; FICHTER, S. V.; KHAMBEKAR, P. K. Mutation testing of software using a mind computer. In: **Proceedings of the International Conference on Parallel Processing**. Chicago, Illinois: [s.n.], 1992. p. 255–266. Citado 2 vezes nas pginas 45 e 63.

OFFUTT, A. J.; ROTHERMEL, G.; ZAPF, C. An experimental evaluation of selective mutation. In: **Proceedings of the 15th International Conference on Software Engineering (ICSE'93)**. Baltimore, Maryland: [s.n.], 1993. p. 100–107. Citado na pgina 42.

OFFUTT, A. J.; UNTCH, R. H. Mutation 2000: Uniting the orthogonal. In: **Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)**. San Jose, California: [s.n.], 2001. p. 34–44. Citado 3 vezes nas páginas 15, 39 e 40.

OFFUTT, A. J.; VOAS, J.; PAYN, J. **Mutation Operators for Ada**. Fairfax, Virginia, 1996. Citado na página 27.

ORZESZYNA, W. **Solutions to the equivalent mutants problem: A systematic review and comparative experiment**. Dissertação (Mestrado) — School of Computing Blekinge Institute of Technology, Estocolmo, Sweden, 2011. Citado na página 39.

PACHECO, P. **An Introduction to Parallel Programming**. [S.l.]: Elsevier Science, 2011. ISBN 9780080921440. Citado 3 vezes nas páginas 52, 53 e 55.

POLO, M.; PIATTINI, M.; GARCIA-RODRIGUEZ, I. Decreasing the cost of mutation testing with second-order mutants. **Software Testing, Verification and Reliability**, v. 19, n. 2, p. 111 – 131, June 2008. Citado na página 42.

POLO, M.; TENDERO, S.; PIATTINI, M. Integrating techniques and tools for testing automation: Research articles. **Software Testing, Verification and Reliability**, v. 17, n. 1, p. 3–39, March 2007. Citado na página 46.

PRESSMAN, R. **Software engineering: a practitioner's approach**. [S.l.]: McGraw-Hill Higher Education, 2010. ISBN 9780073375977. Citado 3 vezes nas páginas 25, 33 e 35.

QUINN, M. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill Higher Education, 2004. 27 – 60 p. ISBN 9780072822564. Citado 2 vezes nas páginas 28 e 55.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 11, n. 4, p. 367–375, abr. 1985. ISSN 0098-5589. Citado na página 35.

RAUBER, T.; RÜNGER, G. **Parallel Programming: For Multicore and Cluster Systems**. [S.l.]: Springer, 2010. ISBN 9783642048173. Citado na página 56.

SANCHES, R. **Qualidade de software: teoria e prática**. [S.l.]: Prentice Hall, 2001. 54 – 94 p. ISBN 9788587918543. Citado na página 32.

SCHULER, D.; ZELLER, A. Javalanche: Efficient mutation testing for java. In: **Proceedings of the 7th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering**. Amsterdam, Netherlands: [s.n.], 2009. p. 297–298. Citado na página 46.

SHULL, F. J.; CARVER, J. C.; VEGAS, S.; JURISTO, N. The role of replications in empirical software engineering. **Empirical Softw. Engg.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 13, n. 2, p. 211–218, abr. 2008. ISSN 1382-3256. Disponível em: <<http://dx.doi.org/10.1007/s10664-008-9060-1>>. Citado na página 29.

STALLINGS, W. **Arquitetura e organização de computadores**. [S.l.]: Pearson, 2010. 647–694 p. ISBN 9788576055648. Citado 4 vezes nas páginas 15, 51, 53 e 54.

The Institute of Electrical and Eletronics Engineers. **IEEE Standard Glossary of Software Engineering Terminology**. 1990. IEEE Standard. Citado na página 32.

UNTCH, R. H. Mutation-based software testing using program schemata. In: **Proceedings of the 30th Annual Southeast Regional Conference (ACM-SE'92)**. Raleigh, North Carolina: [s.n.], 1992. p. 285–291. Citado na página [44](#).

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in Software Engineering: An Introduction**. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0-7923-8682-5. Citado 2 vezes nas páginas [76](#) e [77](#).

WOODWARD, M. R.; HALEWOOD, K. From weak to strong, dead or alive? an analysis of some mutationtesting issues. In: **Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88)**. Banff Albert, Canada: [s.n.], 1988. p. 152–158. Citado na página [43](#).

WU, X. **Performance Evaluation, Prediction and Visualization of Parallel Systems**. [S.l.]: Springer US, 1999. ISBN 9780792384625. Citado na página [58](#).

YOUNG, M. **Software Testing and Analysis: Process, Principles, and Techniques**. [S.l.]: Wiley India Pvt. Limited, 2008. ISBN 9788126517732. Citado na página [36](#).