PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Programa de Pós-Graduação em Informática

Ana Carolina Medeiros Gonçalves

Estratégias de Paralelização para o Algoritmo de Seleção de Instâncias Baseado em Colônia de Formigas

Belo Horizonte, 9 de Abril de 2025.

Ana Carolina Medeiros Gonçalves

Estratégias de Paralelização para o Algoritmo de Seleção de Instâncias Baseado em Colônia de Formigas

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Mestre em Informática.

Orientadora: Dra. Cristiane Neri Nobre

Coorientador: Dr. Henrique Cota de Freitas

Belo Horizonte, 9 de Abril de 2025.

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Gonçalves, Ana Carolina Medeiros
G635e Estratégias de paralelização para o algoritmo de seleção de instâncias baseado em colônia de formigas / Ana Carolina Medeiros Gonçalves. Belo Horizonte, 2025.
108 f. : il.

Orientadora: Cristiane Neri Nobre
Coorientador: Henrique Cota de Freitas
Dissertação (Mestrado) - Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática

1. Aprendizado do computador. 2. Algoritmos. 3. Otimização combinatória. 4. Processamento paralelo (Computadores). 5. Programação paralela (Computação). 6. Programação heurística. I. Nobre, Cristiane Neri. II. Freitas, Henrique Cota de. III. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática. IV. Título.

CDU: 681.3.03
Ficha catalográfica elaborada por Fabiana Marques de Souza e Silva - CRB 6/2086

Ana Carolina Medeiros Gonçalves

Estratégias de Paralelização para o Algoritmo de Seleção de Instâncias Baseado em Colônia de Formigas

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Mestre em Informática.

Profa. Dra. Cristiane Neri Nobre – PUC Minas (Orientadora)

Prof. Dr. Henrique Cota de Freitas – PUC Minas (Coorientador)

Prof. Dr. Mark Alan Junho Song – PUC Minas (Banca Examinadora)

Prof. Dr. Wellington Santos Martins – Universidade Federal de Goiás (UFG) (Banca Examinadora)

Belo Horizonte, Nove de abril de dois mil e vinte e cinco.

Dedico este trabalho aos meus amigos e à minha família, com especial carinho aos meus pais e aos meus queridos animais de estimação.

AGRADECIMENTOS

Agradecimento especial aos meus Orientadores, Profa. Cristiane Neri e Prof. Henrique Cota de Freitas, por toda a dedicação, confiança e pelos valiosos ensinamentos compartilhados ao longo desse percurso. Sou profundamente grata pelo tempo que dispensaram e pelo constante apoio. Foi e tem sido um privilégio e uma honra trabalhar sob suas orientações. Em particular, expresso minha gratidão à Profa. Cristiane Neri por ter me incentivado e encorajado a embarcar nessa jornada do mestrado, um gesto que marcou o início desta importante etapa da minha vida.

Agradeço a todos os colegas que, ao longo dessa trajetória, se tornaram amigos, compartilhando conhecimentos, experiências e reflexões que enriqueceram minha jornada. Em especial, manifesto minha gratidão aos meus amigos Ariane, Ludmila, Patrick e Pedro pelo apoio inestimável e à minha gata Nininha, que esteve ao meu lado durante muitas noites de pesquisa, oferecendo companhia e tranquilidade.

Expresso todo o meu agradecimento à minha família, em especial aos meus pais, Deolinda e Rogério e aos meus tios, em especial Zínia, Paulo e Tarquínio, que desde o início me incentivaram a trilhar o caminho dos estudos e do mestrado. Sem o apoio incondicional de vocês, nada disso seria possível. Gostaria, também, de estender minha gratidão aos meus avós, cuja sabedoria, amor e exemplo deixaram marcas profundas que me guiaram até aqui.

Agradeço, também, aos meus amigos de fora do mestrado pelo encorajamento e pela força em momentos importantes, assim como aos meus amigos espirituais, cuja presença ao longo dessa jornada foi marcada por mensagens de apoio e inspiração.

Por fim, gostaria de fazer um agradecimento especial nesta dissertação ao meu querido amigo Alexander. Embora nos deixe saudades, sua memória permanece viva em meu coração. Onde quer que esteja, receba minha profunda gratidão por tudo o que me proporcionou em vida. Obrigada por tudo, Amigo, e que você esteja em Paz.

Agradeço à PUC Minas, especialmente ao Programa de Pós-Graduação em Informática (PPGINF), à sua Coordenação, Professores e Colaboradores pelo apoio e ambiente acolhedor. Meu agradecimento ao Laboratório de Inteligência Computacional Aplicada (LICAP) e ao Computer Architecture and Parallel Processing Team (CArT) pela infraestrutura essencial ao desenvolvimento deste trabalho, bem como ao programa de bolsas, que me proporcionou a bolsa CAPES, fundamental para a realização deste projeto. Agradeço também à Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG, códigos APQ-03076-18 e APQ-05058-23) e ao Conselho Nacional de Desenvolvimento

Científico e Tecnológico (CNPq, códigos 311573/2022-3 e 311697/2022-4) pelo apoio financeiro e pelas oportunidades concedidas, sem os quais este trabalho não seria possível.

Por fim, a Todos, meu mais sinceros e profundos agradecimentos!

"Programação é como uma arte marcial intelectual, onde a disciplina e a prática constante levam à maestria."

Kathy Sierra

RESUMO

Avanços significativos têm sido observados no campo da aprendizagem de máquina, impulsionados pelo crescimento exponencial dos dados. No entanto, um desafio crucial em problemas de aprendizado é a eficiente seleção de instâncias de dados para mitigar o impacto de informações redundantes ou irrelevantes. O algoritmo *Ant Colony Optimization* (ACO), inspirado no comportamento das formigas na busca por alimentos, é amplamente empregado para essa finalidade. No entanto, sua implementação sequencial pode ser custosa do ponto de vista computacional, o que impulsiona a necessidade de técnicas de paralelização. Este estudo visa comparar três abordagens de paralelização do algoritmo ACO (CPU paralelizada, CPU vetorizada e GPU) em relação ao desempenho computacional e precisão preditiva. Métricas como Ganho e Eficiência foram introduzidas para avaliar o desempenho computacional destas três versões, enquanto as métricas *F-Measure*, *Precision* e *Recall* foram aplicadas para avaliar a qualidade das predições. Foram utilizadas quinze bases de dados públicas no estudo, variando em tamanho e características dos atributos. Os resultados revelaram que a paralelização do ACO, tanto na CPU quanto na GPU, resultou em reduções relevantes nos tempos de execução em comparação com a implementação sequencial. A GPU demonstrou os maiores ganhos de desempenho, superando tanto a versão vetorizada quanto a paralelizada pela CPU, com um ganho médio de 41,84 vezes. A versão vetorizada pela CPU obteve um ganho de 17,69 vezes, enquanto a versão paralelizada pela CPU apresentou um ganho de 16,63 vezes. Em todos os cenários, foram observadas melhorias consideráveis, especialmente em conjuntos de dados mais extensos, evidenciando a eficácia das abordagens paralelizadas em comparação com a implementação sequencial. Os resultados mostraram que todas as versões paralelizadas conseguiram manter a capacidade de selecionar as melhores instâncias de forma comparável ou até melhor do que à versão sequencial. Isso sugere que a paralelização não comprometeu a qualidade das predições. Essa consistência nas métricas destaca a importância de escolher entre as três versões baseadas no contexto específico da aplicação, considerando o tamanho dos dados e os recursos computacionais disponíveis. Este estudo busca aprimorar o desempenho computacional em problemas de otimização combinatória e aprendizado de máquina, além de promover a integração entre essas áreas.

Palavras-chave: Aprendizado de máquina, Seleção de instâncias, Otimização por Colônia de Formiga, CPU, Paralelização, Vetorização, GPU, Desempenho Computacional.

ABSTRACT

Significant advancements have been observed in the field of machine learning, driven by the exponential growth of data. However, a crucial challenge in learning problems is the efficient selection of data instances to mitigate the impact of redundant or irrelevant information. The Ant Colony Optimization (ACO) algorithm, inspired by the behavior of ants in their search for food, is widely used for this purpose. However, its sequential implementation can be computationally expensive, which drives the need for parallelization techniques. This study aims to compare three parallelization approaches of the ACO algorithm (parallelized CPU, vectorized CPU, and GPU) in terms of computational performance and predictive accuracy. Metrics such as Gain and Efficiency were introduced to assess the computational performance of these three versions, while the F-Measure, Precision, and Recall metrics were applied to evaluate the quality of predictions. Twelve public datasets, varying in size and attribute characteristics, were used in the study. The results revealed that ACO parallelization, both on the CPU and GPU, led to significant reductions in execution times compared to the sequential implementation. The GPU demonstrated the highest performance gains, outperforming both the vectorized and CPU-parallelized versions, with an average gain of 41.84 times. The CPU vectorized version achieved a gain of 17.69 times, while the CPU-parallelized version showed a gain of 16.63 times. In all scenarios, substantial improvements were observed, particularly in larger datasets, highlighting the effectiveness of the parallelized approaches compared to the sequential implementation. The results also showed that all parallelized versions were able to maintain the ability to select the best instances comparably or even better than the sequential version. This suggests that parallelization did not compromise the quality of the predictions. This consistency in the metrics underscores the importance of choosing among the three versions based on the specific context of the application, considering the size of the data and available computational resources. This study seeks to enhance computational performance in combinatorial optimization and machine learning problems, as well as promote integration between these areas.

Keywords: Machine Learning, Instance Selection, Ant Colony Optimization, CPU, Parallelization, Vectorization, GPU, Computational Performance.

LISTA DE FIGURAS

FIGURA 1 – Processo de seleção de instância 19

FIGURA 2 – Funcionamento do ACO para a Seleção de Instâncias 27

FIGURA 3 – Classificações de Flynn 29

FIGURA 4 – Funcionamento do ACO Paralelizado por Núcleos da CPU 49

FIGURA 5 – Funcionamento do ACO pela GPU 55

FIGURA 6 – Visão geral da metodologia do trabalho 56

FIGURA 7 – Ganhos obtidos pela ***paralelização*** e ***vetorização*** em cada núcleo da CPU em cada base de dados 67

FIGURA 8 – Eficiência obtida pela ***paralelização*** e ***vetorização*** em cada núcleo da CPU em cada base de dados 68

FIGURA 9 – Ganho e Eficiência obtidos pela ***paralelização*** e ***vetorização*** em cada núcleo da CPU 69

FIGURA 10 – Quantidade de Instâncias x Tempo em cada versão 71

FIGURA 11 – Resultados obtidos, considerando-se a métrica F-measure, a partir das bases pelos algoritmos de AM 77

LISTA DE TABELAS

TABELA 1 – Quantidade de artigos selecionados em cada fase de seleção 35

TABELA 2 – Descrição das bases de dados quanto ao número de instâncias e tipos de atributos 58

TABELA 3 – Tempo de execução obtido pela **paralelização** em múltiplos núcleos da CPU. Os valores entre colchetes representam os valores do desvio padrão. 64

TABELA 4 – Tempo de execução obtido através da **paralelização pela vetorização** em múltiplos núcleos da CPU. Os valores entre colchetes representam os valores do desvio padrão. 65

TABELA 5 – Tempo de execução obtido através da paralelização pela GPU. Os valores entre colchetes representam os valores do desvio padrão. 66

TABELA 6 – Análise Comparativa das Métricas de acesso ao *Hardware* utilizando a Ferramenta Perf 73

TABELA 7 – Quantidade total de instâncias reduzidas na versão paralelizada pela CPU em função do número de núcleos 74

TABELA 8 – Quantidade total de instâncias reduzidas na versão vetorizada paralelizada pela CPU em função do número de núcleos 75

TABELA 9 – Quantidade total de instâncias reduzidas na versão paralelizada pela GPU 76

TABELA 10 – Tabela Comparativa de Equivalência e Diferença entre Versões do ACO com o Teste T de *Student*. O número total de base de dados é 15. . . . 79

TABELA 11 – Tabela Comparativa de Equivalência e Diferença entre Versões do ACO com o Teste T de *Student* 80

TABELA 12 – Publicações relacionadas à pesquisa. 85

TABELA 13 – Valores de predição dos algoritmos de AM nas Bases de Dados em %. Os valores entre colchetes representam os valores do desvio padrão. 96

TABELA 14 – Tabela comparativa entre versões do ACO. O número total de base

de dados é 15. 107

TABELA 15 – Tabela Comparativa de Equivalência e Diferença entre Versões do ACO com o Teste T de *Student* 108

LISTA DE QUADROS

QUADRO 1 – Quantidade de Tipos de Ambiente utilizados nos artigos selecionados 36
QUADRO 2 – Linguagens utilizadas nos artigos selecionados 42
QUADRO 3 – Bibliotecas e Linguagens utilizadas nos artigos selecionados 43
QUADRO 4 – Tabela com os algoritmos de AM e bibliotecas implementadas no *pipeline* 59

LISTA DE ABREVIATURAS E SIGLAS

ACO – *Ant Colony Optimization*

AM – Aprendizado de Máquina

ANN – *Artificial Neural Network*

AS – *Ant System*

CPU – Unidade Central de Processamento ou *Central Processing Unit*

CSV – *Comma-Separated Values*

CUDA – *Compute Unified Device Architecture*

GPU – Unidade de Processamento Gráfico ou *Graphic Processing Unit*

IA – Inteligência Artificial

JIT – *Just-In-Time*

KNN – *K-nearest neighbors*

MIMD – *Multiple Instruction Stream-Multiple Data Stream*

MISD – *Multiple Instruction Stream-Single Data Stream*

MPI – *Message Passing Interface*

OpenMP – *Open Multi-Processing*

SI – Seleção de Instâncias

SIMD – *Single Instruction, Multiple Data*

SIMT – *Single Instruction, Multiple Threads*

SISD – *Single Instruction Stream-Single Data Stream*

SM – *Streaming Multiprocessors*

SVM – *Support Vector Machine*

TBB – *Threading Building Blocks*

SUMÁRIO

1 INTRODUÇÃO **9**
1.1 Problema 11
1.2 Hipóteses 11
1.3 Objetivos 12
1.3.1 Objetivo geral 12
1.3.2 Objetivo específicos 12
1.4 Justificativa 13
1.5 Contribuições da dissertação 14
1.6 Organização da dissertação 14

2 REFERENCIAL TEÓRICO **16**
2.1 Pré-processamento 16
2.2 O Algoritmo Ant Colony Optimization 19
2.2.1 O surgimento do Ant System 19
2.2.2 Ant Colony Optimization 20
2.2.3 ACO na Seleção de Instâncias 25
2.2.4 Passos do Algoritmo 25
2.3 Arquiteturas de Computação Paralela 26
2.3.1 Classificações de Flynn 26
2.3.2 Paralelização pela CPU 28
2.3.3 Vetorização pela CPU 30
2.3.4 Paralelização pela GPU 30
2.3.5 Escalabilidade 31

3 TRABALHOS RELACIONADOS SOBRE OS TIPOS DE PARALELIZAÇÃO DO ALGORITMO ACO **33**
3.1 Questões de Pesquisa 33
3.2 Critérios de Seleção e Exclusão 34
3.2.1 Critérios de Seleção 34

3.2.2 Critérios de Exclusão 35
3.2.2.1 *Duplicação de Dados* 35
3.2.2.2 *Idioma e Acesso* 35
3.2.2.3 *Relevância Temática* 35
3.3 Discussão 36
3.3.0.1 CPU 37
3.3.0.2 GPU 38
3.3.0.3 Vetorização CPU 38
3.4 Conclusão 43

4 ESTRATÉGIAS DE PARALELIZAÇÃO E VETORIZAÇÃO DO ACO 45
4.1 Paralelização pela CPU 45
4.2 Paralelização vetorizada pela CPU 48
4.3 Paralelização pela GPU 51

5 MATERIAIS E MÉTODOS 56
5.1 Descrição das Bases de dados 57
5.2 Algoritmos de Aprendizado de Máquina 58
5.3 Métricas de avaliação 59
5.3.1 Métricas para avaliação de qualidade dos algoritmos de Aprendizado de Máquina 59
5.3.2 Métricas para avaliação de desempenho 60
5.3.3 Métricas estatísticas 61
5.4 Avaliação de Escalabilidade 62
5.5 Ambiente de Hardware 62

6 RESULTADOS E DISCUSSÕES 63
6.1 Resultados de desempenho das paralelizações 63
6.2 Análise dos Contadores de Hardware 72
6.3 Resultados da redução de instâncias pelo ACO 74
6.4 Resultados de predição 76

7 CONSIDERAÇÕES FINAIS 82
7.1 Limitações e trabalhos futuros 84

8 PUBLICAÇÕES RELACIONADAS AO TRABALHO 85

REFERÊNCIAS 86

APÊNDICE A – DESEMPENHO PREDITIVO DOS MODELOS DAS BASES PÚBLICAS 96

APÊNDICE B – DESEMPENHO ESTATÍSTICO DOS MODELOS PELO TESTE T *STUDENT* 107

1 INTRODUÇÃO

Nos últimos anos, a área de Aprendizado de Máquina (AM) tem alcançado resultados satisfatórios em problemas difíceis como reconhecimento de fala, problemas de visão computacional, como o reconhecimento de objetos em imagens e vídeos, processamento de linguagem natural (BENIN, 2023; OLIVEIRA; MELO, 2023; LECUN, 2015; SCHMIDHUBER, 2015), bem como em problemas da Bioinformática ou até mesmo auxiliar no diagnóstico de doenças (SILVA et al., 2025; SAILUNAZ et al., 2023; BADAWY; RAMADAN; HEFNY, 2023; VEDANA et al., 2024). Esses avanços, no entanto, estão intimamente relacionados ao aumento exponencial no volume de dados disponíveis.

Portanto, o crescente volume de dados, impulsionado pelo avanço contínuo do *hardware* de armazenamento e das técnicas de coleta, representam um desafio para as abordagens de mineração de dados. A existência de atributos ou instâncias redundantes e irrelevantes nos conjuntos de dados pode comprometer a qualidade da classificação (AKINYELU, 2020). A Seleção de Instâncias (SI), como uma etapa essencial do pré-processamento, contribui para minimizar esse problema ao remover instâncias desnecessárias, melhorando a eficiência da aprendizagem e simplificando o processo de modelagem (TSAI; et al, 2021; TIWARI et al., 2024). Esse processo visa selecionar um subconjunto representativo dos dados, reduzindo a redundância e eliminando informações irrelevantes ou inconsistentes, o que permite extrair conhecimento relevante e otimizar o tempo de processamento (CHENG; CHU; ZHANG, 2021; TIWARI et al., 2024). Diferentes algoritmos podem ser empregados para essa tarefa, como o DROP *(Decremental Reduction Optimization Procedure)*, o CNN *(Condensed Nearest Neighbor)* e o ICF *(Iterative Case Filtering)*, *K Means Clustering*, entre outros (ZHAI; QI; ZHANG, 2021; SAHA et al., 2022). Neste trabalho, foi aplicado o algoritmo de SI baseado na Otimização por Colônia de Formigas - *Ant Colony Optimization* (ACO).

O Algoritmo Colônia de Formigas, do inglês ACO, uma técnica de otimização meta-heurística inspirada no comportamento de colônias de formigas reais na busca por alimento, é amplamente utilizado no contexto de seleção de instâncias/atributos. A ideia principal é empregar formigas artificiais colaborativas para encontrar o caminho mais curto entre dois pontos em um grafo. Segundo o artigo de Anwar, Salama e Abdelbar (2015b),o ACO é uma abordagem eficiente para lidar com problemas combinatórios, apresentando bom desempenho em termos de qualidade da solução e tempo de execução em problemas de pequena e média escala. Apesar de sua natureza heurística e da ausência de garantias de tempo polinomial no pior caso, o ACO apresentou sucesso na resolução

de problemas de classificação (SALAMA; ABDELBAR; ANWAR, 2016). Estudos mais recentes, como o de Liang et al. (2024) e Jain et al. (2023), reforçam que o ACO continua sendo uma abordagem robusta e eficaz para problemas de otimização complexos, incluindo seleção de instâncias e atributos.

Também evidenciado por Medeiros (2022), Goncalves et al. (2024), Magalhães et al. (2024), o ACO apresentou-se eficiente na seleção das melhores instâncias. No entanto, à medida que o número de instâncias cresce (acima de 4 mil), o tempo de execução pode se tornar impraticável, o que evidencia sua elevada complexidade computacional na prática. Dessa forma, técnicas de paralelismo que utilizam tanto a Unidade Central de Processamento ou *Central Processing Unit* (CPU) quanto a Unidade de Processamento Gráfico ou *Graphic Processing Unit* (GPU) podem reduzir este tempo, tornando viável a execução do algoritmo ACO.

Neste contexto, a computação paralela consiste na execução simultânea de várias tarefas, e.g., *Threads*, por múltiplos núcleos em uma máquina paralela (AL-SHAFEI; ZAREIPOUR; CAO, 2022). Com a disponibilidade cada vez maior de arquiteturas *multi-core* e *many-core*, a computação paralela tem sido utilizada para o crescimento contínuo no desempenho de diversos tipos de aplicações. Para que o desempenho de uma aplicação aumente com as futuras gerações de processadores e aceleradores, é necessário que uma parte significativa de sua computação seja feita com algoritmos paralelos escaláveis (HWU, 2014). Experimentos mostram que a paralelização na CPU alcança uma aceleração quase linear, tornando-se uma abordagem prática para aplicações do mundo real (ALHENAWI et al., 2023).

Uma abordagem adicional de paralelização na CPU é a vetorização, essencial para alcançar um desempenho computacional elevado. Essa técnica permite que o código utilize as capacidades SIMD do *hardware*, proporcionando ganhos de velocidade (LIN; WU; BHATTACHARYYA, 2018). A vetorização destaca-se como uma técnica fundamental para o aprimoramento do processamento de dados em bancos de memória, além de explorar plenamente a eficiência do SIMD, resultando em melhorias de desempenho (POLYCHRONIOU; RAGHAVAN; ROSS, 2015). A principal vantagem da vetorização é que ela permite o processamento de múltiplos elementos de dados em uma única instrução, reduzindo o número de ciclos de clock necessários para os cálculos (XAVIER et al., 2024).

Por outro lado, GPUs são processadores paralelos de baixo custo com muitos núcleos, capazes de executar milhares de *Threads* ou núcleos de processamento simultaneamente em uma configuração SIMD (SATO et al., 2014). Com a GPU de baixo custo disponível em computadores comuns, tornou-se possível resolver problemas de otimização usando a paralelização (TSUTSUI; COLLET, 2013). A paralelização em GPU pode alcan-

çar ganhos de até 100x em comparação com implementações sequenciais, dependendo da complexidade do problema e da eficiência da implementação (ZENG et al., 2023). Através do *Compute Unified Device Architecture* (CUDA), introduzido pela NVIDIA™, as GPUs são concebidas como coprocessadores projetados para executar um grande número de processadores em paralelo. Em CUDA, cada núcleo pode acessar diversos recursos de memória nas GPUs, como memória global, memória compartilhada e registradores, entre outros (GAO et al., 2016).

1.1 Problema

Diante do crescimento exponencial de dados em diferentes áreas e da necessidade de desenvolver soluções eficientes para processamento e análise de grandes volumes de informações, surgem as seguintes Questões de Pesquisa (QP):

QP 1: Quais são os impactos da paralelização do algoritmo ACO em arquiteturas de CPU, CPU vetorizado e GPU em termos de tempo de execução e eficiência para a SI em grandes bases de dados?

QP 2: A aplicação de técnicas de paralelização (CPU, CPU vetorizado e GPU) no algoritmo ACO mantém a precisão dos modelos preditivos, medida por métricas como *F-Measure*, *Precision* e *Recall*, comparável à versão sequencial do algoritmo?

QP 3: Quais são os desafios e limitações na implementação de versões paralelizadas do algoritmo ACO em arquiteturas *multi-core* e GPUs, e como essas limitações afetam o desempenho geral?

QP 4: As implementações paralelizadas do algoritmo ACO em Python conseguem oferecer ganhos de tempo relevantes em comparação com a versão sequencial?

1.2 Hipóteses

Este trabalho é baseado fundamentalmente nas seguintes hipóteses:

Hipótese 1: A paralelização do algoritmo ACO em arquiteturas de CPU, CPU vetorizado e GPU é capaz de reduzir substancialmente o tempo de execução em relação à versão sequencial, mantendo níveis de eficácia comparáveis para a SI em grandes bases de dados.

Hipótese 2: A aplicação de técnicas de paralelização em Python, mesmo considerando a sobrecarga de implementação e comunicação entre *threads* ou processos, consegue proporcionar ganhos de desempenho relevantes em termos de tempo de execução e eficiência computacional.

Hipótese 3: Implementações do algoritmo ACO em GPU utilizando CUDA são mais eficientes em termos de tempo de execução em comparação com implementações multi-core em CPU.

Hipótese 4: A implementação paralelizada do algoritmo ACO em Python, além de proporcionar ganhos de tempo, mantém a portabilidade e a flexibilidade da linguagem, tornando-se uma solução viável para aplicações em cenários de alto desempenho computacional.

1.3 Objetivos

1.3.1 Objetivo geral

O objetivo desta dissertação é investigar e desenvolver soluções paralelas para reduzir o tempo de execução do algoritmo *Ant Colony Optmization* (ACO), para o problema de seleção de instâncias. Para atingir esse objetivo, realiza-se uma análise comparativa entre as três abordagens de paralelização do ACO (CPU, CPU vetorizado e GPU) e a versão sequencial. Essa análise avaliará o desempenho em termos de tempo de execução, ganho de velocidade e eficiência computacional de cada versão paralelizada. Adicionalmente, busca assegurar a manutenção da qualidade preditiva do algoritmo, utilizando as métricas *F-Measure*, *Precision* e *Recall*, obtidas a partir dos algoritmos de AM *Artificial Neural Network* (ANN), *K-nearest neighbors* (KNN), *Random Forest* e *Support Vector Machine* (SVM).

1.3.2 Objetivo específicos

Os objetivos específicos deste trabalho são:

- Investigar e implementar versões paralelizadas do algoritmo *Ant Colony Optmization* (ACO) em CPU, CPU vetorizado e GPU, analisando o impacto de cada abordagem na redução do tempo de execução e na eficiência computacional.
- Desenvolver estratégias que minimizem os custos computacionais do ACO sem comprometer a qualidade das predições, garantindo a preservação das informações relevantes dos dados originais.
- Avaliar experimentalmente as implementações paralelizadas em diferentes bases de dados com variadas dimensões e níveis de complexidade, verificando sua robustez e desempenho em distintos cenários.
- Comparar as soluções desenvolvidas em relação à versão sequencial, mensurando o ganho de velocidade e a eficiência computacional para validar a eficácia das abordagens propostas.

1.4 Justificativa

A crescente complexidade e volume dos dados disponíveis na atualidade exigem abordagens inovadoras e eficientes para extrair informações relevantes e apoiar a tomada de decisões. Nesse contexto, o papel do âmbito de pré-processamento de dados torna-se crucial, pois ele não apenas influencia diretamente a qualidade dos modelos de AM, mas também possibilita o uso de algoritmos que seriam inviáveis com dados brutos ou desorganizados (ALBAHRA et al., 2023; RAO; SAIKRISHNA; SUPRIYA, 2023; LUENGO et al., 2020; SAHA et al., 2022). Assim, a preparação e transformação dos dados de maneira adequada constituem um alicerce fundamental para o sucesso em tarefas como mineração de dados e predição.

Entre as diversas técnicas de pré-processamento, destaca-se a SI, que busca reduzir o tamanho dos conjuntos de dados eliminando redundâncias, ruídos e instâncias irrelevantes, mantendo ou mesmo melhorando a capacidade de generalização dos modelos de AM (LIU; MOTODA, 2002; ANWAR; SALAMA; ABDELBAR, 2015a; TIWARI et al., 2024; SAHA et al., 2022). Esse processo não apenas melhora a qualidade dos dados, mas também reduz os custos computacionais, uma vez que modelos treinados com conjuntos menores são mais eficientes em termos de tempo e recursos. No entanto, por se tratar de um problema NP-Completo, sua solução exata é inviável em cenários com grande quantidade de instâncias, o que torna as heurísticas, como o ACO, uma alternativa atrativa (PAPADIMITRIOU, 1982; DORIGO; CARO, 1999).

A escolha do ACO como meta-heurística baseia-se em sua capacidade de explorar espaços de busca complexos de forma eficiente, utilizando um comportamento coletivo e adaptativo inspirado nas colônias de formigas (DORIGO; CARO, 1999). Para este trabalho, o ACO foi implementado em Python, uma linguagem amplamente reconhecida por sua aplicabilidade nas áreas de Inteligência Artificial (IA) e ciência de dados, bem como por seu ecossistema robusto de bibliotecas e ferramentas que facilitam o desenvolvimento de algoritmos complexos (HILL, 2020; STATISTA, 2024). Embora as linguagens C/C++ ofereçam melhor desempenho bruto e maior controle para paralelização em arquiteturas de alto desempenho, a escolha do Python foi motivada pela sua flexibilidade, facilidade de desenvolvimento e integração com bibliotecas otimizadas para computação paralela e vetorização. Ademais, com o avanço da computação paralela, algoritmos como o ACO podem ser implementados de maneira a aproveitar a arquitetura de CPUs *multi-core* e GPUs, o que não apenas reduz o tempo de execução, mas também aumenta a qualidade das soluções encontradas (POLYCHRONIOU; RAGHAVAN; ROSS, 2015; CHEN; SUN; WANG, 2008; FANG; WU, 2010a; AL-SHAFEI; ZAREIPOUR; CAO, 2022; XAVIER et al., 2024).

A eficiência de tais implementações está diretamente relacionada ao uso de técnicas

como a vetorização, que alavanca as capacidades SIMD das CPUs modernas para realizar operações simultâneas em grandes volumes de dados (BORIN, 2024). Por outro lado, o uso de GPUs potencializa o desempenho ao explorar o paralelismo massivo que caracteriza essas arquiteturas (MCCLANAHAN, 2011; ZENG et al., 2023).

Dessa forma, o presente trabalho justifica-se pela necessidade de explorar, combinar e comparar o impacto de diferentes técnicas de pré-processamento de dados, como a SI, em conjunto com avanços recentes em algoritmos heurísticos e tecnologia de computação paralela. Ao investigar soluções baseadas no ACO, paralelismo em CPU, GPU e vetorizacão, esta pesquisa busca oferecer contribuições relevantes para a área de ciência de dados e AM, atendendo às demandas crescentes de eficiência e escalabilidade em aplicações do mundo real.

1.5 Contribuições da dissertação

Este estudo apresenta como principal contribuição a comparação das três versões de paralelização do algoritmo ACO em Python: CPU, CPU vetorizado e GPU. A análise do desempenho de cada tipo de arquitetura nessas implementações visa aprimorar a eficiência no processamento de dados complexos. Para garantir a reprodutibilidade e incentivar novas pesquisas, as implementações paralelizadas do ACO estão disponíveis nos seguintes links do *GitHub*: Sequencial*, CPU†, CPU vetorizado‡ e GPU§.

Além de contribuir para a comunidade científica, esta pesquisa oferece uma abordagem mais eficiente e escalável para a resolução de problemas de otimização, com aplicações potenciais em diversas áreas, como análise de dados, IA e AM. As contribuições deste estudo têm o potencial de inspirar novas pesquisas e desenvolvimentos, beneficiando tanto o meio acadêmico quanto o setor industrial.

Adicionalmente, este trabalho estabelece uma conexão entre duas áreas da computação: desempenho computacional e IA, evidenciando a importância da integração entre esses campos para avanços tecnológicos e científicos.

1.6 Organização da dissertação

Esta dissertação segue a seguinte estrutura: no Capítulo 2, é abordado o embasamento teórico sobre a SI, a história da criação do algoritmo ACO até o seu uso na SI, e uma explicação sucinta de como cada tipo de paralelização funciona em diferentes

*https://encurtador.com.br/db1eT
†https://encurtador.com.br/Ou0kl
‡https://encurtador.com.br/4IZ07
§https://encurtador.com.br/df30c

arquiteturas de computadores. O Capítulo 3 cita trabalhos correlatos, destacando o uso do ACO e outros estudos pertinentes. O Capítulo 4 detalha como cada uma das estratégias de paralelização (CPU paralelizada, CPU vetorizada e GPU) foram implementadas no ACO. A metodologia é discutida em profundidade no Capítulo 5, abordando aspectos como a especificação do ambiente de *hardware* utilizado para realizar os experimentos, a descrição das bases de dados utilizadas neste trabalho, os algoritmos de AM: ANN, KNN, *Random Forest* e SVM, as métricas de avaliação de AM e o desempenho computacional. Os resultados são apresentados no Capítulo 6, e as conclusões e sugestões para pesquisas futuras estão no Capítulo 7.

2 REFERENCIAL TEÓRICO

Este capítulo explora os princípios fundamentais da Seleção de Instâncias (SI), abrangendo técnicas de pré-processamento de dados e a aplicação do algoritmo ACO para otimização combinatória, desde sua concepção no *Ant System* até variantes como o *ANT-IS*. Além disso, investiga arquiteturas de computação paralela (CPU e GPU), classificadas pela taxonomia de Flynn, e examina estratégias de paralelização, incluindo vetorização, *multi-core* e CUDA, bem como aspectos de escalabilidade, tanto fraca quanto forte.

2.1 Pré-processamento

A qualidade dos dados é extremamente importante, pois pode influenciar a capacidade do modelo de aprender e, em última análise, se tornar generalizável (ALBAHRA et al., 2023). A preparação de dados é uma etapa fundamental para garantir a precisão e a eficácia das previsões na ciência de dado (RAO; SAIKRISHNA; SUPRIYA, 2023). O pré-processamento de dados é capaz de adaptar os dados às exigências colocadas por cada algoritmo de AM, possibilitando processar dados que seriam inviáveis de outra forma (LUENGO et al., 2020). O tipo de pré-processamento também depende do tipo de dado e do respectivo algoritmo de AM empregado (ALBAHRA et al., 2023). Ao considerar que métodos bem conhecidos e amplamente utilizados de AM são frequentemente envolvidos na mineração de dados, a importância do pré-processamento dos dados em mineração de dados pode ser facilmente reconhecida (ALEXANDROPOULOS; KOTSIANTIS; VRAHATIS, 2019).

As técnicas de pré-processamento incluem Balanceamento de Dados, Codificação de Atributos Categóricas, Imputação de Valores Ausentes, Normalização, Redução de *Outliers*, Redução de Ruídos, Seleção de Atributos e Seleção de Instâncias. A seguir, cada técnica é brevemente explicada.

- *Balanceamento de Dados*: Técnica empregada para lidar com conjuntos de dados desbalanceados em problemas de classificação, O *Oversampling* é uma das técnicas de balanceamento que amplia a quantidade de instâncias da classe menos representada para equilibrar a distribuição dos dados, sendo o SMOTE *(Synthetic Minority Over-sampling Technique)* um dos métodos mais utilizados. Já o *Undersampling* reduz o número de instâncias da classe majoritária, selecionando um subconjunto dos dados a fim de minimizar possíveis vieses. Além disso, é possível combinar ambas as

abordagens, aplicando simultaneamente *Oversampling* à classe minoritária e *Undersampling* à majoritária, promovendo um balanceamento mais eficaz (CORDEIRO, 2020).

- *Codificação de Atributos Categóricos*: Técnica que converte dados categóricos em formatos numéricos ou binários, facilitando sua utilização em algoritmos AM que operam com atributos discretos. Exemplos comuns incluem o *One-Hot Encoding*, que transforma atributos categóricos em atributos binários, e o *Label Encoding*, que atribui números inteiros a cada categoria (KAMIRAN; CALDERS, 2012).

- *Imputação de Valores Ausentes*: A imputação de valores ausentes é uma técnica utilizada para preservar a integridade dos dados, substituindo valores ausentes por estimativas apropriadas (ALBAHRA et al., 2023). Os métodos mais comuns incluem a imputação pela média, mediana e moda, que são simples e eficazes. Outras abordagens mais sofisticadas incluem a imputação por KNN, que utiliza os vizinhos mais próximos para estimar valores, a regressão múltipla, que prevê valores com base em outros atributos, e a imputação múltipla, que gera várias versões imputadas para reduzir incertezas nos dados (GABR; HELMY; ELZANFALY, 2023).

- *Normalização*: Técnica que ajusta os valores de atributos numéricos para um intervalo comum, geralmente entre 0 e 1. A normalização também aprimora a convergência de modelos que utilizam Gradiente Descendente* (ALBAHRA et al., 2023).

- *Redução de Outliers*: Técnica que identifica e trata valores atípicos que podem distorcer análises e modelos. Isso é feito por meio da remoção, ajuste ou substituição dos *Outliers*, visando melhorar a qualidade dos dados e a precisão dos resultados (ALBAHRA et al., 2023).

- *Redução de Ruídos*: Técnica que visa eliminar ou minimizar dados irrelevantes ou errôneos, como distorções ou informações inconsistentes, que podem prejudicar a análise e o desempenho dos modelos de AM. Seu objetivo é suavizar ou filtrar o ruído presente nos dados, seja ele aleatório ou determinístico. Essa técnica é muito utilizada em contextos em que os dados reais são propensos ao ruído, como em medições de sensores (IBIAS et al., 2024).

- *Seleção de Atributos*: A Seleção de Atributos, geralmente aplicada como uma etapa de pré-processamento de dados em AM e mineração de dados, tem como objetivo identificar e selecionar um número reduzido de atributos que descrevem o conjunto de dados tão bem quanto ou até melhor do que o conjunto original de atributos (LIU;

*Algoritmo de Otimização empregado no treinamento de modelos de AM e Redes Neurais.

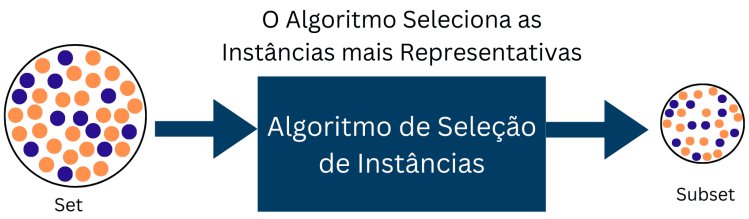
MOTODA, 2008). Essa técnica busca eliminar os atributos redundantes ou irrelevantes, contribuindo para a melhoria da performance dos modelos de aprendizado (TAE; BROILLET-SCHLESINGER; KIM, 2024).

- *Seleção de Instâncias*: A técnica de seleção de instâncias (SI) visa obter um subconjunto da base de dados original, eliminando instâncias irrelevantes, ruidosas e redundantes, mantendo uma acurácia próxima a do conjunto completo (LIU; MOTODA, 2002). Isso melhora a qualidade dos dados, reduz os custos computacionais e gera uma amostra representativa e mínima (CENIKJ et al., 2022). Como a SI requer a busca por todas as combinações possíveis de instâncias, é um problema NP-Completo (PAPADIMITRIOU, 1982), necessitando de soluções heurísticas.

  A SI consiste em escolher um subconjunto apropriado do conjunto de treinamento completo para construir um modelo de classificação (ANWAR; SALAMA; ABDELBAR, 2015a). O objetivo é obter um modelo mais eficaz, capaz de prever com precisão as classes de instâncias não vistas durante o treinamento (TIWARI et al., 2024). A SI contribui para o aprimoramento do desempenho computacional ao reduzir o tempo de execução do algoritmo. Ao selecionar um subconjunto representativo de instâncias, a SI ajuda a otimizar o processo de treinamento, acelerando o algoritmo sem comprometer sua precisão (MELO; DUITAMA; ARIAS, 2022).

  O processo de SI começa com um conjunto de dados de treinamento T, que é o conjunto original. Cada instância nesse conjunto é então avaliada com base em critérios como relevância, representatividade e redundância. A relevância identifica instâncias informativas para o problema, como instâncias que reduzem a incerteza do modelo. A representatividade assegura que o subconjunto selecionado reflita a distribuição geral dos dados, usando técnicas probabilísticas. A redundância busca evitar informações repetitivas, removendo instâncias muito similares ou correlacionadas. Com base nessas avaliações, um subconjunto S de instâncias é selecionado para formar um novo conjunto de dados de treinamento. Esse subconjunto deve conter instâncias mais relevantes e representativas para o problema em questão. A Figura 1 ilustra o processo de SI, no qual um conjunto de dados inicial é processado por um algoritmo de SI para identificar as amostras mais representativas. O resultado é um subconjunto reduzido, que mantém a diversidade e representatividade dos dados originais. Esse subconjunto pode, então, ser utilizado para treinar o modelo de AM, aumentando a eficiência e mantendo a capacidade de generalização (OLVERA-LÓPEZ et al., 2010).

Figura 1: Processo de seleção de instância



Fonte: Elaborado pelo autor

2.2 O Algoritmo Ant Colony Optimization

2.2.1 O surgimento do Ant System

Em 1991, Dorigo, Maniezzo e Colorni (1991) propuseram um sistema de algoritmos com um novo método heurístico chamado *Ant System* (AS) para resolver problemas de otimização combinatória. Esse sistema utiliza computação distribuída, *feedback* positivo e heurística gananciosa construtiva. Inspirado no comportamento das formigas reais, onde trilhas de feromônios são usadas para comunicar informações sobre caminhos, o AS provou ser eficaz ao ser aplicado ao problema do caixeiro viajante, oferecendo soluções de alta qualidade de forma rápida.

Os algoritmos do AS são derivados do estudo de colônias de formigas artificiais. Ao contrário das formigas reais, que são consideradas quase cegas, as formigas artificiais, propostas por Dorigo, Maniezzo e Colorni (1991), possuem alguma memória e interagem em um ambiente discreto. Assim como as formigas reais se comunicam através de trilhas de feromônios, o AS utiliza a mesma abordagem. Os algoritmos do AS, como o *Ant-Density*, *Ant-Quantity* e *Ant-Cycle*, são modelos simples de agentes (formigas) que interagem localmente para resolver problemas complexos de forma distribuída. A principal diferença está na forma como a trilha de feromônios é atualizada durante o processo de busca.

O artigo de Colorni, Dorigo e Maniezzo (1992) aborda a continuação da pesquisa sobre o uso do sistema AS para otimização, focando principalmente no problema do caixeiro viajante. O estudo continua a analisar os três algoritmos *Ant-Density*, *Ant-Quantity* e *Ant-Cycle* cujo desempenho se mostrou dependente da configuração de parâmetros como sensibilidade às trilhas, sensibilidade à distância, taxa de evaporação das trilhas de feromônios e quantidade de feromônio depositada. A pesquisa busca avaliar como esses algoritmos se comportam com um maior número de cidades no problema do caixeiro viajante e como o número de formigas afeta seu desempenho. Embora ainda não haja uma análise matemática detalhada dos parâmetros ideais, os resultados das simulações foram promissores, sugerindo que a abordagem poderia ser aplicada a uma variedade de

problemas de otimização.

Em (DORIGO; MANIEZZO; COLORNI, 1996), uma investigação foi conduzida no sistema AS, aplicando-o ao clássico Problema do Caixeiro Viajante e explorando sua robustez ao ser aplicado a outros problemas de otimização, como o caixeiro viajante assimétrico, a atribuição quadrática e o problema de agendamento *job-shop*. O algoritmo *Ant-Cycle* se destacou, mostrando-se mais eficaz que *Ant-Density* e *Ant-Quantity*. Após o sistema AS mostrar resultados promissores, houve uma intensificação nos estudos, resultando no surgimento de diversas variações de algoritmos, como o *Ant Colony System* (DORIGO; GAMBARDELLA, 1997), o *MAX-MIN Ant System* (STÜTZLE; HOOS, 1998), dentre outros.

2.2.2 Ant Colony Optimization

Em 1999, a meta-heurística *Ant Colony Optmization* (ACO) foi proposta por Dorigo e Caro (1999) para unificar as diversas versões do Algoritmo das Formigas (AS) e facilitar o desenvolvimento de novas aplicações.

O ACO utiliza uma população de formigas artificiais para resolver problemas de otimização por meio da construção incremental de soluções em um grafo que representa o espaço de busca. As decisões das formigas são guiadas por uma combinação de vantagem heurística e feromônios, cuja intensidade reflete o aprendizado coletivo da colônia. Além disso, o mecanismo de evaporação de feromônios evita a convergência prematura e promove a exploração do espaço de busca (DORIGO; CARO, 1999).

Embora cada formiga utilize apenas informações locais para construir soluções, a qualidade global depende da interação mediada pelas trilhas de feromônios. Essas trilhas adaptam a representação do problema conforme as formigas exploram o espaço, mas não atribuem adaptabilidade[†] individual aos agentes.

No contexto deste trabalho, o espaço de busca é modelado como um grafo, em que os vértices representam as instâncias do conjunto de entrada e as arestas correspondem à distância euclidiana entre elas. As formigas partem de vértices distintos, constroem subconjuntos próprios navegando pelo grafo e avaliam suas soluções com um algoritmo de AM. O conjunto de melhor desempenho é selecionado como saída do algoritmo. O modelo de feromônio associa parâmetros τ_{ij} a cada caminho, representando o conhecimento acumulado pela colônia. Trilhas mais intensas indicam componentes promissoras, enquanto a evaporação regula o equilíbrio entre exploração e intensificação (DORIGO; CARO, 1999).

Neste estudo, foi implementado o algoritmo ANT-IS, proposto por Miloud-Aouidate

[†]Capacidade de ajustar comportamento ou características em resposta ao ambiente ou feedback, visando melhorar desempenho ou contribuir para objetivos globais (DORIGO; CARO, 1999).

e Baba-Ali (2013), com base nos princípios do ACO. Nele, uma formiga k, posicionada em uma instância i no instante t, decide seu próximo destino j com base nas Equações 2.1, 2.2 e 2.3. Os passos principais do algoritmo consistem em:

1. Para cada instância, calcule a distância Euclidiana entre ela e todas as outras instâncias do conjunto de dados original.
2. Inicialize a matriz C com -1.
3. Inicialize os valores de feromônio.
4. Crie a mesma quantidade de formigas quantas forem as instâncias da base e posicione cada formiga em uma instância distinta.
5. Cada formiga escolhe sua possível instância de destino de acordo com as Equações 2.1, 2.2 e 2.3.
6. Cada formiga k atualiza a matriz de validação C, dependendo do parâmetro a_j^k:
   (a) if $\mathrm{a}_j^k = 1$, a instância destino j é escolhida e $C(k, j) = 1$.
   (b) Caso contrário, $C(k, j) = 0$, retorne ao passo 5.
7. Cada formiga calcula o valor $\mathrm{L}^k(\mathrm{t})$ que representa o comprimento do caminho feito por ela desde o início até o instante t.
8. Os valores $\mathrm{P}_{ij}(\mathrm{t})$ são calculados de acordo com a Equação 2.3.
9. Os valores $\mathrm{P}_{ij}(\mathrm{t})$ são atualizados conforme Equação 2.4, simulando o depósito e evaporação do feromônio ao final de cada iteração.
10. Se todas as formigas tiverem completado seu *tour*, vá para o passo 11, caso contrário, vá para o passo 5.
11. Dentre os conjuntos soluções de cada formiga, armazene aquele com a mais alta taxa de classificação obtida pelo algoritmo k-NN para k=1.
12. Limpe a lista de instâncias visitadas de cada formiga.

$$FProb_{ij}^2 = Prob_{ij}^k * a_j^k \tag{2.1}$$

$$Prob_{ij}^k(t) = \begin{cases} \dfrac{P_{ij}(t) * n_{ij}}{\sum_{l \in N_i^k(t)} P_{il}^l * n_{il}}, sej \in N_i^k \\ 0, senão \end{cases} \tag{2.2}$$

$$P_{ij}^{k} \begin{cases} \Delta P_{ij}^{k}(t) = \dfrac{Q}{L^{k}(t)}, se(i,j) \in T^{k}(t) \\ 0, senão \end{cases} \tag{2.3}$$

Considerando:

- T: conjunto de instâncias
- $n = |T|$: número de instâncias
- b_i(t): número de formigas na instância i no instante t
- $n_{ij} = 1/d_{ij}$: visibilidade de uma instância j para uma formiga na instância i
- P_{ij}: valor do feromônio no arco (i, j)
- C: matriz (b_i(t)xn) contendo a validação dos pontos de destino
- a_j^k: parâmetro binário aleatório

E, ao final de cada iteração do algoritmo, o valor do feromônio anteriormente depositado em todos os caminhos sofre evaporação, segundo a Equação 2.4.

$$P_{ij}(t+1) = (1-\rho) * P_{ij}(t) + \sum_{k=1}^{m} \Delta P_{ij}^{k}(t) \tag{2.4}$$

Com os fundamentos teóricos devidamente explorados, é possível detalhar o funcionamento do ACO por meio de um modelo simplificado de sua estrutura algorítmica. O Pseudocódigo 1 fornece uma visão geral das etapas principais, incluindo a geração e a ativação de formigas e a atualização das trilhas de feromônios. Essa representação é adaptada de Dorigo e Caro (1999) e ilustra de maneira sistemática os processos envolvidos na construção e na otimização de soluções no algoritmo. Em linhas gerais, o método proposto consiste na seguinte implementação:

1. **ACO_meta_heuristic()** (**Linha 1**): Inicializa a execução do algoritmo ACO e repete o processo enquanto o critério de parada não for satisfeito. O critério de parada pode estar relacionado ao número máximo de iterações, ao tempo de execução ou à convergência para uma solução ótima.

2. **generate_and_activate_ants()** (**Linha 3**): Responsável por gerar e ativar novas formigas que irão explorar o espaço de busca. Cada formiga representa uma solução candidata para o problema.

Algoritmo 1: ACO Meta-heuristic

```
1:  ACO_meta_heuristic() ;
2:  while TERMINATION CRITERION NOT SATISFIED do
        // schedule_activities:
3:      generate_and_activate_ants() ;
4:      evaporate_pheromone() ;
5:      daemon_actions() // optional
6:  end
7:  ;
8:  Function ants_generation_and_activity():
9:      while AVAILABLE RESOURCES do
10:         schedule_creation_of_new_ant() ;
11:         new_active_ant() ;
12:     end
13: ;
14: Function new_active_ant():
15:     initialize_ant() ;
16:     M ← update_ant_memory() ;
17:     while CURRENT_STATE ≠ TARGET_STATE do
18:         A ← read_local_ant_routing_table() ;
19:         P ← compute_transition_probabilities(A, M, Ω) ;
20:         next_state ← apply_decision_policy(P, Ω) ;
21:         move_to_next_state(next_state) ;
22:         if ONLINE_STEP_BY_STEP_PHEROMONE_UPDATE then
23:             deposit_pheromone_on_the_visited_arc() ;
24:             update_ant_routing_table() ;
25:         end
26:         M ← update_internal_state() ;
27:     end
28:     if ONLINE_DELAYED_PHEROMONE_UPDATE then
29:         for VISITED_ARC ∈ ψ do
30:             deposit_pheromone_on_the_visited_arc() ;
31:             update_ant_routing_table() ;
32:         end
33:     end
34:     die() ;
```

Fonte: Adaptado de (DORIGO; CARO, 1999)

3. **evaporate_pheromone()** (**Linha 4**): Responsável por realizar a evaporação do feromônio em todas as arestas da rede. Esse mecanismo reduz gradualmente a influência de soluções anteriores, impedindo que caminhos já explorados dominem a busca. Dessa forma, favorece-se a diversificação e aumenta-se a probabilidade de exploração de rotas alternativas, evitando que as formigas percorram repetidamente os mesmos trajetos.

4. **daemon_actions()** (**Linha 5**): Executa ações adicionais, como a aplicação de heurísticas de busca local para refinar as soluções. Esta etapa é opcional e pode ser utilizada para melhorar a qualidade das soluções encontradas.

5. **ants_generation_and_activity()** (**Linha 8**): Cria e ativa formigas enquanto houver recursos disponíveis (ex.: limite de tempo ou memória). Cada nova formiga será processada por meio da função `new_active_ant()`.

6. **new_active_ant()** (**Linha 11**): Modela o comportamento de uma formiga durante a exploração do ambiente. As principais etapas executadas são:

   (a) **initialize_ant()** (**Linha 15**): Inicializa os parâmetros da formiga, como sua posição inicial e memória interna.

   (b) **update_ant_memory()** (**Linha 16**): Atualiza a memória da formiga, registrando os estados já visitados.

   (c) **read_local_ant_routing_table()** (**Linha 18**): Consulta a tabela de roteamento local para identificar os próximos estados possíveis a serem explorados.

   (d) **compute_transition_probabilities(A, M, Ω)** (**Linha 19**): Calcula as probabilidades de transição para cada estado vizinho com base na quantidade de feromônio acumulado (A), na memória da formiga (M) e em parâmetros específicos (Ω).

   (e) **apply_decision_policy(P, Ω)** (**Linha 20**): Aplica a política de decisão para selecionar o próximo estado de acordo com as probabilidades calculadas.

   (f) **move_to_next_state(*next_state*)** (**Linha 21**): Move a formiga para o próximo estado escolhido.

   (g) **deposit_pheromone_on_the_visited_arc()** (**Linha 23**): Se a atualização de feromônio em tempo real estiver ativada, a formiga deposita feromônio no arco percorrido a cada passo, reforçando os caminhos visitados.

   (h) **update_ant_routing_table()** (**Linha 24**): Atualiza a tabela de roteamento com base no novo feromônio depositado.

   (i) **update_internal_state()** (**Linha 26**): Atualiza o estado interno da formiga, registrando os caminhos percorridos e a qualidade da solução atual.

7. **online_delayed_pheromone_update** (**Linha 28**): Se a atualização de feromônio for atrasada, ao final do percurso, a formiga deposita feromônio em cada arco visitado. Este processo reforça os melhores caminhos após a conclusão de uma trajetória completa.

8. **die()** (**Linha 34**): Finaliza o ciclo de vida da formiga após atingir o estado-alvo ou completar sua exploração.

2.2.3 ACO na Seleção de Instâncias

O algoritmo ACO para SI, desenvolvido por Miloud-Aouidate e Baba-Ali (2013), opera de forma estruturada e eficiente. Sua execução começa com o cálculo da matriz de distâncias euclidianas entre as instâncias, seguido pela determinação da visibilidade, que é inversamente proporcional às distâncias. A colônia é então inicializada com uma matriz de feromônios, que guia as formigas nas instâncias mais relevantes. Em cada iteração, as formigas percorrem caminhos probabilisticamente, considerando tanto a visibilidade quanto a concentração de feromônios, atualizando as trilhas com base na qualidade das soluções e evitando a convergência prematura. A solução final é determinada pelo classificador KNN, que avalia a acurácia de cada conjunto de instâncias selecionado, garantindo a escolha mais eficaz.

A seguir, apresenta-se uma explicação detalhada, passo a passo, do funcionamento do ACO para SI.

2.2.4 Passos do Algoritmo

1. **Inicialização do Algoritmo**

   (a) A base de dados é carregado a partir de um arquivo no formato *Comma-Separated Values* (CSV).

   (b) Separar os atributos da base de dados entre X(*Features*) e Y(*Target*)

2. **Calcular distâncias e visibilidades entre as instâncias**

   (a) `get_pairwise_distance(matrix: np.ndarray) -> np.ndarray`
   Calcula a matriz de distâncias entre todas as instâncias do conjunto de dados usando a distância euclidiana. A Função recebe uma matriz de dados X e retorna uma matriz de distâncias.

   (b) `get_visibility_rates_by_distances(distances: np.ndarray) -> np.ndarray`
   Calcula a visibilidade entre as instâncias, que é inversamente proporcional à distância entre elas.

3. **Inicialização da colônia de formigas e das trilhas de feromônio**

   (a) `create_colony(num_ants)`
   Cria uma matriz de formigas, onde cada linha representa uma formiga e cada célula indica se a formiga já visitou uma instância (inicialmente, todas as formigas não visitaram nada, portanto, os valores são -1).

   (b) `create_pheromone_trails(search_space: np.ndarray, initial_pheromone: float) -> np.ndarray`

Cria a matriz de trilhas de feromônio, onde todas as posições começam com o valor `initial_pheromone`, exceto a diagonal principal (já que não pode haver uma formiga indo da instância para ela mesma).

4. **Atualização das trilhas de feromônio**

   (a) `get_pheromone_deposit(ant_choices: List[Tuple[int, int]], distances: np.ndarray, deposit_factor: float) -> float`
   Calcula o valor de depósito de feromônio que uma formiga deve deixar, com base na distância total do caminho percorrido. O depósito é inversamente proporcional ao comprimento do caminho (quanto menor o caminho, maior o depósito).

5. **Determinar a ordem de seleção das instâncias pelas formigas**

   (a) `get_probabilities_paths_ordered(ant: np.array, visibility_rates: np.array, phe_trails)`
   Para uma formiga, esta função calcula as probabilidades de escolha das próximas instâncias a serem visitadas, com base na quantidade de feromônio e na visibilidade das distâncias. A função ordena as instâncias pela probabilidade de serem escolhidas, sendo priorizadas as instâncias mais atraentes.

6. **Encontrar a melhor solução**

   (a) `get_best_solution(ant_solutions: np.ndarray, X, Y)`
   Avalia todas as soluções das formigas (conjunto de instâncias selecionadas) usando um classificador KNN. Para cada solução, calcula a acurácia da classificação e retorna a solução com a maior acurácia.

7. **Finalização do Algoritmo**

   (a) As instâncias selecionadas pela melhor solução são salvas em um novo arquivo CSV.

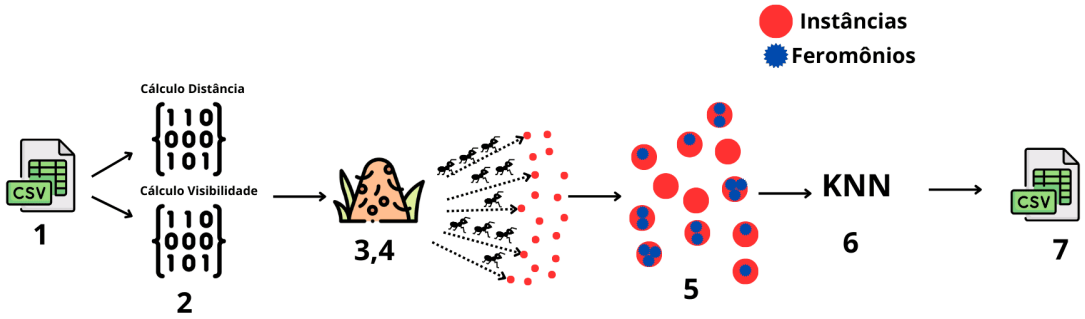
A Figura 2 ilustra o funcionamento do ACO na SI, destacando as etapas do processo. Cada número em cada etapa da figura corresponde a um dos passos previamente explicados.

2.3 Arquiteturas de Computação Paralela

2.3.1 Classificações de Flynn

No artigo de (FLYNN, 1966), os tipos de arquiteturas de computação são classificados de acordo com a Taxonomia de Flynn em quatro categorias: *Single Instruction*

Figura 2: Funcionamento do ACO para a Seleção de Instâncias



Fonte: Elaborado pelo autor

Stream-Single Data Stream (SISD), *Single Instruction, Multiple Data* (SIMD), *Multiple Instruction Stream-Single Data Stream* (MISD) e *Multiple Instruction Stream-Multiple Data Stream* (MIMD). Essas categorias são definidas com base na forma como os sistemas processam instruções e dados. O artigo teve uma continuação no artigo Flynn (1972) com um foco maior na implementação prática e na otimização de sistemas de alta performance, diferente do primeiro artigo que aborda uma classificação mais teórica.

Conforme descrito por Flynn (1972), as características de cada uma das quatro categorias são apresentadas a seguir:

- SISD: A arquitetura mais comum de organização de computadores, caracterizada por um único fluxo de instruções operando sobre um único fluxo de dados. Nesse modelo, as instruções são executadas de forma sequencial, uma após a outra, com cada instrução manipulando um único conjunto de dados por vez. Essa estrutura remonta aos primeiros computadores, sendo uma das suas principais limitações a dependência de dados e a latência provocada por instruções de desvio condicional (*branching*), as quais podem interromper o fluxo de execução. No entanto, técnicas como *pipeline* e concorrência permitem a melhoria do desempenho, ao possibilitar o processamento simultâneo de múltiplas instruções em estágios sobrepostos. As principais aplicações dessa arquitetura incluem processadores escalares e superescalares, que executam uma operação por vez por instrução.

- SIMD: A arquitetura é definida por um único fluxo de instruções operando sobre múltiplos fluxos de dados simultaneamente. Nesse modelo, a mesma operação é aplicada a diversos elementos de dados ao mesmo tempo, o que se revela especialmente vantajoso em aplicações que demandam processamento vetorial ou de *arrays*, como gráficos 3D e AM. Exemplos de sua aplicação incluem processadores vetoriais e GPU. A principal vantagem da arquitetura reside na capacidade de realizar operações em paralelo, o que resulta em um aumento substancial de eficiência para tarefas que podem ser decompostas em operações idênticas sobre grandes conjuntos

de dados. Contudo, seu desempenho pode ser limitado por desafios relacionados à comunicação entre os elementos de processamento e à dificuldade de lidar com desvios condicionais, nos quais diferentes elementos de dados seguem caminhos distintos.

- MISD: A arquitetura, embora menos comum em aplicações e predominantemente teórica, é caracterizada pela operação de múltiplos fluxos de instruções sobre um único fluxo de dados. Nesse modelo, diversas instruções são aplicadas ao mesmo conjunto de dados, com cada uma realizando operações distintas. Um exemplo de aplicação pode ser encontrado no conceito de sistemas de tolerância a falhas, onde múltiplos processadores executam diferentes instruções sobre os mesmos dados para garantir a precisão dos resultados. No entanto, na prática, sistemas MISD são raros, pois a maioria das aplicações não se beneficia de múltiplas operações simultâneas sobre os mesmos dados. A complexidade inerente à coordenação entre os fluxos de instruções torna essa arquitetura particularmente desafiadora de ser implementada de forma eficiente.

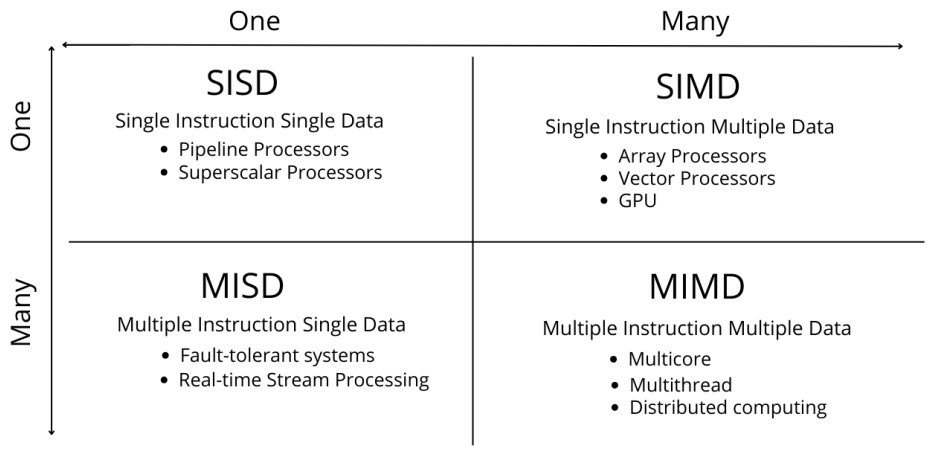
- MIMD: A arquitetura é definida por múltiplos fluxos de instruções operando simultaneamente sobre múltiplos fluxos de dados. Esse modelo é característico de sistemas multiprocessados e *multicore*, onde cada processador ou núcleo tem a capacidade de executar instruções independentes sobre diferentes conjuntos de dados. Os sistemas MIMD são altamente flexíveis, adequados para uma vasta gama de aplicações, desde computação científica até servidores de banco de dados e sistemas distribuídos. A principal vantagem do MIMD reside na capacidade de executar tarefas independentes em paralelo, o que pode resultar em ganhos substanciais de desempenho. No entanto, a eficiência desses sistemas está intimamente ligada à capacidade de gerenciar a comunicação e a sincronização entre os processadores, o que pode se tornar um gargalo à medida que o número de processadores aumenta.

A Figura 3 apresenta uma visão sintetizada das quatro classificações de Flynn, descritas anteriormente.

2.3.2 Paralelização pela CPU

As CPUs convencionais evoluíram incorporando várias formas de paralelismo, como *pipelines* massivamente superscalares, execução de dezenas de instruções fora da sequência e capacidades avançadas de SIMD, todas implementadas em múltiplos núcleos por *chip* de CPU (POLYCHRONIOU; RAGHAVAN; ROSS, 2015). Recentemente, os fabricantes de microprocessadores fornecem processadores que possuem múltiplos núcleos de 2, 4 ou mais, e PCs que utilizam tais processadores estão disponíveis a um custo razoável. Como

Figura 3: Classificações de Flynn



Fonte: Adaptado de (FLYNN, 1966, 1972)

a memória principal é compartilhada entre os núcleos, o processamento paralelo pode ser realizado de forma eficiente em processadores *multi-core* (TSUTSUI; FUJIMOTO, 2015; ALHENAWI et al., 2023). A popularidade dos processadores *multi-core* indica que a tendência futura dos computadores será *multi-core*, *Hyper-Threading* e *caches* grandes (FANG; WU, 2010a).

No entanto, é importante destacar que, em alguns processadores Intel® mais recentes, o suporte ao *Hyper-Threading* foi removido ou desativado devido a preocupações com segurança, como vulnerabilidades relacionadas a ataques de *Spectre e Meltdown*, conforme mencionado pela (INTEL, 2023). Adicionalmente, a Intel® implementou uma nova abordagem com sua arquitetura híbrida, que integra *Performance Cores (P-cores)* e *Efficient Cores (E-cores)*. Enquanto os *P-cores* são projetados para otimizar o desempenho e minimizar a latência em tarefas exigentes, os *E-cores* são voltados para a eficiência energética, sendo ideais para processos de segundo plano e multitarefas leves. Essa divisão estratégica proporciona um equilíbrio notável entre desempenho e consumo energético, adaptando-se de forma dinâmica às demandas de carga de trabalho (RUKMABHATLA et al., 2021).

Com o rápido avanço da arquitetura de computadores *multi-core* e da tecnologia de computação paralela, surge uma solução viável para problemas de otimização combinatória (DONGDONG et al., 2010). A vantagem da computação paralela é resolver problemas computacionais grandes e complexos rapidamente, e possuir uma boa escalabilidade (FANG; WU, 2010a). A disponibilidade de arquiteturas paralelas a baixo custo aumentou o interesse pela paralelização do algoritmo ACO (CHEN; SUN; WANG, 2008).

Implementações paralelas se tornaram populares na última década para melhorar a eficiência de algoritmos meta-heurísticos baseados em populações. Algoritmos paralelos não só se beneficiam do uso de vários elementos computacionais para reduzir o tempo

de execução, mas também introduzem um novo padrão de exploração que muitas vezes é útil para melhorar a qualidade dos resultados em relação às implementações sequenciais (LIU; HE, 2012a). Para maximizar a potência de computação, os programadores de *software* precisam mudar padrões de *design* e métodos de codificação. As tarefas de computação são divididas em várias partes e executadas por um número de processadores simultaneamente, em vez de usar o método tradicional linear, ou seja, cada processador lida com uma tarefa (FANG; WU, 2010a; ALHENAWI et al., 2023).

2.3.3 Vetorização pela CPU

A vetorização de CPU representa uma técnica essencial na otimização do desempenho de códigos em linguagens como C/C++ e Fortran. Ela transforma laços e operações em arranjos de dados, permitindo que o processador execute várias operações simultaneamente em elementos de dados contíguos (BORIN, 2024). A aplicação inteligente dessa abordagem é fundamental para alcançar um desempenho superior em sistemas computacionais. Os compiladores modernos são capazes de gerar códigos vetoriais, capitalizando as instruções SIMD (XAVIER et al., 2024). Processadores modernos possuem instruções e unidades funcionais vetoriais que são capazes de carregar e realizar operações com múltiplos dados consecutivos na memória (vetores) de uma só vez (BORIN, 2024). Os notáveis avanços no desempenho dos computadores foram, em grande parte, impulsionados por melhorias na arquitetura, resultado da convergência entre técnicas de vetorização e processamento paralelo (GENTZSCH, 1993).

2.3.4 Paralelização pela GPU

A GPU é um componente de processamento paralelo especializado em otimizar a execução de cálculos gráficos. Sua arquitetura é projetada para efetuar uma vasta gama de operações de ponto flutuante, fundamentais para a representação visual de ambientes tridimensionais. As GPUs são notáveis por sua altíssima capacidade de paralelismo e flexibilidade programável. O poder computacional paralelo de uma GPU transcende em várias ordens de grandeza o desempenho de uma CPU convencional (MCCLANAHAN, 2011; ZENG et al., 2023).

Geralmente, a GPU engloba um vasto conjunto de processadores CUDA (no caso das GPUs NVIDIA), cada um deles associado a um dos diversos *Streaming Multiprocessors* (SM). As *Threads* são organizadas em blocos, sendo que cada bloco é atribuído a um único SM. Núcleos pertencentes ao mesmo SM compartilham, entre outras coisas, o arquivo de registro, memória local, busca e decodificação de instruções e unidades de carga/armazenamento. A latência, frequentemente na ordem de centenas de ciclos, no acesso à memória global (principal), é um dos principais desafios para cálculos paralelos

e eficientes em GPUs (SKINDEROWICZ, 2016; ZENG et al., 2023).

O barramento de memória da GPU supera em largura o da CPU, apresentando uma banda de dados relativamente ampla (frequentemente na casa das centenas de GB/s), porém, muitas vezes, ainda insuficiente para alimentar todos os núcleos da GPU. Por essa razão, o paradigma de programação da GPU requer o emprego de um grande número de *Threads*. Enquanto um conjunto de *Threads* aguarda a conclusão da transferência de dados ou para a memória global, é possível realizar cálculos para *Threads* que já têm acesso aos dados transferidos. Em suma, um grande conjunto de elementos de processamento na GPU possibilita alcançar altas taxas de aceleração, desde que os cálculos sejam em sua maioria independentes e os dados sejam transferidos de maneira oportuna (SKINDEROWICZ, 2016).

As GPUs são programáveis através de diversas Interfaces de Programação, como CUDA, OpenCL, DirectX, OpenMP, OpenACC, OMP4Py, CuPy, PyCUDA e entre outros (STEIN, 2018). Neste trabalho, uma GPU NVIDIA é utilizada, o que implica que a paralelização da GPU é realizada pelo modelo de programação CUDA. Nesse contexto, a GPU atua como um co-processador *Single Instruction, Multiple Threads* (SIMT) de uma CPU convencional. Esse modelo baseia-se no conceito de *Kernels*, que são funções executadas em paralelo por um determinado número de núcleos CUDA. Essas *Threads* são agrupadas em blocos e distribuídas nos SMs da GPU para execução independente umas das outras (DELéVACQ et al., 2013).

2.3.5 Escalabilidade

A escalabilidade representa a capacidade de um sistema, rede ou processo de lidar com um aumento na demanda de trabalho, seja por meio da adição de novos recursos (escalabilidade horizontal) ou pela otimização do uso dos recursos já disponíveis (escalabilidade vertical) (CUNHA et al., 2018). De modo geral, um sistema é considerado escalável quando pode crescer ou se adaptar a cargas de trabalho maiores sem comprometer seu desempenho ou eficiência (CUNHA et al., 2018; VIANA, 2022).

Nos sistemas paralelos, a escalabilidade está relacionada à capacidade de melhorar o desempenho à medida que mais recursos computacionais, como núcleos de processamento, são adicionados. O conceito de tamanho do problema refere-se à quantidade de trabalho associada a uma entrada específica e é tratado como uma variável quantitativa (VIANA, 2022). Para que um programa paralelo seja considerado escalável, o aumento no número de núcleos deve ser acompanhado por um crescimento proporcional do tamanho do problema, de modo que a eficiência do sistema seja preservada (CUNHA et al., 2018; VIANA, 2022).

Existem dois tipos de escalabilidade: escalabilidade fraca e escalabilidade forte,

que serão conceituadas a seguir.

- Escalabilidade Fraca: Capacidade de um sistema paralelo de manter um tempo de execução constante ao aumentar simultaneamente o número de unidades de processamento e o tamanho do problema. Ou seja, à medida que mais recursos computacionais são adicionados, a quantidade de dados processados cresce na mesma proporção, garantindo que a carga de trabalho por núcleo permaneça equilibrada (CUNHA et al., 2018). Um programa paralelo é considerado fracamente escalável quando, ao aumentar o número de núcleos de processamento, o tamanho do problema cresce na mesma razão, mantendo a eficiência do sistema. Dessa forma, a escalabilidade fraca mede a habilidade do programa de aumentar seu desempenho sem perdas relevantes ao expandir os recursos computacionais e a carga de trabalho proporcionalmente (VIANA, 2022).
- Escalabilidade Forte: Capacidade de um sistema paralelo de reduzir o tempo de execução ao dividir um mesmo volume de trabalho entre um maior número de unidades de processamento, mantendo o tamanho do problema constante (CUNHA et al., 2018). A escalabilidade forte mede a habilidade de um programa paralelo de aumentar seu desempenho ao adicionar mais núcleos de processamento sem comprometer a eficiência, desde que o tamanho do problema permaneça fixo (VIANA, 2022).

Em resumo, a escalabilidade forte é mais adequada para cenários com pouca disponibilidade de dados, enquanto a escalabilidade fraca pode ser mais apropriada para situações em que a carga de trabalho pode ser ajustada conforme os recursos computacionais disponíveis.

3 TRABALHOS RELACIONADOS SOBRE OS TIPOS DE PARALELIZAÇÃO DO ALGORITMO ACO

A Revisão Sistemática da Literatura propicia a estruturação de trabalhos relevantes que respondem a questões pertinentes à área e auxiliam na pesquisa. Um dos artigos de referência nesse contexto é o de (KEELE et al., 2007), que apresenta diretrizes fundamentais para a condução desse tipo de estudo. Neste capítulo, esse artigo foi utilizado como referência para fundamentar a metodologia adotada, fornecendo uma estrutura clara e detalhada para cada etapa do processo, desde o planejamento até a disseminação dos resultados.

Com o intuito de examinar os principais estudos sobre a paralelização do ACO em CPU, GPU ou por meio de vetorização, adotou-se como referencial o levantamento conduzido por Pedemonte, Nesmachnow e Cancela (2011), um survey abrangente sobre paralelizações do ACO. Além disso, a estruturação desta seção seguiu as diretrizes estabelecidas por Keele et al. (2007), referência na condução de Revisões Sistemáticas da Literatura. Nesse contexto, os trabalhos de Silva et al. (2022) e Soares, Nobre e Freitas (2019) também foram considerados por aplicarem tais diretrizes em suas investigações.

3.1 Questões de Pesquisa

Este estudo tem como objetivo principal a identificação e análise das estratégias de paralelização do Algoritmo ACO. O foco da pesquisa recai sobre as variações decorrentes da paralelização em ambientes de CPU, GPU e Vetorização de CPU, com especial atenção às comparações entre essas abordagens. Além disso, são abordadas as principais bibliotecas em linguagens como Python e C/C++ para a implementação de paralelizações. Um aspecto de extrema importância consiste na avaliação do tempo de execução das estratégias, visando discernir a eficiência relativa de cada abordagem. Busca-se também mapear os segmentos do código mais propícios e eficazes para a aplicação de técnicas de paralelização, contribuindo, assim, para um entendimento prático mais abrangente dessas estratégias.

Por meio desses objetivos, almeja-se oferecer uma contribuição substancial para o entendimento prático e teórico das práticas de paralelização associadas ao ACO, delineando não apenas as tendências emergentes, mas, também, destacando as nuances de desempenho e eficácia nas diversas configurações de paralelização.

Três questões (QPs) foram elaboradas com o propósito de aprimorar a compreensão

dos questionamentos de pesquisa previamente esboçados:

- [$QP1$] Como as estratégias de paralelização do ACO se diferenciam em termos de desempenho ao serem implementadas em ambientes distintos, como CPU, GPU e Vetorização de CPU?
- [$QP2$] Quais são as comparações relevantes entre GPU, CPU e técnicas de Vetorização de CPU no contexto da paralelização do ACO e de que forma essas comparações contribuem para a compreensão mais profunda da eficiência e escalabilidade do algoritmo?
- [$QP3$] Quais bibliotecas destacam-se como principais ferramentas em linguagens de programação como Python e C/C++ para a implementação eficaz de paralelizações ou vetorizações associadas ao ACO?

3.2 Critérios de Seleção e Exclusão

Nesta seção são descritos os critérios de seleção e exclusão que orientaram a escolha dos artigos nesta revisão sistemática da literatura.

3.2.1 Critérios de Seleção

Os artigos selecionados foram identificados por meio de buscas realizadas nas bases ACM, IEEE, ScienceDirect, ResearchGate e Springer em novembro de 2023. No total, foram encontrados 300 artigos. A triagem foi conduzida com base nos títulos, resumos e palavras-chave, utilizando a seguinte string de busca em inglês:

([Title:Ant Colony Optimization] OR [Abstract:Ant Colony Optimization] OR [Keywords: Ant Colony Optimization]) ***AND*** *(([Title:Parallel] OR [Abstract:Parallel] OR [Keywords: Parallel])* ***OR*** *([Title: Vectorization] OR [Abstract:Vectorization] OR [Keywords: Vectorization])* ***OR*** *([Title: GPU] OR [Abstract:GPU] OR [Keywords: GPU])* ***OR*** *([Title: Cuda] OR [Abstract:Cuda] OR [Keywords: Cuda])* ***OR*** *([Title: Open MP] OR [Abstract:Open MP] OR [Keywords: Open MP])* ***OR*** *([Title: Shared Memory] OR [Abstract:Shared Memory] OR [Keywords: Shared Memory]))*

O Quadro 1 oferece uma visão abrangente das fases de busca inicial, filtro por *abstract* e leitura diagonal. A seleção foi restrita aos artigos que responderam a, pelo menos, uma das questões de pesquisa, passando previamente pelos critérios de exclusão descritos na subseção seguinte. O quadro inicia com o número total de artigos identificados, seguido pela quantidade retida em cada etapa do processo. No total, 49 artigos

foram selecionados com base em sua qualidade e relevância para os objetivos desta revisão sistemática da literatura.

Tabela 1: Quantidade de artigos selecionados em cada fase de seleção

Base de Dados	Busca Inicial	Filtro por Título	Filtro por *Abstract*	Leitura Diagonal
ACM	77	30	19	5
IEEE	102	45	39	21
Research Gate	53	12	9	6
Science Direct	83	25	15	10
Springer	40	12	10	6
Total	**300**	**124**	**92**	**49**

Fonte: Dados da Pesquisa

3.2.2 Critérios de Exclusão

Com o intuito de preservar o máximo de trabalhos pertinentes, a exclusão foi conduzida mediante a aplicação dos seguintes critérios:

3.2.2.1 ***Duplicação de Dados***

Em casos de trabalhos duplicados, provenientes do mesmo estudo, optou-se por considerá-los equivalentes, mantendo-se a versão mais recente. Quando um estudo é identificado em diferentes fontes, a opção pode recair sobre a exclusão de duplicatas, a fim de prevenir possíveis vieses na análise.

3.2.2.2 ***Idioma e Acesso***

Artigos redigidos em idiomas distintos do inglês e português foram excluídos da revisão. Além disso, estudos indisponíveis para acesso também foram excluídos da revisão.

3.2.2.3 ***Relevância Temática***

Optou-se por não incorporar artigos que abordassem algoritmos alternativos de seleção de instâncias paralelizados. Mesmo na ausência de publicações específicas sobre o ACO paralelizado empregando a seleção de instâncias, determinou-se priorizar, nesta revisão de literatura, exclusivamente, o ACO paralelizado, mesmo que seu uso se estenda a aplicações diversas, distintas da seleção de instâncias.

3.3 Discussão

Nesta seção são discutidas as questões de pesquisa apresentadas anteriormente na Seção 3.1 sobre questões de pesquisa, conforme os dados e informações contidos nos 49 artigos selecionados pela revisão.

QP1: Como as estratégias de paralelização do ACO se diferenciam em termos de desempenho ao serem implementadas em ambientes distintos, como CPU, GPU e vetorização de CPU?

Para elucidar esta questão, inicialmente, elaborou-se o Quadro 1 com o intuito de ilustrar a distribuição e os tipos de artigos que empregam paralelização por CPU, GPU ou vetorização de CPU. É relevante salientar que alguns artigos podem aparecer mais de uma vez na tabela, devido à utilização de múltiplos ambientes. Nota-se, portanto, que a maioria dos artigos opta por paralelizações via CPU ou GPU, enquanto apenas os estudos referenciados por Peake et al. (2018) e Lloyd e Amos (2016) fazem uso da vetorização de CPU.

Quadro 1 – Quantidade de Tipos de Ambiente utilizados nos artigos selecionados

Tipo de Ambiente	Quantidade de Artigos	Estudos Primários
CPU	23	(TSUTSUI; FUJIMOTO, 2015), (JIE; CAIYUN; ZHONG, 2008), (DONGDONG et al., 2010), (FANG; WU, 2010b), (ALHENAWI et al., 2023), (LIU; HE, 2012b), (ARNAUTOVIć et al., 2013), (LI et al., 2010), (PENG et al., 2006), (DOUGLAS; LEVI, 2002), (DELISLE et al., 2001), (CHEN; SUN; WANG, 2008), (MANFRIN; BIRATTARI; STüTZLE, 2006), (YANG; FANG; DUAN, 2016), (DZALBS; KALGANOVA, 2020), (ZHOU et al., 2018), (FU; LEI; ZHOU, 2010), (ELSAID et al., 2018), (BAYDOGMUS, 2022), (TURNER; WHITE, 2013), (WANG, 2023), (ELSAID et al., 2017), (MARKVICA; SCHAUER; RAIDL, 2015)
GPU	26	(SINNOTT-ARMSTRONG; GREENE; MOORE, 2010), (SATO et al., 2014), (ZENG et al., 2023), (CEKMEZ; OZSIGINAN; SAHINGOZ, 2014), (DAWSON; STEWART, 2014), (YOUNESS et al., 2015), (GAO et al., 2016), (JIENING; JIANKANG; CHUNFENG, 2009), (DELISLE; KRAJECKI; GRAVEL, 2009), (ZHU; CURRY, 2009), (ARNAUTOVIć et al., 2013), (MENEZES et al., 2019), (WEI; WU; WU, 2013), (GUERRERO et al., 2014), (ZHOU; HE; QIU, 2017), (DZALBS; KALGANOVA, 2020), (ZHOU et al., 2022), (PAPENHAUSEN; MUELLER, 2018), (CECILIA et al., 2013), (CECILIA et al., 2018), (DELEVACQ et al., 2013), (SKINDEROWICZ, 2016), (FU; LEI; ZHOU, 2010), (TUFTELAND; ØDESNELTVEDT; GOODWIN, 2016), (DAWSON; STEWART, 2013), (MARKVICA; SCHAUER; RAIDL, 2015)
Vetorização CPU	2	(PEAKE et al., 2018), (LLOYD; AMOS, 2016)

Fonte: Dados da Pesquisa

Para separar qual o tipo de estratégia utilizada de modo geral, em cada um dos ambientes de implementação, foram divididos em subseções e alguns artigos não explicam como foi o processo de paralelização. Por exemplo, Sinnott-Armstrong, Greene e Moore

(2010) apresentam apenas qual a linguagem utilizada, sem detalhar o processo de paralelização utilizado. Logo, os mesmos não serão inseridos nas seções abaixo e vale ressaltar que alguns artigos utilizam mais de um tipo de implementação, sendo possível aparecer em mais de uma seção.

3.3.0.1 CPU

- Os estudos conduzidos por Tsutsui e Fujimoto (2015), Jie, CaiYun e Zhong (2008), Alhenawi et al. (2023), Liu e He (2012b), Douglas e Levi (2002), Manfrin, Birattari e Stützle (2006), Baydogmus (2022) e Markvica, Schauer e Raidl (2015) adotaram uma abordagem de paralelização, na qual cada *Thread* é equiparada a uma formiga, permitindo a paralelização da trilha de feromônios e dos cálculos para a determinação das melhores soluções. Essa estratégia resulta numa redução do tempo de execução, uma vez que múltiplas formigas podem simultaneamente buscar e avaliar as melhores soluções. Embora diversos estudos mencionem que o aumento no número de threads resulta em ganhos de desempenho((JIE; CAIYUN; ZHONG, 2008) (DOUGLAS; LEVI, 2002) (DELISLE; KRAJECKI; GRAVEL, 2009) (CHEN; SUN; WANG, 2008), não foi observado ou explicitamente discutido em muitos deles se há um limite superior para o número de *threads* após o qual os ganhos se tornam insignificantes. No entanto, é sabido que a eficiência da paralelização tende a decair à medida que os custos de sincronização e comunicação aumentam, especialmente em arquiteturas com memória compartilhada.

- Os estudos conduzidos por Dongdong et al. (2010), Delisle, Krajecki e Gravel (2009), Fang e Wu (2010b), Arnautović et al. (2013), Delisle et al. (2001), Chen, Sun e Wang (2008), Manfrin, Birattari e Stützle (2006), Yang, Fang e Duan (2016), e Zhou et al. (2018) adotaram uma estratégia semelhante de paralelização, na qual, cada *Threads* é tratada como uma formiga. Além disso, esses estudos empregaram memória compartilhada para facilitar a comunicação entre as *Threads*, visando a obtenção mais rápida e eficiente das melhores soluções. Observa-se que, à medida que o número de *Thread* é aumentado, há um correspondente aumento no ganho, conforme evidenciado em trabalhos como (DELISLE; KRAJECKI; GRAVEL, 2009) e Chen, Sun e Wang (2008).

- O estudo conduzido por Li et al. (2010) adota uma abordagem que paraleliza cada formiga, utilizando cada *Thread* disponível no processador, conforme evidenciado nos exemplos anteriores. No entanto, os autores foram além ao paralelizar praticamente todo o código, desde a atualização dos feromônios até a sincronização das formigas na busca pelas melhores soluções. Vale ressaltar que essas implementações

não apenas preservaram o tempo de execução, mas também mantiveram a qualidade das soluções obtidas, resultando em um ganho médio de 1.5.

3.3.0.2 GPU

- Os estudos de Sinnott-Armstrong, Greene e Moore (2010), Zeng et al. (2023), Cekmez, Ozsiginan e Sahingoz (2014), Youness et al. (2015), Gao et al. (2016), Arnautović et al. (2013), Wei, Wu e Wu (2013), Guerrero et al. (2014), Zhou, He e Qiu (2017), Zhou et al. (2022), Papenhausen e Mueller (2018), Cecilia et al. (2013), Delevacq et al. (2013), Skinderowicz (2016), Zhou et al. (2018), Fu, Lei e Zhou (2010), Tufteland, Ødesneltvedt e Goodwin (2016), Markvica, Schauer e Raidl (2015) apresentam uma implementação em que cada *Thread* da GPU é uma formiga, integrando os cálculos das matrizes de feromônio e distâncias entre as formigas, diretamente, na GPU. Além disso, são executadas operações de probabilidade de movimento e seleção das melhores soluções, todas encapsuladas nas chamadas de *Kernel*. Uma prática adotada por todos os artigos revisados é a utilização da memória compartilhada para armazenar dados entre diferentes *Threads* ou Blocos, visando diminuir potenciais gargalos de desempenho em GPUs. Esta abordagem, aliada às técnicas de paralelização discutidas neste contexto, resultou em ganhos de desempenho em comparação com as versões sequenciais de todas as análises realizadas.

- Os estudos conduzidos por Dawson e Stewart (2014), Dzalbs e Kalganova (2020), e Dawson e Stewart (2013) seguem a mesma lógica de implementação previamente explicada. No entanto, o diferencial reside na comparação de desempenho entre implementações com e sem o uso de memória compartilhada. Em particular, Dawson e Stewart (2014) e Dawson e Stewart (2013) constataram que, surpreendentemente, a implementação sem o uso de memória compartilhada alcançou um ganho superior. Em contrapartida, Dzalbs e Kalganova (2020) apresentaram resultados mais favoráveis com o compartilhamento de memória.

- Zhu e Curry (2009) reproduziram as implementações anteriores, observando um aumento proporcional no ganho à medida que a quantidade de *Threads* é aumentada, conforme evidenciado no exemplo apresentado no artigo.

3.3.0.3 Vetorização CPU

- Peake et al. (2018) e Lloyd e Amos (2016) implementaram uma vetorização via CPU para acelerar o processo de busca do melhor "próximo vizinho" no algoritmo ACO. Isso se deve ao fato de que o processo sequencial dessa etapa consome muito

tempo, especialmente na medida que a quantidade de instâncias aumenta. Ambos os estudos alcançaram expressivos ganhos. No entanto, enquanto Lloyd e Amos (2016) obtiveram um ganho de desempenho médio de 10x na paralelização, Peake et al. (2018) observaram que o ganho aumenta conforme a quantidade que instâncias crescem.

As estratégias de paralelização do Algoritmo ACO exibem variações expressivas, em termos de desempenho, quando implementadas em diferentes ambientes como CPU, GPU e vetorização de CPU. Ao analisar os estudos mencionados, observa-se que a paralelização via CPU e GPU é comumente adotada, resultando em reduções substanciais no tempo de execução. Já nas implementações que utilizam CPU, como nos estudos de Peake et al. (2018) e Lloyd e Amos (2016), a vetorização proporciona ganhos expressivos, especialmente em cenários com aumento na quantidade de instâncias do problema.

QP2: Quais são as comparações significativas entre GPU, CPU e técnicas de vetorização no contexto da paralelização do ACO e de que forma essas comparações contribuem para a compreensão mais profunda da eficiência e escalabilidade do algoritmo?

A maioria dos artigos se limita a implementar uma paralelização do ACO em um dos ambientes mencionados anteriormente e comparar os resultados com a versão sequencial do algoritmo. Portanto, esses casos não serão abordados neste contexto. No entanto, há outras abordagens que serão discutidas a seguir.

Apenas um estudo realizou comparações de desempenho entre diferentes abordagens de paralelização utilizando bibliotecas de CPU:

- Dongdong et al. (2010) realizaram uma análise comparativa da eficiência da paralelização entre duas bibliotecas: *Threading Building Blocks* (TBB) e *Open Multi-Processing* (OpenMP). Observou-se que o OpenMP alcançou um ganho superior em comparação com o TBB. Além disso, à medida que o número de *Thread* aumenta, o ganho também aumenta e o tempo de execução diminui no caso do OpenMP. Por outro lado, no TBB, à medida que o número de *Threads* aumenta, o ganho diminui; entretanto, com um menor número de *Threads*, o ganho é maior. Independentemente da quantidade de *Threads* utilizadas, a biblioteca OpenMP sempre superou o TBB em termos de ganho.

Apenas um estudo aborda a comparação de desempenho do algoritmo em diferentes bibliotecas de CPU, implementadas em linguagens diferentes e todas executadas na mesma máquina:

- Em uma análise conduzida por Alhenawi et al. (2023), a paralelização do ACO foi comparada entre a biblioteca *Message Passing Interface* (MPI) em C e o *Apache Spark* em Python, ambos implementados na mesma máquina. Os resultados indicaram que ambas as implementações alcançaram o desempenho máximo ao utilizar 4 núcleos. No entanto, a implementação com a biblioteca MPI em C obteve, em média, um ganho superior a 25% em comparação com a implementação em Python utilizando o *Apache Spark*, em todos os cenários avaliados.

Apenas um estudo realizou uma comparação de desempenho entre a paralelização em diferentes GPUs:

- Esse estudo, conduzido por Sato et al. (2014), realizou uma comparação de desempenho entre as GPUs Tesla K20c e Xeon Phi 5110P. O Tesla K20c alcançou um ganho de 33.5 em comparação com o algoritmo sequencial, enquanto o Xeon Phi 5110P atingiu um ganho de 5.5. Os autores explicam essa grande diferença de ganho entre as GPUs pelo fato do Tesla K20c possuir mais núcleos do que o Xeon Phi 5110P (2496 núcleos versus 60 núcleos).

Alguns estudos empregaram a mesma linguagem de programação para comparar o desempenho da paralelização entre CPU e GPU:

- Em uma pesquisa conduzida por Arnautović et al. (2013), foi comparada a implementação da paralelização do ACO utilizando a biblioteca OpenMP na CPU e a biblioteca CUDA na GPU. Ambas as abordagens de paralelização resultaram em melhorias de desempenho em comparação com a versão sequencial do algoritmo. No entanto, a versão para GPU alcançou os melhores ganhos em todos os casos analisados. Segundo os autores, a implementação para GPU é aproximadamente 50 vezes mais rápida do que a versão sequencial.

- Em seu estudo, Dzalbs e Kalganova (2020) compararam a paralelização realizada pela CPU e pelas GPUs Xeon Phi e Nvidia GTX 1070. Para a CPU, a paralelização foi conduzida com o OpenMP, enquanto para o Xeon Phi, utilizou-se a biblioteca Intel MKL e para a GPU, empregou-se o CUDA. Os resultados obtidos em cada ambiente de comparação com a execução sequencial revelaram um ganho de 25.4 na CPU, de 148 no Xeon Phi, enquanto na GPU a implementação não alcançou uma aceleração relevante. Isso se deve ao acesso frequente aos metadados, que é relativamente lento nas GPUs.

- Em seu estudo, Tufteland, Ødesneltvedt e Goodwin (2016) realizaram uma comparação de desempenho entre o algoritmo sequencial e as paralelizações conduzidas

pela CPU e GPU utilizando bibliotecas em Python. Para a CPU, a paralelização foi realizada com a biblioteca *Just-In-Time* (JIT), enquanto, que, para a GPU foi utilizada a biblioteca CUDA. Os resultados revelaram que a implementação na GPU foi, em média, 6.6 vezes mais rápida do que na CPU e 193.8 vezes mais rápida do que o código sequencial. Por outro lado, a CPU revelou-se 29.3 vezes mais rápida do que a execução sequencial.

- Em sua pesquisa, Markvica, Schauer e Raidl (2015) compararam implementações em GPU e CPU em Python, utilizando a biblioteca OpenCL em ambos os *hardwares*. Surpreendentemente, os resultados indicaram que a versão de CPU superou a versão de GPU, em termos de desempenho. Os autores explicaram essa discrepância ao apontar para a latência no acesso à memória, pois o algoritmo requer escrita e leitura frequentes ao longo de sua execução, enquanto a memória compartilhada da GPU é limitada. Como conclusão, os autores destacaram que a GPU pode representar um gargalo para algoritmos que não exigem um alto grau de tarefas paralelizadas.

As comparações entre GPU, CPU e técnicas de vetorização no contexto da paralelização do ACO oferecem resultados sobre a eficiência e escalabilidade do algoritmo em diferentes ambientes computacionais. Os estudos destacam que, embora a GPU geralmente apresente uma vantagem em termos de ganho em relação à CPU e à execução sequencial, esse desempenho pode ser mitigado por questões como acesso lento à memória e latência. Por exemplo, enquanto implementações de GPU com bibliotecas, como CUDA, podem alcançar ganhos relevantes em comparação com as versões sequenciais e até mesmo com implementações de CPU, a eficácia dessa abordagem pode ser comprometida por demandas intensivas de acesso à memória. Além disso, estudos que comparam diferentes bibliotecas de CPU, como OpenMP e TBB, proporcionam a escolha da biblioteca na escalabilidade e eficiência do algoritmo, destacando como variações no número de *threads* podem afetar o desempenho. Da mesma forma, comparações entre diferentes tipos de GPU, como Tesla K20c e Xeon Phi 5110P, ilustram como diferenças na arquitetura podem impactar o ganho alcançado. Em suma, essas comparações não apenas evidenciam os benefícios da paralelização do ACO em termos de aceleração computacional, mas também destacam considerações cruciais a serem feitas ao selecionar o ambiente de computação mais adequado para uma aplicação específica, levando em conta fatores como acesso à memória, arquitetura do hardware e características do algoritmo.

QP3: Quais bibliotecas destacam-se como principais ferramentas em linguagens de programação como Python e C para a implementação eficaz de paralelizações ou vetorizações associadas ao ACO?

A implementação eficaz de paralelizações associadas ao ACO em linguagens de pro-

gramação são essenciais para otimizar o desempenho em cenários de otimização temporal. Este parágrafo introduz a relevância da seleção de bibliotecas na implementação de paralelizações associadas ao ACO, destacando a importância dessa escolha para o desempenho e a eficácia do algoritmo em contextos específicos.

No Quadro 2, é possível observar a quantidade e as linguagens de programação utilizadas em cada artigo. Destaca-se que alguns artigos podem aparecer em mais de uma entrada na tabela, uma vez que incorporam diversas linguagens de programação para a implementação de técnicas de paralelização.

Quadro 2 – Linguagens utilizadas nos artigos selecionados

Linguagem	**Quantidade de Artigos**	**Estudos Primários**
C/C++	40	(TSUTSUI; FUJIMOTO, 2015), (SINNOTT-ARMSTRONG; GREENE; MOORE, 2010), (SATO et al., 2014), (PEAKE et al., 2018), (LLOYD; AMOS, 2016), (JIE; CAIYUN; ZHONG, 2008), (ZENG et al., 2023), (CEKMEZ; OZSIGINAN; SAHINGOZ, 2014), (DAWSON; STEWART, 2014), (YOUNESS et al., 2015), (DONGDONG et al., 2010), (GAO et al., 2016), (JIENING; JIANKANG; CHUNFENG, 2009), (DELISLE; KRAJECKI; GRAVEL, 2009), (FANG; WU, 2010b), (ZHU; CURRY, 2009), (ALHENAWI et al., 2023), (LIU; HE, 2012b), (ARNAUTOVIć et al., 2013), (MENEZES et al., 2019), (LI et al., 2010), (WEI; WU; WU, 2013), (PENG et al., 2006), (DOUGLAS; LEVI, 2002), (DELISLE et al., 2001), (CHEN; SUN; WANG, 2008), (MANFRIN; BIRATTARI; STüTZLE, 2006), (GUERRERO et al., 2014), (ZHOU; HE; QIU, 2017), (YANG; FANG; DUAN, 2016), (DZALBS; KALGANOVA, 2020), (ZHOU et al., 2022), (PAPENHAUSEN; MUELLER, 2018), (CECILIA et al., 2013), (CECILIA et al., 2018), (DELEVACQ et al., 2013), (TSUTSUI; FUJIMOTO, 2010), (SKINDEROWICZ, 2016), (ZHOU et al., 2018)
Java	1	(SINNOTT-ARMSTRONG; GREENE; MOORE, 2010)
MATLAB	2	(GAO et al., 2016), (FU; LEI; ZHOU, 2010)
Python	9	(SINNOTT-ARMSTRONG; GREENE; MOORE, 2010), (ALHENAWI et al., 2023), (ELSAID et al., 2018), (BAYDOGMUS, 2022), (TURNER; WHITE, 2013), (WANG, 2023), (TUFTELAND; ØDESNELTVEDT; GOODWIN, 2016), (ELSAID et al., 2017), (MARKVICA; SCHAUER; RAIDL, 2015)
Não Especificada	1	(DAWSON; STEWART, 2013)

Fonte: Dados da Pesquisa

Conforme a análise apresentada no Quadro 2, 75% dos artigos escolheram estratégias de paralelização ou vetorização utilizando a linguagem C/C++. Tal resultado era previsto, dado que C/C++ é amplamente reconhecida como uma linguagem de baixo nível ideal para tais abordagens. Seguindo essa tendência, a linguagem Python surge como a segunda mais prevalente, sendo adotada em 18% dos artigos selecionados.

No Quadro 3, é possível verificar as bibliotecas de paralelização de CPU ou de GPU utilizadas em cada artigo. Caso a biblioteca não seja especificada no artigo, este será listado na linha “Não especificada”. Ressalta-se que vetorizações não foram incluídas

nesta tabela, uma vez que nenhum artigo, independentemente da linguagem utilizada, empregou uma biblioteca específica para vetorização.

Quadro 3 – Bibliotecas e Linguagens utilizadas nos artigos selecionados

Biblioteca	Linguagem	Estudos Primários
Apache Spark	Python	(ALHENAWI et al., 2023)
Cuda	C/C++	(SATO et al., 2014), (ZENG et al., 2023), (CEKMEZ; OZSIGINAN; SAHINGOZ, 2014), (DAWSON; STEWART, 2014), (YOUNESS et al., 2015), (GAO et al., 2016), (ZHU; CURRY, 2009), (ARNAUTOVIć et al., 2013), (MENEZES et al., 2019), (WEI; WU; WU, 2013), (GUERRERO et al., 2014), (DZALBS; KALGANOVA, 2020), (PAPENHAUSEN; MUELLER, 2018), (CECILIA et al., 2013), (CECILIA et al., 2018), (DELEVACQ et al., 2013), (SKINDEROWICZ, 2016)
Cuda	MATLAB	(GAO et al., 2016), (FU; LEI; ZHOU, 2010)
Cuda	Python	(SINNOTT-ARMSTRONG; GREENE; MOORE, 2010), (TUFTELAND; ØDESNELTVEDT; GOODWIN, 2016)
Cpython	Python	(TUFTELAND; ØDESNELTVEDT; GOODWIN, 2016)
Intel MKL	C/C++	(DZALBS; KALGANOVA, 2020)
Jit	Python	(TUFTELAND; ØDESNELTVEDT; GOODWIN, 2016)
MPI	C/C++	(JIE; CAIYUN; ZHONG, 2008), (ALHENAWI et al., 2023), (DOUGLAS; LEVI, 2002), (CHEN; SUN; WANG, 2008), (MANFRIN; BIRATTARI; STüTZLE, 2006), (YANG; FANG; DUAN, 2016)
MPI	Python	(ELSAID et al., 2018), (ELSAID et al., 2017)
OpenMP	C/C++	(TSUTSUI; FUJIMOTO, 2015), (DONGDONG et al., 2010), (DELISLE; KRAJECKI; GRAVEL, 2009), (FANG; WU, 2010b), (LIU; HE, 2012b), (ARNAUTOVIć et al., 2013), (PENG et al., 2006), (DELISLE et al., 2001), (DZALBS; KALGANOVA, 2020)
OpenCL	C/C++	(MARKVICA; SCHAUER; RAIDL, 2015)
OpenCL	Python	(GUERRERO et al., 2014), (ZHOU et al., 2018)
OpenGL	C/C++	(JIENING; JIANKANG; CHUNFENG, 2009)
TBB	C/C++	(DONGDONG et al., 2010), (LI et al., 2010)
Não Especificada	Python	(BAYDOGMUS, 2022), (TURNER; WHITE, 2013), (WANG, 2023)
Não Especificada	C/C++	(ZHOU; HE; QIU, 2017)

Fonte: Dados da Pesquisa

No Quadro 3, observa-se que a maioria dos artigos optou por utilizar as bibliotecas CUDA ou OpenMP. Ao somar todos os artigos, independentemente da linguagem utilizada, constata-se que o CUDA foi empregado em 44% dos casos, enquanto o OpenMP, uma biblioteca exclusiva para C/C++, foi a segunda mais utilizada e presente em 22% dos artigos.

3.4 Conclusão

Essa análise detalhada dos estudos sobre a paralelização do ACO proporciona uma compreensão abrangente das estratégias, desafios e resultados obtidos nesse contexto. A revisão sistemática da literatura revela uma predominância de implementações em C/C++, seguidas por Python, destacando a preferência por linguagens de programação que oferecem maior controle sobre o *hardware* e permitem abordagens de baixo nível.

A comparação entre diferentes ambientes de implementação, como CPU, GPU e técnicas de vetorização de CPU, evidencia que a escolha do ambiente impacta diretamente o desempenho do algoritmo. A GPU geralmente se destaca, proporcionando ganhos expressivos em comparação com CPU e execução sequencial. No entanto, considerações como acesso à memória e características específicas do algoritmo podem modular esses resultados, como ilustrado por estudos que apontam para possíveis gargalos em determinadas situações.

As comparações entre bibliotecas, especialmente em ambientes de CPU, destacam a importância da escolha certa para otimizar o desempenho. OpenMP e CUDA emergem como ferramentas frequentemente utilizadas, com resultados que variam de acordo com a implementação específica e características do *hardware*. Além disso, estudos que abordam a paralelização em linguagens como Python apontam para desafios e vantagens específicas, evidenciando a diversidade de abordagens e ambientes de implementação.

Ao contemplar as questões de pesquisa, essa revisão sistemática contribui substancialmente para o entendimento prático e teórico das práticas de paralelização associadas ao ACO. As conclusões destacam não apenas as tendências emergentes, mas também nuances de desempenho e eficácia em diversas configurações. A escolha adequada da linguagem de programação, ambiente de implementação e bibliotecas desempenham um papel crucial na obtenção de resultados otimizados. Portanto, pesquisadores e desenvolvedores podem empregar esta seção para orientar escolhas fundamentadas ao aplicar estratégias de paralelização ao ACO em diferentes contextos.

4 ESTRATÉGIAS DE PARALELIZAÇÃO E VETORIZAÇÃO DO ACO

Neste capítulo, são apresentados os modelos de paralelização e vetorização do algoritmo ACO. As seções seguintes detalham a implementação das versões paralelizadas para CPU, GPU e vetorização de CPU no ACO. Cabe destacar que todos os códigos, tanto a versão sequencial quanto as variantes paralelizadas, foram desenvolvidos em Python, versão 3.10.

4.1 Paralelização pela CPU

A paralelização da CPU é realizada por meio da utilização da biblioteca Multiprocessing*, que cria processos em cada núcleo de uma CPU com múltiplos núcleos, permitindo a execução simultânea das formigas na busca pelas melhores instâncias.

Ela ocorre em dois momentos principais: durante a inicialização da colônia de formigas e durante a busca de soluções. A função *initialize_diagonal_parallel*, responsável por marcar a diagonal principal da matriz de colônia, é executada em paralelo utilizando o método "Pool.map", o que distribui o trabalho entre os processos disponíveis. Da mesma forma, a busca de soluções pelas formigas, realizada pela função *ant_search_solution*, é paralelizada ao ser chamada com diferentes argumentos para cada formiga, também por meio do método "Pool.map".

A utilização da paralelização nesses momentos permitiu um melhor aproveitamento dos recursos computacionais. Durante a inicialização da colônia, a divisão do trabalho garantiu que cada nó da matriz fosse processado de forma independente e simultânea, agilizando a configuração inicial. Na busca de soluções, a paralelização possibilitou que múltiplas formigas explorassem o espaço de soluções ao mesmo tempo, promovendo uma exploração mais ampla e eficiente do problema.

O processo de paralelização pode ser visualizado no pseudocódigo apresentado no Algoritmo 2.

De maneira geral, o método proposto para paralelização na CPU é implementado da seguinte forma:

1. **run_colony(X, Y, initial_pheromone, evaporation_rate, Q)** (Linha 1): Esta função principal gerencia a execução paralela da colônia de formigas utilizando

*Documentação Oficial pode ser encontrada em: https://docs.python.org/3/library/multiprocessing.html

Algoritmo 2: ACO Multiprocessing

```
1:  Function RunColony(X, Y, INITIAL_PHEROMONE, EVAPORATION_RATE, Q):
2:      for EACH ANT i IN COLONY do
3:          submit ant task to multiprocessing executor;
4:      end
5:      best_solution ← getBestSolution(the_colony, X, Y);
6:      instances_selected ← getNonZeroIndices(best_solution);
7:      return instances_selected;
8:  ;
9:  Function AntSearchSolution(I, ANT, Q, EVAPORATION_RATE):
10:     while −1 IN ANT do
11:         last_choice ← antChoices[i][-1];
12:         ant_pos ← last_choice[1];
13:         choices ← getProbabilitiesPathsOrdered();
14:         for EACH choice IN choices do
15:             next_instance, probability, ajk ← choice;
16:             final_probability ← probability × ajk;
17:             if final_probability ≠ 0 then
18:                 update ant choices and colony state;
19:                 compute pheromone deposit;
20:                 update pheromone trails;
21:             end
22:             else
23:                 mark instance as unavailable;
24:             end
25:         end
26:     end
```

Fonte: Dados da Pesquisa

multiprocessamento. Os parâmetros de entrada incluem:

- X: **ndarray** — Representa as instâncias do conjunto de dados.
- Y: **ndarray** — Representa os rótulos do conjunto de dados.
- *initial_pheromone*: **float** — Valor inicial do feromônio em cada caminho.
- *evaporation_rate*: **float** — Taxa de evaporação do feromônio, responsável por reduzir a intensidade do feromônio ao longo do tempo.
- Q: **float** — Constante que regula a quantidade de feromônio depositada pelas formigas.

(a) **parallel for** (Linha 2): Executa a exploração das soluções em paralelo para cada formiga i da colônia. Essa abordagem paraleliza a busca de soluções, acelerando o processo.

(b) **submit ant task to multiprocessing executor** (Linha 3): Cada formiga é submetida como uma tarefa assíncrona a um executor de multiprocessamento, permitindo a exploração simultânea do espaço de busca.

(c) **end parallel for** (Linha 4): Marca o término da execução paralela, aguardando a conclusão de todas as tarefas.

(d) *best_solution* ← **getBestSolution(the_colony, X, Y)** (Linha 5): Após a execução da colônia, a melhor solução encontrada é identificada com base nos critérios definidos (ex.: qualidade da seleção de instâncias).

(e) *instances_selected* ← **getNonZeroIndices(best_solution)** (Linha 6): Extrai os índices das instâncias selecionadas, que correspondem aos elementos não nulos da melhor solução.

(f) **return** *instances_selected* (Linha 7): Retorna a lista de índices das instâncias selecionadas como resultado do algoritmo.

2. **ant_search_solution(i, ant, Q, evaporation_rate)** (Linha 9): Esta função define a lógica de busca de solução para cada formiga. Os parâmetros incluem:

   i: **int** — Identificador da formiga.
   ant: **List[int]** — Representa a solução parcial da formiga em construção.
   Q: **float** — Constante que regula o depósito de feromônio.
   evaporation_rate: **float** — Taxa de evaporação do feromônio.

   (a) **while** -1 **in ant** (Linha 10): Executa a busca enquanto houver posições não preenchidas na solução da formiga.

   (b) *last_choice* ← **antChoices[i][-1]** (Linha 11): Obtém a última escolha feita pela formiga *i*.

   (c) *ant_pos* ← **last_choice[1]** (Linha 12): Armazena a posição atual da formiga.

   (d) *choices* ← **getProbabilitiesPathsOrdered()** (Linha 13): Calcula e retorna as escolhas possíveis, ordenadas pelas probabilidades de transição.

   (e) **for each** *choice* **in** *choices* (Linha 14): Itera sobre todas as escolhas disponíveis para a próxima etapa.

      i. $next\_instance, probability, ajk \leftarrow choice$ (Linha 15): Decompõe a escolha em três componentes:

         - *next_instance* — Próxima instância a ser visitada.
         - *probability* — Probabilidade associada à escolha.
         - ajk — Quantidade de feromônio associada ao caminho.

ii. $final\_probability \leftarrow probability \times ajk$ (Linha 16): Calcula a probabilidade final combinando a probabilidade de transição e a influência do feromônio.

iii. **if** $final\_probability \neq 0$ (Linha 17): Se a probabilidade final for diferente de zero:

A. **update** ant choices and colony state (Linha 18): Atualiza as escolhas da formiga e o estado da colônia.

B. **compute** pheromone deposit (Linha 19): Calcula a quantidade de feromônio a ser depositada com base na solução encontrada.

C. **update** pheromone trails (Linha 20): Atualiza as trilhas de feromônio, reforçando os caminhos percorridos.

iv. **else** (Linha 22): Caso contrário:

A. **mark** instance as unavailable (Linha 23): Marca a instância como indisponível, impedindo sua seleção em futuras iterações.

(f) **end for** (Linha 25): Finaliza o laço de iteração pelas escolhas.

3. **end while** (Linha 26): Termina a execução da busca quando todas as posições da solução forem preenchidas.

A Figura 4 ilustra a paralelização do ACO, destacando a inicialização da colônia e a seleção das melhores soluções de forma paralela. Esse processo transforma uma tarefa anteriormente sequencial em múltiplas execuções distribuídas entre diversos núcleos, conforme representado a seguir.

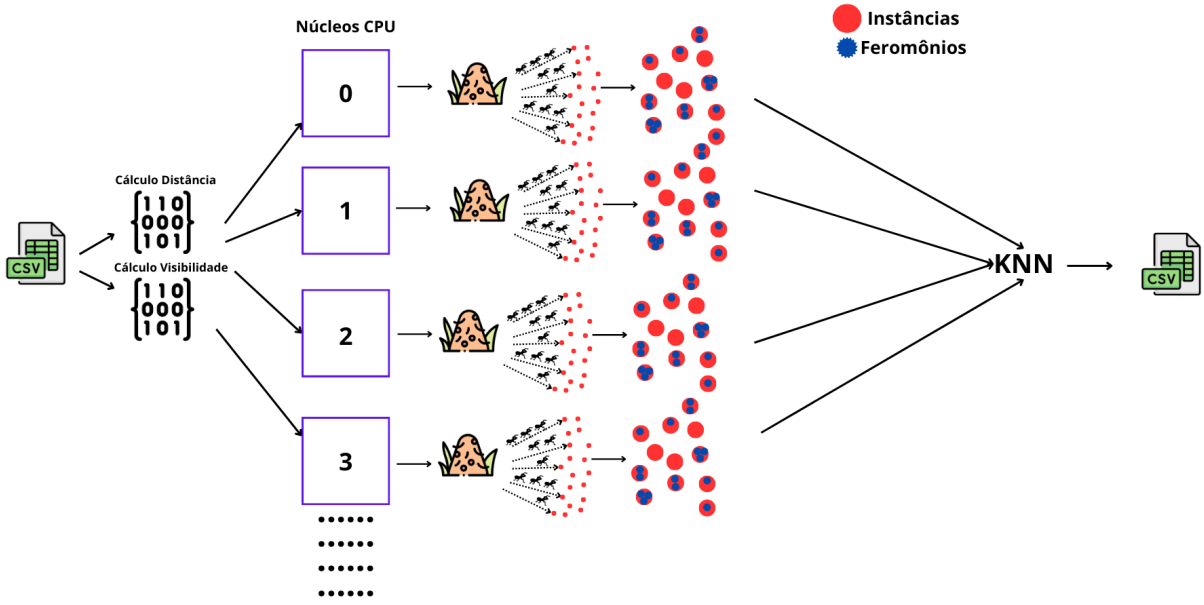
4.2 Paralelização vetorizada pela CPU

Para a vetorização, utilizou-se a biblioteca de computação numérica NumPy, essencial para a implementação de operações vetorizadas. Adicionalmente, a paralelização foi empregada por meio da biblioteca *Multiprocessing*, possibilitando a execução simultânea da busca pelas formigas e maximizando o aproveitamento da capacidade de processamento paralelo do sistema.

A implementação do código incorporou paralelização e vetorização em etapas específicas para aprimorar o desempenho. A paralelização seguiu o mesmo modelo da versão "ACO paralelizado", sem alterações nessa abordagem. Por outro lado, a vetorização substituiu de forma estratégica *loops* aninhados e operações elementares por operações matriciais utilizando a biblioteca NumPy.

Na etapa de cálculo das taxas de visibilidade, os loops duplos que processavam elementos individualmente foram substituídos por uma operação condicional vetorizada,

Figura 4: Funcionamento do ACO Paralelizado por Núcleos da CPU



Fonte: Elaborado pelo autor

permitindo a aplicação simultânea da fórmula sobre toda a matriz de distâncias. A evaporação de feromônios, anteriormente implementada com *loops*, foi reformulada como uma multiplicação direta da matriz por um fator de redução. Já no cálculo das probabilidades de escolha dos caminhos, operações matriciais foram empregadas para eliminar a necessidade de iterações sobre as instâncias, garantindo maior eficiência e elegância na execução.

De forma resumida, as principais modificações de otimização do código, por meio da vetorização, incluem a substituição de *loops* por operações matriciais com NumPy, a redução de condicionais, a atualização simultânea da matriz de feromônios sem a utilização de *loops*, a minimização de acessos individuais à memória em favor de manipulações globais, e o agrupamento de cálculos por meio da combinação de operações intermediárias em chamadas únicas, resultando em uma execução mais eficiente e elegante.

A vetorização paralelizada pode ser visualizada no pseudocódigo apresentado no Algoritmo 3. De forma geral, o método proposto para vetorizar e paralelizar na CPU é implementado da seguinte maneira:

1. **Input:** X (*feature matrix*), Y (*labels*), Q, *initial_pheromone*, *evaporation_rate* (Linha 1): Define os parâmetros de entrada para o algoritmo. O conjunto de dados é composto pela matriz de características X e pelos rótulos Y. Os parâmetros adicionais incluem a constante Q (usada para o depósito de feromônio), o valor inicial do feromônio (*initial_pheromone*) e a taxa de evaporação do feromônio (*evaporation_rate*).

Algoritmo 3: ACO Vectorized

1: **Input:** X (feature matrix), Y (labels), Q, *initial_pheromone*, *evaporation_rate*

2: *distances* ← get_pairwise_distance(X) ▷ Compute distances between features using NumPy (Operation: Euclidean Distance)

3: *visibility_rates* ← get_visibility_rates_by_distances(*distances*) ▷ Compute visibility rates using NumPy (Operation: Division)

4: *the_colony* ← create_colony($X.shape[0]$) ▷ Initialize ant colony using NumPy (Operation: Matrix Filling)

5: *ant_choices* ← create_list($X.shape[0]$) ▷ Initialize ant choices using NumPy (Operation: List Filling)

6: *pheromone_trails* ← create_pheromone_trails($X.shape[0], initial\_pheromone$) ▷ Initialize pheromone trails using NumPy (Operation: Matrix Filling)

7: *best_solution* ← get_best_solution(the_colony, X, Y) ▷ Select best solution using NumPy (Operation: Performance Evaluation)

8: *indices_selected* ← get_nonzero_indices(*best_solution*) ▷ Get indices of selected features using NumPy (Operation: Selection of Non-zero Elements)

9: *reduced_dataframe* ← get_subset_dataframe($original\_df, indices\_selected$) ▷ Get reduced dataframe using NumPy (Operation: Data Indexing)

10: *run_colony*(X, Y, *initial_pheromone, evaporation_rate,* Q) ▷ Parallel ants searching for the best solution using Multiprocessing

Fonte: Dados da Pesquisa

2. *distances* ← get_pairwise_distance(X) (Linha 2): Computa as distâncias entre as características usando a operação de distância euclidiana. A função `get_pairwise_distance` calcula a distância entre cada par de instâncias no conjunto de dados X.

3. *visibility_rates* ← get_visibility_rates_by_distances(*distances*) (Linha 3): Calcula as taxas de visibilidade, que são inversamente proporcionais às distâncias. A função `get_visibility_rates_by_distances` usa as distâncias calculadas na etapa anterior e aplica uma operação de divisão para obter essas taxas.

4. *the_colony* ← create_colony($X.shape[0]$) (Linha 4): Inicializa a colônia de formigas. A função `create_colony` cria uma estrutura (geralmente uma matriz ou lista) para armazenar as formigas na colônia, com o tamanho baseado no número de instâncias no conjunto de dados ($X.shape[0]$).

5. *ant_choices* ← create_list($X.shape[0]$) (Linha 5): Cria uma lista para armazenar as escolhas feitas pelas formigas ao longo da busca. A função `create_list` cria uma

lista vazia, que será preenchida conforme as formigas tomam suas decisões durante a execução.

6. $pheromone\_trails \leftarrow$ create_pheromone_trails($X.shape[0], initial\_pheromone$) (Linha 6): Inicializa as trilhas de feromônio. A função `create_pheromone_trails` cria uma matriz onde cada célula representa a intensidade de feromônio nas trilhas percorridas pelas formigas. O valor inicial do feromônio é dado pelo parâmetro $initial\_pheromone$.

7. $best\_solution \leftarrow$ get_best_solution(the_colony, X, Y) (Linha 7): Seleciona a melhor solução encontrada pelas formigas na colônia. A função `get_best_solution` avalia a performance das soluções de acordo com as características X e os rótulos Y, e seleciona a melhor solução com base em algum critério de desempenho (por exemplo, a maior precisão).

8. $indices\_selected \leftarrow$ get_nonzero_indices($best\_solution$) (Linha 8): Obtém os índices das características selecionadas na melhor solução. A função `get_nonzero_indices` retorna os índices das características que foram selecionadas como parte da solução final (valores não nulos ou positivos).

9. $reduced\_dataframe \leftarrow$ get_subset_dataframe($original\_df, indices\_selected$) (Linha 9): Gera um dataframe reduzido a partir do dataframe original, utilizando os índices das características selecionadas. A função `get_subset_dataframe` usa os índices selecionados para filtrar o dataframe original e manter apenas as colunas relevantes.

10. $run\_colony(X, Y, initial\_pheromone, evaporation\_rate, Q)$ (Linha 10): Executa a busca paralelizada da colônia de formigas. A função `run_colony` gerencia a execução paralela das formigas, onde cada formiga procura por uma solução em paralelo. Os parâmetros necessários são passados para guiar o processo de busca, e o uso de multiprocessing acelera a execução.

4.3 Paralelização pela GPU

A implementação pela GPU é realizada por meio das bibliotecas cuDF e cuPy, projetadas para executar operações de manipulação de dados e cálculos numéricos de forma eficiente em GPUs NVIDIA. A conversão do conjunto de dados em um *Dataframe* do cuDF armazena as informações diretamente na VRAM da GPU, permitindo a execução paralela de operações como seleção de colunas e linhas sem a necessidade de transferências constantes entre CPU e GPU. Da mesma forma, a função euclidean_distances() do cuPy armazena os arrays na VRAM e distribui os cálculos das distâncias euclidianas entre os

núcleos da GPU, garantindo maior eficiência computacional. Dentro das funções de cálculos intensivos, como ant_search_solution(), os cálculos são inteiramente realizados em arrays cuPy armazenados na VRAM, aproveitando a arquitetura massivamente paralela das GPUs NVIDIA para acelerar o processamento sem recorrer à memória da CPU.

A principal melhoria foi a substituição do uso de NumPy (executado na CPU) por CuPy, permitindo que operações matriciais sejam realizadas diretamente na VRAM, reduzindo a latência causada por transferências de dados. Além disso, a manipulação de dados foi otimizada ao utilizar cuDF para leitura e escrita de arquivos CSV, garantindo que os dados permaneçam na GPU durante todo o processamento, minimizando a movimentação entre CPU e GPU e maximizando o desempenho.

A paralelização por GPU utilizando CUDA pode ser visualizada no pseudocódigo apresentado no Algoritmo 4. De maneira geral, a abordagem proposta para paralelização utilizando a GPU é implementada da seguinte forma:

1. **Function: get_pairwise_distance_gpu(matrix: ndarray) → ndarray:** (Linha 1): A função `get_pairwise_distance_gpu` calcula as distâncias entre as instâncias da matriz de características utilizando a GPU. Essa função envolve a alocação de memória na GPU, o envio da matriz para a GPU, a execução de um *kernel* para calcular as distâncias e a transferência do resultado de volta para o CPU.

2. **allocate GPU memory for matrix** (Linha 2): Aloca a memória necessária na GPU para armazenar a matriz de entrada.

3. **transfer matrix to GPU** (Linha 3): Transfere a matriz de características para a memória da GPU para que o cálculo possa ser feito diretamente na GPU.

4. **launch GPU kernel to compute pairwise distances** (Linha 4): Lança um *kernel* na GPU para calcular as distâncias entre todas as instâncias da matriz de características, usando operações paralelizadas para otimizar o desempenho.

5. **transfer result back to CPU** (Linha 5): Transfere o resultado do cálculo (as distâncias) de volta para a memória do CPU para o processamento posterior.

6. **return result** (Linha 6): Retorna o resultado das distâncias calculadas de volta para a função que chamou `get_pairwise_distance_gpu`.

7. **Function: create_colony(num_ants) → ndarray:** (Linha 8): A função `create_colony` cria a colônia de formigas na GPU. Primeiro, ela aloca memória para a colônia na GPU, depois inicializa a colônia na CPU e a transfere para a GPU.

8. **allocate GPU memory for colony** (Linha 9): Aloca memória na GPU para armazenar os dados da colônia de formigas.

Algoritmo 4: ACO GPU

```
1: function GET_PAIRWISE_DISTANCE_GPU(matrix: ndarray) → ndarray:
2:     allocate GPU memory for matrix
3:     transfer matrix to GPU
4:     launch GPU kernel to compute pairwise distances
5:     transfer result back to CPU
6:     return result
7: end function

8: function CREATE_COLONY(num_ants) → ndarray:
9:     allocate GPU memory for colony
10:    initialize colony on CPU
11:    transfer colony to GPU
12:    return colony
13: end function

14: function GET_PHEROMONE_DEPOSIT_GPU(ant_choices: List[Tuple[int, int]],
    distances: ndarray, deposit_factor: float) → float:
15:    allocate GPU memory for ant choices and distances
16:    transfer ant choices and distances to GPU
17:    launch GPU kernel to compute pheromone deposit
18:    transfer result back to CPU
19:    return result
20: end function

21: function GET_PROBABILITIES_PATHS_ORDERED_GPU(ant: ndarray,
    visib_rates: ndarray, phe_trails: ndarray) → Tuple[Tuple[int, float]]:
22:    allocate GPU memory for ant, visibilities, and pheromone trails
23:    transfer ant, visibilities, and pheromone trails to GPU
24:    launch GPU kernel to compute probabilities
25:    transfer result back to CPU
26:    return sorted probabilities
27: end function
```

Fonte: Dados da Pesquisa

9. **initialize colony on CPU** (Linha 10): Inicializa a colônia de formigas na memória do CPU antes de transferir os dados para a GPU.

10. **transfer colony to GPU** (Linha 11): Transfere os dados da colônia para a GPU para que a busca e as atualizações de feromônio sejam realizadas na GPU.

11. **return colony** (Linha 12): Retorna a colônia de formigas para a função que a chamou.

12. **Function: get_pheromone_deposit_gpu(ant_choices: List[Tuple[int, int]],**

distances: ndarray, deposit_factor: float) → float: (Linha 14): A função `get_pheromone_deposit_gpu` calcula o depósito de feromônio nas trilhas das formigas com base nas escolhas feitas e nas distâncias calculadas. Essa operação é executada na GPU para maior desempenho.

13. **allocate GPU memory for ant choices and distances** (Linha 15): Aloca memória na GPU para armazenar as escolhas das formigas e as distâncias associadas.

14. **transfer ant choices and distances to GPU** (Linha 16): Transfere as escolhas das formigas e as distâncias para a GPU, onde o cálculo será realizado.

15. **launch GPU kernel to compute pheromone deposit** (Linha 17): Lança um *kernel* na GPU para calcular o depósito de feromônio nas trilhas, considerando as escolhas das formigas e as distâncias.

16. **transfer result back to CPU** (Linha 18): Transfere o resultado do depósito de feromônio calculado de volta para o CPU.

17. **return result** (Linha 19): Retorna o resultado do depósito de feromônio para a função que a chamou.

18. **Function: get_probabilities_paths_ordered_gpu(ant: ndarray, visib_rates: ndarray, phe_trails: ndarray) → Tuple[Tuple[int, float]]:** (Linha 21): A função `get_probabilities_paths_ordered_gpu` calcula as probabilidades das possíveis trilhas para cada formiga, ordenando-as com base na visibilidade e nas trilhas de feromônio. A operação é realizada na GPU.

19. **allocate GPU memory for ant, visibilities, and pheromone trails** (Linha 22): Aloca memória na GPU para armazenar os dados das formigas, visibilidades e trilhas de feromônio.

20. **transfer ant, visibilities, and pheromone trails to GPU** (Linha 23): Transfere os dados das formigas, as taxas de visibilidade e as trilhas de feromônio para a GPU.

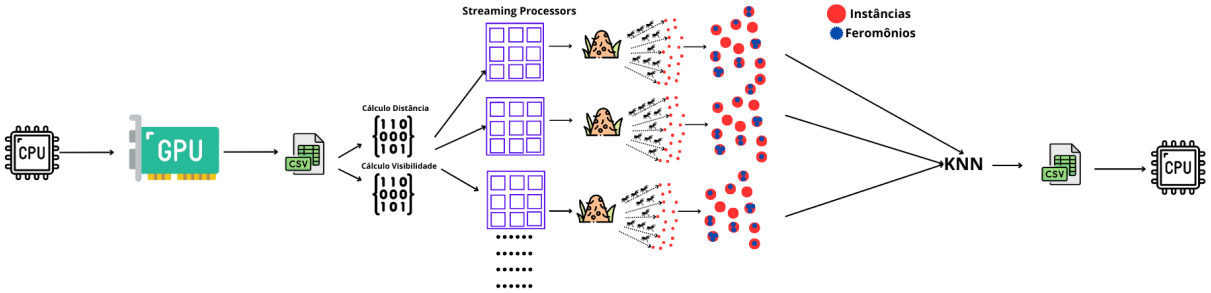
21. **launch GPU kernel to compute probabilities** (Linha 24): Lança um kernel na GPU para calcular as probabilidades das trilhas com base nas visibilidades e nas trilhas de feromônio.

22. **transfer result back to CPU** (Linha 25): Transfere o resultado das probabilidades calculadas de volta para o CPU.

23. **return sorted probabilities** (Linha 26): Retorna as probabilidades das trilhas ordenadas para a função que a chamou.

A Figura 5 ilustra a paralelização do ACO na GPU. O processo tem início na CPU, uma vez que nem todo o código foi implementado para execução na GPU. Após essa inicialização, a CPU delega à GPU a tarefa de carregar os dados da base, calcular distâncias e visibilidade, e armazenar esses resultados na VRAM. Essas informações são então distribuídas entre os SM, que coordenam a execução nos CUDA *Cores*. Nesse estágio, inúmeras formigas são criadas e exploram simultaneamente o espaço de soluções, compartilhando informações entre os CUDA Cores de um mesmo SM. Após identificar as melhores soluções, os resultados são armazenados na VRAM e submetidos ao algoritmo de aprendizado de máquina KNN, que seleciona as instâncias mais relevantes e as salva em um novo arquivo CSV. Finalmente, ao término de todos os processos na GPU, a CPU conclui a execução do programa.

Figura 5: Funcionamento do ACO pela GPU

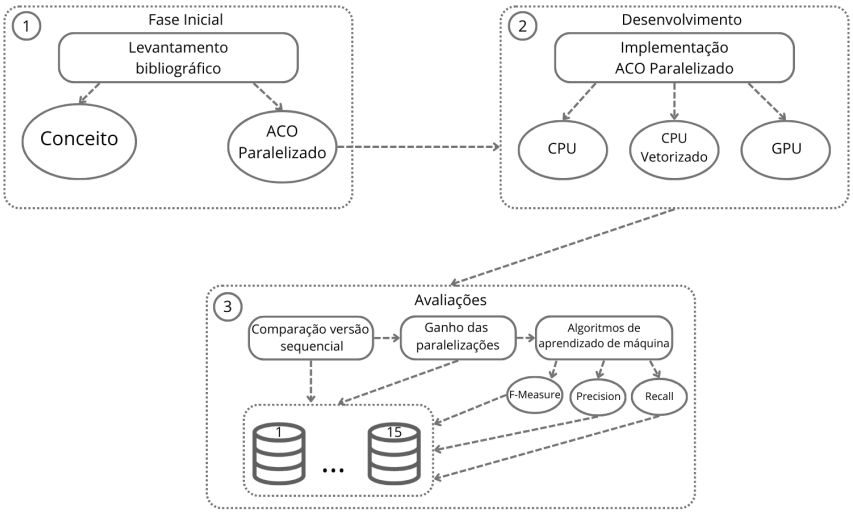


Fonte: Elaborado pelo autor

5 MATERIAIS E MÉTODOS

No presente capítulo, são detalhados os procedimentos adotados para a condução deste estudo, incluindo a descrição das bases de dados utilizadas, os algoritmos de AM empregados e as etapas de preparação e validação dos modelos. A Figura 6 fornece uma visão geral da metodologia adotada nesta dissertação, estruturada em três fases:

Figura 6: Visão geral da metodologia do trabalho



Fonte: Elaborado pelo autor

1. **Fase Inicial:** Envolve a revisão do referencial teórico sobre os diferentes métodos de paralelização do ACO, abordando conceitos fundamentais, bibliotecas utilizadas e as principais limitações inerentes a cada abordagem. Esse levantamento foi realizado na Revisão Sistemática da Literatura, conforme apresentado no Capítulo 3, página 33.

2. **Desenvolvimento:** Abrange o projeto e a implementação das diferentes estratégias de paralelização do ACO, incluindo paralelização via CPU, vetorização paralelizada na CPU e paralelização via GPU. As implementações foram desenvolvidas em Python (versão 3.10.9) e estão detalhadas no Capítulo 4, página 45.

3. **Avaliações:** O objetivo das avaliações foi verificar se as três versões paralelizadas mantêm a capacidade de selecionar as melhores instâncias, ao mesmo tempo em que

reduzem o tempo de execução e apresentam um ganho de desempenho relevante em comparação à versão sequencial. Para isso, analisou-se o tempo de execução das versões paralelizadas em relação à versão sequencial, utilizando 15 bases de dados. Após a aplicação das diferentes versões do ACO para redução das bases, quatro algoritmos de Aprendizado de Máquina — ANN, KNN, *Random Forest* e SVM — foram empregados para calcular e comparar as métricas de desempenho entre as versões paralelizadas e a versão sequencial.

5.1 Descrição das Bases de dados

Neste trabalho, foram utilizadas 15 (quinze) bases de dados públicas, selecionadas com o objetivo de abranger variações relevantes em relação ao número de instâncias e tipos de atributos. A escolha dessas bases visa garantir uma avaliação abrangente do desempenho dos métodos propostos, testando sua eficácia em diferentes cenários.

A organização das bases de dados conforme o tipo de atributo — contínuos, discretos, categóricos ordinais e não ordinais, e binários — desempenha um papel na análise da SI e no impacto sobre a eficiência dos algoritmos de AM. Atributos contínuos representam atributos numéricos com valores dentro de um intervalo, enquanto os discretos assumem apenas valores contáveis. Já os categóricos ordinais possuem uma relação de ordenação entre as categorias, ao contrário dos categóricos não ordinais, que não apresentam hierarquia. Os binários, por sua vez, são atributos limitados a duas possíveis categorias. Essa distinção estrutural influencia diretamente a forma como os algoritmos processam os dados.

Ademais, a quantidade de instâncias disponíveis em cada base possibilita a avaliação do desempenho das versões paralelizadas do ACO em diferentes escalas. Por fim, a classificação das bases de acordo com o número de classes permite a diferenciação entre problemas de classificação e regressão. É importante destacar que, como o ACO opera exclusivamente com instâncias numéricas, foi aplicada a codificação *Label Encoding* em atributos categóricos, atribuindo valores inteiros a cada categoria e, assim, garantindo a compatibilidade dos dados numéricos com o algoritmo.

A Tabela 2 apresenta a descrição completa das bases de dados, incluindo seus respectivos nomes, quantidade de instâncias, tipos de atributos de entrada, quantidade de classes e número total de atributos.

Tabela 2 – Descrição das bases de dados quanto ao número de instâncias e tipos de atributos

		Atributos de Entrada						
Base de Dados	**Inst.**	**Cont.**	**Disc.**	**C. Ord.**	**C. N. Ord.**	**Bin.**	**Classes**	**Total**
Post Operative	90	-	-	4	3	2	C(3)	9
Iris	150	4	-	1	1	-	C(3)	6
Heart Failure	299	1	6	-	-	6	C(3)	13
Data Science	607	-	3	-	4	5	C(4)	12
Cyber Salarie	1,247	-	2	5	4	-	R	11
Marketing Analysis	2,205	-	19	-	3	17	C(2)	39
DNA	3,186	-	-	-	1	180	R	181
Abalone	4,177	7	-	-	2	-	C(2)	9
Banana	5,300	2	-	-	-	1	C(3)	3
Wine	6,463	9	2	1	-	1	C(2)	13
Twonorm	7,400	20	-	-	-	1	R	21
Mushrooms	8,124	-	-	-	16	7	C(2)	23
Home Equity	10,459	5	5	1	-	1	C(3)	12
Body performance	13,394	3	4	-	3	-	R	11
Presos	14,252	129	1	-	-	-	C(2)	130

Esta tabela está ordenada, de forma crescente, pela quantidade de instâncias.
Inst.: Instâncias. *Cont.*: Contínuo. *Disc.*: Discreto. *C. Ord.*: Atributos Categóricos Ordinais. *C. N. Ord.*: Atributos Categóricos Não Ordinais. *Bin.*: Atributos Binários. *Classes*: Quantidade de Classes (C: Classificação e R: Regressão). *Total*: Quantidade Total de Atributos.

Fonte: Dados da Pesquisa

5.2 Algoritmos de Aprendizado de Máquina

Para a obtenção e comparação dos resultados das bases, foram empregados os algoritmos ANN, KNN, *Random Forest* e SVM em um *pipeline*, implementados em Python com a biblioteca *scikit-learn*, possibilitando a execução eficiente em múltiplos arquivos e a consolidação dos resultados em um único documento.

A escolha dos algoritmos ANN, KNN, *Random Forest* e SVM foi feita com base nas características e propriedades distintas de cada um, que permitem uma comparação ampla no processo de análise dos dados.

- ***Artificial Neural Network (ANN)***: Utilizado para modelar relações não lineares nos dados, adequado para problemas com grande quantidade de instâncias e atributos.

- ***K-Nearest Neighbors (KNN)***: Algoritmo simples e eficaz para problemas em que a similaridade entre instâncias é importante, sendo útil para problemas com fronteiras de decisão não lineares.

- ***Random Forest***: Modelo baseado em árvores de decisão, robusto contra *overfitting** e capaz de lidar com dados de alta dimensionalidade e variabilidade.

*Situação em que o modelo aprende demais sobre os dados de treinamento, incluindo instâncias irrelevantes ou com "ruídos", ao invés de aprender padrões gerais

- ***Support Vector Machine (SVM)***: Algoritmo eficaz para classificações binárias, útil em ambientes com alta dimensionalidade e problemas com classes desbalanceadas.

Esses algoritmos foram escolhidos por suas diferentes abordagens para análise de dados e por sua aplicabilidade em uma variedade de problemas, permitindo uma avaliação mais completa do desempenho em relação aos dados fornecidos. Todos os algoritmos foram executados com seus hiperparâmetros *default*, sem ajustes manuais. A Tabela 4 apresenta os módulos presentes na biblioteca *scikit-learn* utilizados em cada algoritmo de AM.

Quadro 4 – Tabela com os algoritmos de AM e bibliotecas implementadas no *pipeline*

Algoritmo de AM	**Biblioteca (*scikit-learn*)**
ANN	MLPClassifier
KNN	KNeighborsClassifier
Random Forest	RandomForestClassifier
SVM	SVC

Fonte: Dados da Pesquisa

Os conjuntos de dados foram particionados, destinando-se 80% para treinamento e 20% para teste. Na sequência, aplicou-se a técnica de *cross-validation* com 10 dobras exclusivamente sobre os dados de treinamento. Entre as dobras geradas, o valor mais elevado das métricas de AM alcançado durante o treinamento foi selecionado como o resultado final para o respectivo conjunto de dados.

5.3 Métricas de avaliação

5.3.1 Métricas para avaliação de qualidade dos algoritmos de Aprendizado de Máquina

Para avaliar a eficiência das bases de dados reduzidas pelo ACO, serão utilizadas as seguintes métricas de avaliação: *F-Measure*, *Precision* e *Recall*.

A *F-Measure* (Equação 5.1), representa a média harmônica entre precisão e sensibilidade.

$$F\text{-}Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{5.1}$$

A *Precision* (Equação 5.2), mede, dentre todas as instâncias classificadas em uma determinada classe, quantas são verdadeiramente da classe.

$$Precision = \frac{VP}{VP + FP} \tag{5.2}$$

A *Recall* (Equação 5.3), mede o quanto o modelo acertou dentre todas as instâncias de uma determinada classe.

$$Recall = \frac{VP}{VP + FN} \tag{5.3}$$

onde VP=Verdadeiro Positivo, VN=Verdadeiro Negativo, FP= Falso Positivo e FN =Falso Negativo.

5.3.2 Métricas para avaliação de desempenho

Para avaliar o desempenho das versões paralelizadas do ACO em execução tanto na CPU quanto na GPU, em comparação com a versão sequencial, são adotadas duas métricas de desempenho distintas: Ganho (*Speedup*) e Eficiência. Estas medidas são fundamentais para uma análise criteriosa da eficácia das implementações paralelas em relação ao aproveitamento dos recursos computacionais, especificamente em termos de Núcleos de Processamento da CPU.

O ganho (*speedup*) é uma métrica que indica o aumento de velocidade obtido ao executar um algoritmo de forma paralela, em comparação com sua versão sequencial. Essencialmente, ele quantifica a eficiência da paralelização em um algoritmo. O ganho é calculado pela razão entre o tempo de execução do algoritmo sequencial e o tempo de execução do algoritmo paralelo. Matematicamente, o ganho é representado pela Equação 5.4:

$$Ganho = \frac{Sequencial\ T}{Paralelo\ T} \tag{5.4}$$

onde:

- *Sequencial T*: Tempo de execução da versão sequencial
- *Paralelo T*: Tempo de execução da versão paralela

A eficiência é uma medida da utilização eficaz dos recursos computacionais em um sistema paralelo, refletindo o quão bem este sistema aproveita seus recursos para resolver um problema. Matematicamente, a eficiência é calculada como o ganho dividido pelo número de núcleos de processamento utilizados. Essencialmente, a eficiência expressa a relação entre o ganho de desempenho e a quantidade de recursos empregados, destacando a eficácia da paralelização. Esta relação pode ser representada pela Equação 5.5:

$$Eficiência = \frac{Ganho}{Número\ de\ Núcleos\ de\ Processamento} \tag{5.5}$$

5.3.3 Métricas estatísticas

Para avaliar a variabilidade e a confiabilidade dos resultados obtidos pelas métricas de desempenho computacional, este trabalho utiliza métricas estatísticas, como o Desvio Padrão e o Teste T.

O desvio padrão é uma medida que quantifica a dispersão ou variabilidade dos resultados em relação à média. Quanto maior o desvio padrão, maior é a flutuação dos dados em torno da média, indicando uma maior variabilidade. Por outro lado, um desvio padrão menor sugere que os valores estão mais concentrados em torno da média, refletindo uma menor variação nos resultados. A Equação 5.6 apresenta a equação do desvio padrão:

$$s = \sqrt{\frac{\sum(x_i - \bar{x})^2}{n-1}} \tag{5.6}$$

onde:

- x_i: Valores observados.
- $\bar{x}$: Média dos valores.
- n: Tamanho da Amostra.

O Teste T de *Student* para amostras independentes é um teste estatístico utilizado para comparar as médias de duas amostras independentes e verificar se há uma diferença estatisticamente relevante entre elas. Esse teste é empregado para avaliar se as melhorias observadas, em comparação com outras abordagens, são de fato estatisticamente relevantes e não meramente decorrentes de variações aleatórias. A Equação 5.7 formaliza o cálculo do teste T para amostras independentes:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{5.7}$$

onde:

- $\bar{X}_1$ e $\bar{X}_2$: Média das duas amostras.
- s_1 e s_2 : Desvio padrão das duas amostras.
- n_1 e n_2: Tamanho das amostras.

5.4 Avaliação de Escalabilidade

A avaliação de desempenho das paralelizações do ACO foi realizada por meio de testes de escalabilidade forte, nos quais o tamanho do problema, representado pela quantidade de instâncias em cada base de dados, foi mantido fixo enquanto o número de unidades de processamento aumentava. Dessa forma, a carga de trabalho permaneceu constante em todas as execuções, independentemente do grau de paralelização. Esse tipo de análise permite avaliar a eficiência do sistema ao distribuir o mesmo volume de dados entre um maior número de núcleos, observando se há ganhos de desempenho proporcionais ao aumento dos recursos computacionais.

Embora a escalabilidade forte mantenha o tamanho do problema inalterado, outras características das bases de dados, além da quantidade de instâncias, podem influenciar o processamento. A quantidade e os tipos de atributos presentes nos conjuntos de dados também variam, o que pode impactar a eficiência do algoritmo de maneiras distintas. Além disso, a análise não se restringiu apenas ao aumento do número de núcleos na CPU, mas também considerou a utilização da GPU, investigando como diferentes configurações de hardware afetam a redução do tempo de execução para um mesmo volume de dados.

5.5 Ambiente de Hardware

Todos os experimentos foram executados em uma máquina disponibilizada pelo Laboratório de Inteligência Computacional Aplicada (LICAP) da PUC Minas, conforme as seguintes especificações:

- **CPU**: Processador Intel® Xeon® E5-2696 v3 com 2.30 GHz, 18 núcleos (36 *Threads*), 45 MB de *Cache Intel® Smart* e TDP de 145W.
- **RAM**: 128 GB de RAM DDR4 a 2400 MHz em 8 *Slots*.
- **GPU**: Nvidia® GeForce RTX 3050, equipada com um total de 2560 núcleos CUDA a 1.55 GHz no *clock* base e 1.78 GHz no *boost clock*, além de 8 GB de memória GDDR6.
- **Sistema Operacional**: Ubuntu Server 22.04 LTS.

6 RESULTADOS E DISCUSSÕES

Neste capítulo são apresentados os resultados obtidos nas iterações paralelizadas pela CPU, GPU e vetorização da CPU, contrastando com a abordagem sequencial. Para analisar, comparar e monitorar o desempenho das quatro versões do ACO, identificando gargalos e avaliando as métricas geradas em cada uma nos *hardwares* utilizados, empregou-se a ferramenta *Perf*. Adicionalmente, será conduzida uma análise comparativa entre as versões paralelizadas, considerando não apenas o tempo de execução, mas também as métricas *F-measure*, *Precicion* e *Recall* avaliadas em cada conjunto de dados.

É importante destacar que todas as bases de dados com até 5300 instâncias foram submetidas a 10 iterações em cada variante do algoritmo. Para bases com mais de 5300 instâncias, foram realizadas 5 iterações devido ao longo tempo de execução, especialmente na versão sequencial do algoritmo. O tempo de execução foi determinado pela média dos tempos obtidos em cada execução.

6.1 Resultados de desempenho das paralelizações

Nesta subseção, são apresentados os tempos de execução para cada uma das versões paralelizadas pelo ACO: CPU paralelizada, CPU vetorizada e GPU. As Tabelas 3, 4 e 5 exibem as bases de dados, a quantidade de instâncias, o tempo de execução do ACO em cada uma dessas bases e o tempo paralelizado, representando, respectivamente, os ambientes de paralelização na CPU, na CPU vetorizada e na GPU. As Tabelas 3 e 4 apresentam os tempos obtidos de acordo com a quantidade de núcleos utilizados, enquanto a Tabela 5 não exibe a quantidade de núcleos CUDA, pois sua influência nos tempos de execução não foi analisada. Além disso, as tabelas comparam o desvio padrão (DP) entre as versões sequencial e paralelizada, indicando a variabilidade dos resultados obtidos. Os valores do DP estão apresentados entre colchetes "[ ]".

A medida do desvio padrão revela variações nos tempos de execução das versões paralelizadas do ACO, indicando uma faixa de tempo esperada que depende do número de núcleos da CPU e do uso da GPU. Esse desvio padrão, consistentemente baixo, tende a diminuir com o aumento do número de núcleos, refletindo maior estabilidade nos tempos de execução e maior convergência das soluções. Isso é evidenciado em diversas bases de dados, como na base *'DNA'*, onde o desvio padrão médio foi de 48,77 segundos, representando cerca de 15% a 20% da média, sugerindo variações não significativas entre os valores. Na base *'Wine'*, o desvio padrão médio foi de 8,92 minutos, com variação similar, também

Tabela 3: Tempo de execução obtido pela **paralelização** em múltiplos núcleos da CPU. Os valores entre colchetes representam os valores do desvio padrão.

Base de Dados	**Instâncias**	**Sequencial**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**
Post Operative	90	1s	0.651s	0.656s	0.670s	0.672s	0.675s	0.68s	0.70s	0.71s	0.73s
		[0.05s]	[0.04s]	[0.03s]	[0.03s]	[0.02s]	[0.04s]	[0.04s]	[0.05s]	[0.06s]	[0.07s]
Iris	150	3.64s	1.46s	1.51s	1.54s	1.57s	1.59s	1.64s	1.67s	1.69s	1.71s
		[0.12s]	[0.08s]	[0.07s]	[0.06s]	[0.06s]	[0.05s]	[0.05s]	[0.06s]	[0.07s]	[0.08s]
Heart Failure	299	20s	5.3s	5.4s	5.8s	5.9s	6s	6.2s	6.3s	6.4s	6.5s
		[0.48s]	[0.41s]	[0.36s]	[0.31s]	[0.27s]	[0.24s]	[0.23s]	[0.22s]	[0.21s]	[0.20s]
Data Science	607	1m56s	18.4s	18.5s	18.7s	18.9s	19s	19.5s	20s	21s	22s
		[3s]	[2.60s]	[2.30s]	[2.10s]	[1.80s]	[1.70s]	[1.50s]	[1.40s]	[1.30s]	[1.20s]
Cyber Salarie	1247	19m	1m49s	1m31s	1m30s	1m29s	1m28s	1m27s	1m30s	1m33s	1m35s
		[11s]	[10.00s]	[9.00s]	[7.50s]	[7.00s]	[6.40s]	[6.20s]	[5.80s]	[5.50s]	[5.30s]
Marketing Analysis	2205	1h39m	7m45s	5m46s	5m31s	5m23s	5m22s	5m13s	5m33s	5m43s	5m51s
		[1m07s]	[56s]	[47s]	[42s]	[38s]	[35s]	[33s]	[31s]	[30s]	[29s]
DNA	3186	4h54m	35m	30m	29,9m	29,6m	29,2m	28,8m	29m	30m	31m
		[5m28s]	[4m43s]	[4m12s]	[3m57s]	[3m41s]	[3m25s]	[3m11s]	[3m02s]	[2m55s]	[2m50s]
Abalone	4177	12h07m	40m	35m	32m	31m	29m30s	29m	28m55s	28m9s	27m27s
		[15m23s]	[13m12s]	[11m49s]	[10m36s]	[9m52s]	[9m19s]	[8m48s]	[8m34s]	[8m15s]	[8m00s]
Banana	5300	23h00m	53m	40m	38m	37m	33m	31m35s	31m07s	31m06s	30m38s
		[26m56s]	[22m32s]	[20m7s]	[18m41s]	[17m54s]	[17m3s]	[16m35s]	[16m12s]	[15m50s]	[15m30s]
Wine	6463	43h54m	2h03m	1h39m	1h26m	1h19m	1h14m	1h11m	1h09m	1h05m	1h01m
		[41m]	[35m14s]	[30m9s]	[27m5s]	[25m2s]	[24m8s]	[23m4s]	[22m29s]	[22m15s]	[22m00s]
Twonorm	7400	68h57m	3h20m	2h51m	2h47m	2h43m	2h29m	2h21m	2h13m	2h21m	2h35m
		[1h12m]	[1h01m]	[53m7s]	[48m5s]	[44m3s]	[42m1s]	[40m9s]	[39m5s]	[38m2s]	[37m10s]
Mushrooms	8124	92h12m	7h24m	6h27m	6h10m	5h57m	5h50m	5h46m	5h39m	5h32m	5h27m
		[1h41m]	[1h20m]	[1h11m]	[1h02m]	[58m7s]	[55m3s]	[53m9s]	[52m4s]	[51m29s]	[51m03s]
Home Equity	10459	191h40m	15h09m	13h10m	12h31m	12h19m	12h06m	11h59m	11h48m	11h36m	11h27m
		[3h33m]	[3h02m]	[2h41m]	[2h26m]	[2h17m]	[2h11m]	[2h06m]	[2h01m]	[1h59m]	[1h57m]
Body performance	13394	434h52m	20h55m	19h59m	19h11m	18h43m	18h31m	18h10m	17h56m	17h43m	17h29m
		[7h48m]	[6h32m]	[5h47m]	[5h13m]	[4h49m]	[4h32m]	[4h21m]	[4h13m]	[4h11m]	[4h07m]
Presos	14252	530h8m	29h49m	29h03m	28h37m	27h57m	27h19m	26h51m	26h27m	25h53m	25h14m
		[9h11m]	[7h46m]	[7h02m]	[6h33m]	[6h04m]	[5h48m]	[5h32m]	[5h37m]	[5h31m]	[5h23m]

Fonte: Dados da Pesquisa

entre 15% e 20% da média. Além disso, na base *'Post Operative'*, o desvio padrão diminui de 0,05s para 0,02s ao passar de 2 para 8 núcleos, o que reflete uma maior convergência das soluções do ACO com a paralelização. Essas reduções indicam que o aumento no número de núcleos melhora a precisão e a estabilidade dos resultados, evidenciando a eficácia da paralelização no ACO.

Os resultados obtidos das paralelizações em CPU, CPU vetorizada e GPU evidenciaram variações notáveis de desempenho conforme o número de núcleos utilizados e o tipo de processamento, sendo importante destacar que tais variações não ocorreram para poucas instâncias, como evidenciado na base *'Post Operative'*. Esse comportamento levanta uma questão de escalabilidade para cenários com um número reduzido de instâncias. Na paralelização CPU, observou-se uma diminuição progressiva do tempo de execução com o aumento do número de núcleos, embora a redução do tempo nem sempre fosse linear, sugerindo uma eficiência marginal decrescente com a adição de mais núcleos.

Tabela 4: Tempo de execução obtido através da **paralelização pela vetorização** em múltiplos núcleos da CPU. Os valores entre colchetes representam os valores do desvio padrão.

Base de Dados	**Instâncias**	**Sequencial**	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**
Post Operative	90	1s	0.64s	0.69s	0.66s	0.67s	0.67s	0.67s	0.67s	0.70s	0.70s	0.71s
		[0.05s]	[0.04s]	[0.03s]	[0.03s]	[0.02s]	[0.02s]	[0.02s]	[0.03s]	[0.03s]	[0.04s]	[0.04s]
Iris	150	3.64s	2.17s	1.51s	1.52s	1.52s	1.56s	1.56s	1.57s	1.57s	1.58s	1.59s
		[0.12s]	[0.08s]	[0.07s]	[0.06s]	[0.05s]	[0.05s]	[0.05s]	[0.06s]	[0.06s]	[0.07s]	[0.08s]
Heart Failure	299	20s	6s	5.3s	5.1s	4.5s	4.4s	4.3s	4.2s	4.6s	4.7s	4.8s
		[0.48s]	[0.41s]	[0.36s]	[0.31s]	[0.27s]	[0.24s]	[0.23s]	[0.22s]	[0.21s]	[0.20s]	
Data Science	607	1m56s	20s	18s	14.4s	14s	13.7s	13.6s	14s	15s	16s	18s
		[3s]	[2.60s]	[2.30s]	[2.10s]	[1.80s]	[1.70s]	[1.50s]	[1.40s]	[1.30s]	[1.20s]	
Cyber Salarie	1247	19m	2m	1m46s	1m26s	1m20s	1m17s	1m18s	1m19s	1m20s	1m21s	1m24s
		[11s]	[10.00s]	[9.00s]	[7.50s]	[7.00s]	[6.40s]	[6.20s]	[5.80s]	[5.50s]	[5.30s]	
Marketing Analysis	2205	1h39m	8m	6m32s	5m21s	5m17s	5m05s	4m53	5m03s	5m10s	5m15s	5m26s
		[1m07s]	[56s]	[47s]	[42s]	[38s]	[35s]	[33s]	[31s]	[30s]	[29s]	
DNA	3186	4h54m	31m	30m	28m	26m	25m	24m	24m	25m12s	24m42s	24m24s
		[5m28s]	[4m43s]	[4m12s]	[3m57s]	[3m41s]	[3m25s]	[3m11s]	[3m02s]	[2m55s]	[2m50s]	
Abalone	4177	12h7m	46m	30m	30m	27m	26m	25m30s	25m	24m30s	24m	24m
		[15m23s]	[13m12s]	[11m49s]	[10m36s]	[9m52s]	[9m19s]	[8m48s]	[8m34s]	[8m15s]	[8m00s]	
Banana	5300	23h00	1h24m	49m	36m	32m	29m47s	30m	28m	27m	26m	25m
		[26m56s]	[22m32s]	[20m7s]	[18m41s]	[17m54s]	[17m3s]	[16m35s]	[16m12s]	[15m50s]	[15m30s]	
Wine	6463	43h54m	2h35m	2h15m	1h30m	1h20m	1h15m	1h08m	1h05m	1h00m	55m	50m
		[41m]	[35m14s]	[30m9s]	[27m5s]	[25m2s]	[24m8s]	[23m4s]	[22m29s]	[22m15s]	[22m00s]	
Twonorm	7400	68h57m	3h33m	3h01m	2h27m	2h21m	2h19m	2h07m	2h02m	1h51m	1h44m	1h31m
		[1h12m]	[1h01m]	[53m7s]	[48m5s]	[44m3s]	[42m1s]	[40m9s]	[39m5s]	[38m2s]	[37m10s]	
Mushrooms	8124	92h12m	6h44m	6h22m	5h22m	5h17m	4h52m	4h50m	4h43m	4h38m	4h31m	4h15m
		[1h41m]	[1h20m]	[1h11m]	[1h02m]	[58m7s]	[55m3s]	[53m9s]	[52m4s]	[51m29s]	[51m03s]	
Home Equity	10459	191h40m	13h57m	13h40m	12h1m	11h29m	11h4m	11h00m	10h52m	10h37m	10h33m	10h23m
		[3h33m]	[3h02m]	[2h41m]	[2h26m]	[2h17m]	[2h11m]	[2h06m]	[2h01m]	[1h59m]	[1h57m]	
Body performance	13394	434h52m	19h17m	19h38m	18h15m	17h37m	17h15m	16h21m	16h10m	15h43m	15h22m	14h53m
		[7h48m]	[6h32m]	[5h47m]	[5h13m]	[4h49m]	[4h32m]	[4h21m]	[4h13m]	[4h11m]	[4h07m]	
Presos	14252	530h8m	28h7m	27h17m	25h51m	25h29m	24h59m	24h33m	24h05m	23h42m	23h11m	22h35m
		[9h11m]	[7h46m]	[7h02m]	[6h33m]	[6h04m]	[5h48m]	[5h32m]	[5h37m]	[5h31m]	[5h23m]	

Fonte: Dados da Pesquisa

Na CPU vetorizada, os ganhos de desempenho foram ainda mais pronunciados, com tempos de execução menores em comparação à paralelização tradicional, especialmente em bases de dados maiores como *'Twonorm'* e *'Mushrooms'*. Na vetorização GPU, os tempos de execução foram drasticamente reduzidos em comparação com as versões sequenciais e de CPU, apresentando uma superioridade clara da GPU para tarefas de processamento intensivo de dados.

Na Figura 7, são apresentados conjuntos de gráficos de pontos que ilustram o ganho obtido em cada versão paralelizada pela CPU do ACO, variando conforme a quantidade de núcleos utilizados. O eixo vertical representa o ganho, enquanto o eixo horizontal indica a quantidade de núcleos. A ordem dos gráficos está de acordo com a quantidade de instâncias, de maneira crescente.

Para a maioria das bases de dados analisadas, o ganho de desempenho tende a se intensificar com o aumento do número de núcleos. A versão paralelizada do ACO na CPU geralmente apresenta um incremento de desempenho estável e previsível à medida que

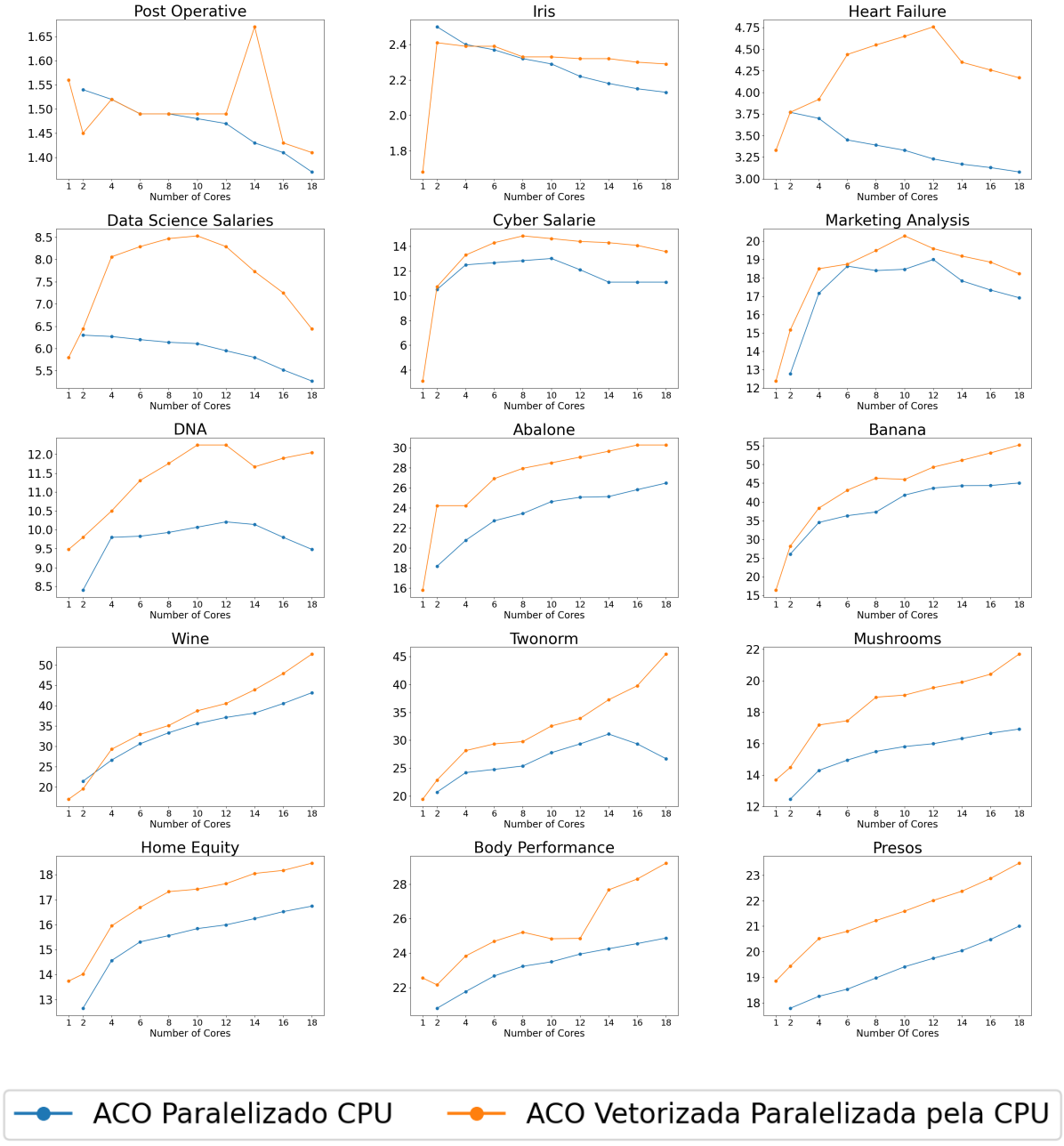
Tabela 5: Tempo de execução obtido através da paralelização pela GPU. Os valores entre colchetes representam os valores do desvio padrão.

Base de Dados	**Instâncias**	**Sequencial**	**GPU**
Post Operative	90	1s [0.05s]	0.60s [0.02s]
Iris	150	3.64s [0.12s]	1.38s [0.14s]
Heart Failure	299	20s [0.48s]	3s [0.31s]
Data Science	607	1m56s [3s]	10s [0.88s]
Cyber Salarie	1247	19m [11s]	1m [2.04s]
Marketing Analysis	2205	1h39m [1m07s]	4m [12.21s]
DNA	3186	4h54m [5m28s]	14m [2m03s]
Abalone	4177	12h7m [15m23s]	20m [4m56s]
Banana	5300	23h00 [26m56s]	27m [5m04s]
Wine	6463	43h54m [41m]	35m [7m16s]
Twonorm	7400	68h57m [1h12m]	47m [9m34s]
Mushrooms	8124	92h12m [1h41m]	1h07m [11m55s]
Home Equity	10459	191h40m [3h33m]	2h47m [14m56s]
Body performance	13394	434h52m [7h48m]	5h57m [22m57s]
Presos	14252	530h8m [9h11m]	7h33m [38m52s]

Fonte: Dados da Pesquisa

mais núcleos são empregados, embora com algumas variações. A partir da base de dados *'Abalone'*, observa-se que, conforme o número de instâncias nas bases de dados cresce, o ganho de desempenho também se expande. A versão paralelizada e vetorizada do ACO na CPU exibe um ganho similar, com pequenas vantagens adicionais em determinadas

Figura 7: Ganhos obtidos pela ***paralelização*** e ***vetorização*** em cada núcleo da CPU em cada base de dados

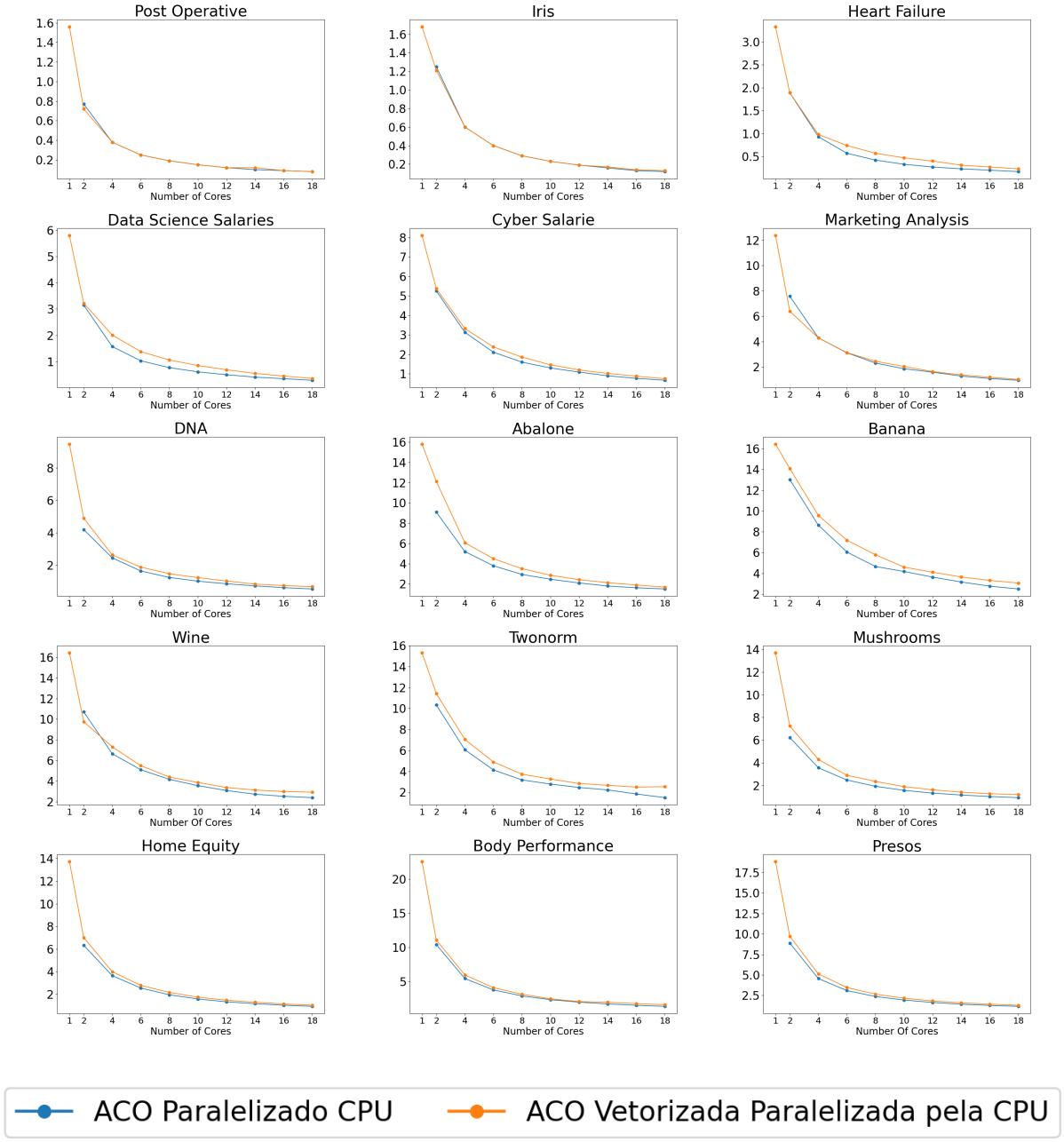


Fonte: Dados da Pesquisa

bases de dados.

Seguindo a lógica de visualização da Figura 7, a Figura 8 também exibe conjuntos de gráficos de pontos. No entanto, nesta figura, é ilustrada a eficiência obtida em cada versão paralelizada do ACO, variando conforme a quantidade de núcleos utilizados. O eixo vertical representa a eficiência, enquanto o eixo horizontal indica a quantidade de núcleos.

Figura 8: Eficiência obtida pela ***paralelização*** e ***vetorização*** em cada núcleo da CPU em cada base de dados



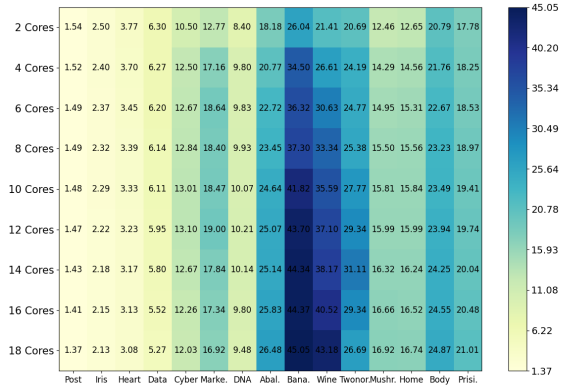
Fonte: Dados da Pesquisa

Ambas as abordagens apresentam eficiências de paralelismo bastante similares, indicando que o algoritmo está aproveitando efetivamente o paralelismo disponível. No entanto, os gráficos sugerem que, embora a versão vetorizada apresente ganhos maiores em termos de desempenho, para bases de dados como “Heart Failure” e “Data Science Salaries”, na maioria dos casos, a vetorização não oferece uma vantagem clara sobre a paralelização tradicional na CPU em termos de eficiência e aproveitamento de recursos. Por exemplo, em “Heart Failure”, a eficiência da vetorização e da paralelização tradicional é bastante próxima, especialmente quando o número de núcleos aumenta. Da mesma

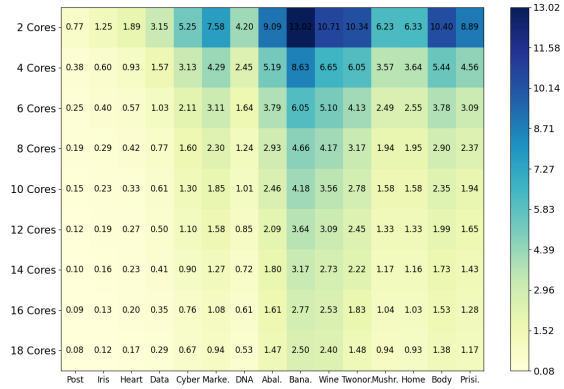
forma, em “Data Science Salaries”, a vetorização mostra apenas uma melhoria marginal em relação à paralelização tradicional.

Outra forma de visualização são os gráficos da Figura 9, que apresentam *heatmaps* para uma comparação visual detalhada do ganho e da eficiência entre as versões paralelizada na CPU e vetorizada paralelizada na CPU. Em cada gráfico, os valores obtidos por núcleo em cada base de dados são representados por cores, variando do amarelo para valores mais baixos até tons azulados à medida que os valores aumentam. Além disso, os *heatmaps* evidenciam uma tendência de aumento no desempenho conforme o número de núcleos cresce.

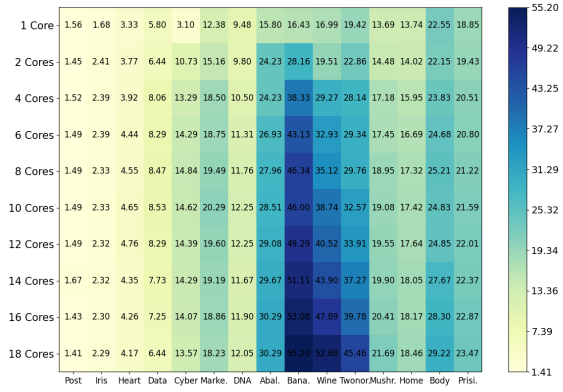
Figura 9: Ganho e Eficiência obtidos pela ***paralelização*** e ***vetorização*** em cada núcleo da CPU



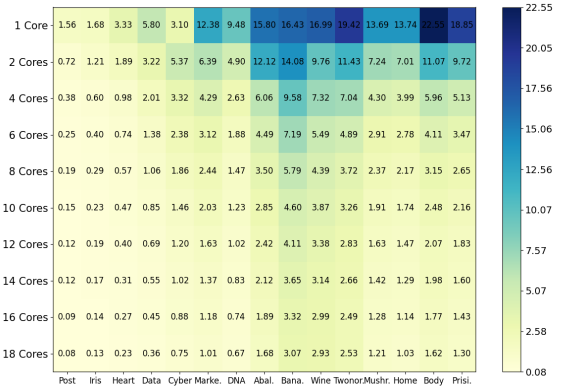
(a) Ganho obtido pela paralelização



(b) Eficiência obtida pela paralelização



(c) Ganho obtido pela vetorização



(d) Eficiência obtida pela vetorização

Fonte: Dados da Pesquisa

Na Figura 9(a), o *heatmap* de ‘ganho’ ilustra como o desempenho do algoritmo melhora à medida que mais núcleos são utilizados. Observa-se que o ganho é mais pronunciado nas bases de dados com maior número de instâncias, como nas bases *‘Banana’* e *‘Wine’*, onde o aumento do número de núcleos resulta em ganhos médios significativos,

da ordem de aproximadamente 50%. Em contrapartida, para bases de dados com menos instâncias , como *'Post Operative'* e *'Heart Failure'*, o ganho é menor e se estabiliza com o aumento dos núcleos.

Na Figura 9(c), o *heatmap* de ganho para a versão vetorizada mostra padrões semelhantes ao observado na paralelização simples, com um ganho de desempenho em bases de dados com grandes instâncias como *'Twonorm'* e *'Body Performance'*. No entanto, a vetorização proporciona ganhos adicionais em algumas bases de dados, como *'Abalone'* e *'Marketing Analysis'*, onde se observa um desempenho superior em comparação com a versão apenas paralelizada. Em ambas as versões, o *heatmap* de eficiência revela uma redução na eficiência à medida que o número de núcleos aumenta, com valores semelhantes em ambas as abordagens.

Uma alternativa de visualização, cujo foco principal é a comparação entre as quatro versões do ACO (sequencial, paralelizada em CPU, vetorizada e paralelizada em CPU, e paralelizada em GPU), está ilustrada na Figura 10. A visualização é apresentada em uma escala logarítmica, proporcionando uma análise mais clara das diferenças de desempenho entre as versões à medida que o número de instâncias aumenta. O eixo x representa a quantidade de instâncias processadas, enquanto o eixo y exibe o tempo de execução em minutos, aplicando a transformação logaritmica natural para melhor visualização das diferenças.

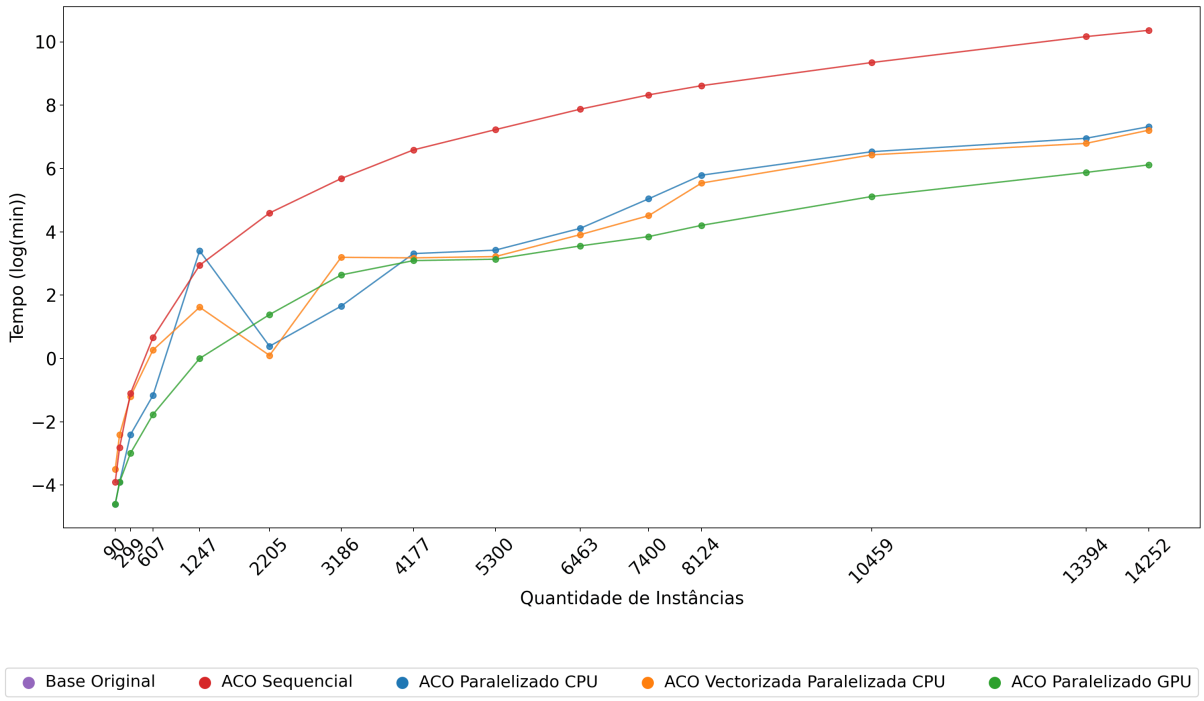
A versão sequencial apresenta o maior tempo de execução entre todas as abordagens, exibindo um crescimento exponencial à medida que o número de instâncias aumenta. Em contraste, a versão paralelizada reduz significativamente o tempo de processamento. Embora essa diferença seja menos expressiva em bases com um número reduzido de instâncias, à medida que a quantidade cresce, a eficácia da paralelização torna-se mais evidente.

A versão vetorizada, por sua vez, apresenta um comportamento variável: em algumas bases, seu tempo de execução se aproxima do da versão paralelizada, enquanto em outras, exibe oscilações, como observado nas 2205 instâncias da base "Marketing Analysis".

Já a execução na GPU apresenta um desempenho superior à versão sequencial, mas, em certos cenários, fica aquém das versões paralelizada e vetorizada. Esse comportamento sugere que a sobrecarga gerada pela transferência de dados entre a CPU e a GPU pode impactar o desempenho,especialmente em bases menores. No entanto, para conjuntos de dados maiores, como na base 'Wine' com 6463 instâncias, a versão em GPU mostra-se consistentemente mais eficiente do que as demais abordagens.

Em resumo, as versões paralelizada e vetorizada apresentam maior eficiência do que a versão sequencial em grandes volumes de dados, com a escolha entre elas dependendo do tamanho da base. A versão em GPU pode ser vantajosa para bases muito grandes,

Figura 10: Quantidade de Instâncias x Tempo em cada versão



Fonte: Dados da Pesquisa

mas pode sofrer penalizações devido à sobrecarga na transferência de dados. Os tempos de execução evidenciaram uma redução notável com a paralelização: a CPU paralelizada apresentou redução de tempo com o aumento de núcleos, enquanto a CPU vetorizada obteve ganhos ainda mais expressivos, especialmente em bases de dados maiores como *'Twonorm'* e *'Mushrooms'*. A GPU, por sua vez, apresentou os menores tempos de execução, destacando-se em tarefas que exigem alto poder de processamento, aproveitando sua arquitetura naturalmente paralelizável.

Os resultados, portanto, ressaltam a eficácia das abordagens paralelizadas, com a GPU oferecendo os melhores tempos de execução, seguida pela CPU vetorizada, embora a eficácia geral entre CPU paralelizada e vetorizada seja similar.

Esse comportamento está de acordo com o que foi identificado na revisão sistemática, apresentada no Capítulo 3, na qual a maioria dos estudos reporta que a paralelização pela CPU e a vetorização paralelizada resultam em ganhos médios de desempenho de aproximadamente 10 vezes (PEAKE et al., 2018; BAYDOGMUS, 2022), um valor próximo ao obtido nesta dissertação de 16.63 na versão somente paralelizada e 17.69 na versão vetorizada e paralelizada.

Além disso, observa-se que, a partir de um determinado volume de instâncias, o aumento do número de núcleos contribui diretamente para um maior ganho computaci-

onal. A GPU, por sua vez, apresentou os melhores resultados entre as três abordagens analisadas, o que também é corroborado por estudos prévios (ZHOU; HE; QIU, 2017; ARNAUTOVIć et al., 2013) que indicam que, em alguns cenários, essa arquitetura pode alcançar ganhos de desempenho próximos a 100 vezes, mas em média chega a uma melhora de 50 vezes, valor próximo ao que foi obtido nesta dissertação que foi de 41.84 vezes, evidenciando sua superioridade em problemas altamente paralelizáveis.

6.2 Análise dos Contadores de Hardware

Para analisar, comparar e monitorar o desempenho das quatro versões do ACO, identificando gargalos e avaliando as métricas geradas em cada uma, utilizou-se a ferramenta *Perf*, disponível nos sistemas operacionais Linux. Para isso, foi utilizada a base de dados *"Abalone"*, escolhida por seu tamanho de 4.177 instâncias. Esse tamanho permite testar o desempenho das versões do ACO de forma eficiente, proporcionando dados suficientes para observar variações relevantes nas métricas de desempenho, sem o volume excessivo encontrado em bases de dados maiores. As métricas analisadas, que medem o uso do *hardware*, incluem o total de instruções obtidos na CPU, total de ciclos obtidos na CPU, taxa de *cache-misses*, *context-switches* e *page-faults*, proporcionando uma visão abrangente do comportamento computacional de cada versão do ACO.

A taxa de *cache-misses* quantifica a frequência com que a CPU não encontra os dados necessários na *cache*, resultando em mais acessos à memória principal. A *cache* L1 é a memória cache de nível 1, localizada diretamente no núcleo da CPU, oferecendo a menor latência e acesso rápido aos dados frequentemente utilizados. O LLC (*Last Level Cache*) é o nível final de *cache* da CPU, compartilhado entre seus núcleos. Ele atua como um intermediário entre a *cache* L1 e a memória principal, reduzindo a necessidade de acessos à RAM. No processador utilizado, que adota a microarquitetura Haswell, o LLC é compartilhado por todos os núcleos. Os *context-switches* representam o número de vezes que a CPU alterna entre diferentes processos, o que pode impactar o desempenho devido à sobrecarga associada à troca de contexto. Por fim, os *page-faults* indicam falhas de página, ou seja, acessos à memória que exigem a recuperação de dados do disco ou de áreas menos acessíveis da RAM, o que pode comprometer a velocidade de execução do código.

A Tabela 6 apresenta o desempenho conforme as métricas previamente descritas, obtido por meio da ferramenta Perf, comparando a versão sequencial com as versões paralelizadas do ACO. Cabe destacar que, devido à disparidade nas ordens de magnitude dos valores — alguns valores na casa dos milhões e outros na casa dos milhares —, nas métricas de *Context-switches* e *Page-faults* foram utilizados as notações "K" e "M". O "K" representa milhares e o "M" representa milhões.

Tabela 6: Análise Comparativa das Métricas de acesso ao *Hardware* utilizando a Ferramenta Perf

Métrica	**Sequencial**	**Paralelizado CPU**	**Vetorizado CPU**	**Paralelizado GPU**
Instruções (bilhões)	337.78	22.84	18.45	8.12
Ciclos (bilhões)	154.06	11.59	9.33	5.83
Cache-misses L1 (%)	1.31	0.96	0.77	0.34
Cache-misses LLC (%)	7.98	2.10	1.53	1.18
Context-switches	1.13M	188.71K	95.43K	43.68K
Page-faults	86.50M	277.36K	154.33K	79.43K

Na versão paralelizada na CPU, as instruções totais executadas reduziram de 337,78 bilhões para 22,84 bilhões, enquanto a quantidade total de ciclos executados diminuiu de 154,06 bilhões para 11,59 bilhões. Essa redução decorre da divisão das tarefas entre múltiplos núcleos, o que introduz uma sobrecarga de sincronização, mas melhora a distribuição do processamento. A memória *cache* Intel® *Smart Cache* (45 MB) e a RAM de 128 GB contribuíram para a mitigação desse impacto, reduzindo a latência de acesso aos dados. Além disso, os *context-switches* e os *page-faults* apresentaram reduções de aproximadamente 83% e 99,7%, respectivamente, indicando uma execução mais eficiente das tarefas.

Na versão vetorizada paralelizada na CPU, observou-se uma redução adicional nas métricas. As instruções e ciclos totais apresentaram decréscimos percentuais de 19,2% e 19,5%, respectivamente, em comparação à versão paralelizada na CPU. Esse resultado é atribuído à utilização das instruções Intel® AVX2, que permitem a execução de múltiplas operações vetoriais por ciclo de *clock*. Os *cache-misses* L1 e LLC foram percentualmente reduzidos em 19,8% e 27,1%, respectivamente, apresentando um uso mais eficiente dos dados armazenados em cache. Além disso, os *context-switches* e os *page-faults* apresentaram quedas percentuais de 49,4% e 44,4%, respectivamente, reforçando a eficiência da vetorização na minimização dos acessos à memória principal.

Na versão paralelizada na GPU, as reduções nas métricas de CPU foram mais expressivas devido à execução da maior parte do código nesse *hardware*, que possui uma arquitetura massivamente paralela para processamento simultâneo de um grande volume de operações. As instruções e ciclos totais caíram para 8,12 bilhões e 5,83 bilhões, representando reduções percentuais de 64,0% e 37,5%, respectivamente, em comparação à versão vetorizada na CPU. Os *cache-misses* L1 e LLC atingiram os menores valores registrados, 0,34% e 1,18%, respectivamente, refletindo a eficiência da memória VRAM de 8 GB de memória GDDR6, que minimizou a necessidade de acessos à memória principal. Além disso, os *context-switches* e os *page-faults* apresentaram reduções percentuais de 54,2% e 48,5%, respectivamente, devido à execução massiva de CUDA *cores* na GPU, reduzindo a alternância entre processos e interrupções no fluxo de execução na CPU.

Os resultados apresentados na Tabela 6 indicam que todas as versões paralelizadas reduziram de forma relevante as métricas de *hardware* em comparação com a versão sequencial. Esses avanços são resultado das otimizações algorítmicas e da utilização de *hardwares* com maior capacidade de processamento paralelo e eficiência no gerenciamento de memória. A exploração de múltiplos recursos, como núcleos e memórias, melhora o desempenho de tarefas computacionalmente intensivas e permite que os ganhos de desempenho superem as previsões da Lei de Amdahl. A Lei de Amdahl afirma que o ganho de desempenho de uma aplicação paralelizada é limitado pela fração do código que não pode ser paralelizada. No entanto, os resultados demonstram que os ganhos obtidos superaram as limitações teóricas indicadas por essa lei, devido à eficácia da paralelização e à arquitetura de hardware utilizada.

6.3 Resultados da redução de instâncias pelo ACO

Para avaliar e comparar a redução das instâncias do ACO e verificar se o comportamento da versão paralelizada se assemelha ao da versão sequencial, as Tabelas 7, 8 e 9 apresentam, respectivamente, a comparação entre a versão sequencial e suas versões paralelizadas na CPU, vetorizadas na CPU e executadas na GPU.

Tabela 7: Quantidade total de instâncias reduzidas na versão paralelizada pela CPU em função do número de núcleos

Base de Dados	**Instâncias**	**Sequencial**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**
Post Operative	90	47	46	45	43	41	42	44	45	47	43
Iris	150	83	76	74	69	72	68	73	79	69	76
Heart Failure	299	150	161	136	153	146	157	151	148	138	142
Data Science	607	307	326	311	302	289	280	276	295	308	319
Cyber Salarie	1247	619	667	636	615	574	568	610	632	651	589
Marketing Analysis	2205	1107	1044	1184	1150	1093	1014	1002	1076	1131	1116
DNA	3186	1474	1704	1627	1571	1504	1612	1664	1446	1557	1467
Abalone	4177	2174	2242	2140	2081	1986	1928	1897	2030	2122	2193
Banana	5300	2634	2836	2703	2433	2412	2586	2682	2613	2767	2497
Wine	6463	3361	3604	3693	3463	3959	3915	3737	3688	3395	3552
Twonorm	7400	3781	3959	3915	3552	3341	3330	3693	3863	3744	3360
Mushrooms	8124	4148	4005	3838	4241	3969	4111	4049	4296	3742	3690
Home Equity	10459	5201	5344	4813	4750	4941	5460	5159	4707	5112	5293
Body performance	13394	6027	7194	6847	6638	6342	6168	6081	6523	6776	7020
Presos	14252	7294	7057	6961	6437	6470	7169	6667	6561	7668	7362

Fonte: Dados da Pesquisa

Com base nos dados apresentados na Tabela 7, observou-se que a média de redução percentual da versão sequencial, em comparação ao tamanho total da base de dados, foi de aproximadamente 49,5%. Isso implica que, em média, a versão sequencial foi capaz de reduzir quase metade das instâncias originais. Quanto ao impacto do número de núcleos na versão paralelizada, observa-se que, embora o aumento do número de núcleos altere a redução de instâncias, esse efeito não segue um padrão linear. Em alguns casos, como na

base *"Post Operative"*, a redução foi maior com 8 núcleos (54,4% de redução em relação à base original), enquanto em outros, como na base *"Body Performance"*, a redução foi menor com 12 núcleos (54,6% em relação à base original). De forma geral, o número de núcleos que proporcionou a maior redução foi o de 10 núcleos, com uma redução média de aproximadamente 50,8% em relação ao tamanho original das bases de dados. Assim, conclui-se que a utilização de um número maior de núcleos não resultou, necessariamente, em uma redução proporcionalmente maior. Considerando todos os núcleos, a média total da redução das instâncias na versão paralelizada foi de aproximadamente 50,2%.

Tabela 8: Quantidade total de instâncias reduzidas na versão vetorizada paralelizada pela CPU em função do número de núcleos

Base de Dados	**Instâncias**	**Sequencial**	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**
Post Operative	90	47	47	48	45	44	42	41	43	45	48	46
Iris	150	83	75	79	77	75	71	70	69	72	76	79
Heart Failure	299	150	170	158	155	149	144	140	146	135	151	137
Data Science	607	307	321	307	296	284	279	274	291	303	340	314
Cyber Salarie	1247	619	686	660	630	608	584	572	563	599	622	645
Marketing Analysis	2205	1107	1131	1115	1166	1100	992	1139	1058	1074	1012	997
DNA	3186	1474	1643	1552	1609	1462	1591	1685	1647	1440	1529	1490
Abalone	4177	2174	2212	2242	2140	2081	1986	1928	1897	2193	2122	2030
Banana	5300	2634	2701	2396	2804	2677	2544	2645	2581	2478	2738	2428
Wine	6463	3361	3744	3611	3493	3826	3648	3863	3781	3401	3360	3341
Twonorm	7400	3781	3719	3604	3737	3648	3463	3395	3688	3826	3493	3611
Mushrooms	8124	4148	4156	3671	3727	4054	4197	3656	3956	3900	4103	3804
Home Equity	10459	5201	5413	5020	4727	5221	5282	5403	5533	4797	4895	5095
Body performance	13394	6027	6870	7059	6750	6508	6278	6135	6029	6429	6670	6897
Presos	14252	7294	7223	6744	7540	7211	6939	7197	7440	6537	7112	6413

Fonte: Dados da Pesquisa

Com base nos dados apresentados na Tabela 8, observa-se que o impacto do número de núcleos na versão vetorizada paralelizada também não segue um padrão linear. Em alguns casos, como na base *"Cyber Salarie"*, a redução foi maior com 10 núcleos (54,1% de redução em relação à base original), enquanto em outros, como na base *"DNA"*, a redução foi menor com 6 núcleos (54,1% em relação à base original). De forma geral, o número de núcleos que proporcionou a maior redução foi o de 10 núcleos, com uma redução média de aproximadamente 50,5% em relação ao tamanho original das bases de dados. Assim, conclui-se que a utilização de um número maior de núcleos também não resultou, necessariamente, em uma redução proporcionalmente maior. Considerando todos os núcleos, a média total da redução das instâncias na versão vetorizada paralelizada foi de aproximadamente 50,3%.

Com base nos dados apresentados na Tabela 9, observa-se que a versão paralelizada em GPU apresentou resultados variados em relação à redução de instâncias. Em alguns casos, como na base *"Marketing Analysis"*, a redução foi de 1032 instâncias reduzidas (46,8% em relação ao tamanho da base original), enquanto em outros, como na base *"Mushrooms"*, a redução foi menor, com 4346 instâncias restantes (46,5% em relação ao

Tabela 9: Quantidade total de instâncias reduzidas na versão paralelizada pela GPU

Base de Dados	**Instâncias**	**Sequencial**	**GPU**
Post Operative	90	47	42
Iris	150	83	70
Heart Failure	299	150	147
Data Science	607	307	305
Cyber Salarie	1247	619	572
Marketing Analysis	2205	1107	1032
DNA	3186	1474	1587
Abalone	4177	2174	1880
Banana	5300	2634	2387
Wine	6463	3361	3248
Twonorm	7400	3781	3401
Mushrooms	8124	4148	4346
Home Equity	10459	5201	5596
Body performance	13394	6027	6730
Presos	14252	7294	6841

Fonte: Dados da Pesquisa

tamanho da base original). Considerando todas as bases, a média total da redução das instâncias na versão paralelizada em GPU foi de aproximadamente 49,8% em relação ao tamanho original das bases de dados.

Com base nos resultados apresentados, conclui-se que as versões paralelizadas obtiveram reduções percentuais próximas às da versão sequencial, indicando que as paralelizações não tiveram um impacto relevante na redução das instâncias. A versão sequencial alcançou uma redução média de aproximadamente 49,5%, enquanto as versões paralelizadas pela CPU apresentaram reduções médias de 50,2% e 50,3%, respectivamente. A versão paralelizada pela GPU, por sua vez, obteve uma redução média de 49,8%, com variações mais acentuadas em algumas bases de dados. Esses resultados indicam que, embora a paralelização tenha proporcionado uma melhoria no desempenho computacional em termos de tempo de execução, ela não exerceu um impacto na redução do número de instâncias, mantendo-se semelhante aos resultados obtidos pela abordagem sequencial.

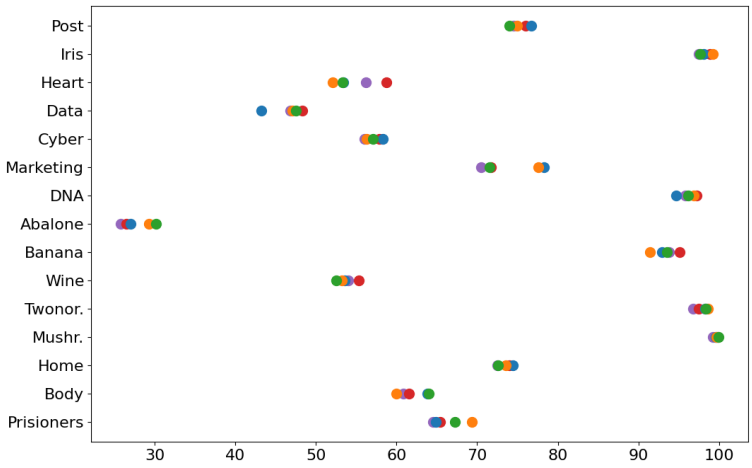
6.4 Resultados de predição

Para avaliar a eficácia das versões paralelizadas do algoritmo ACO na seleção das melhores instâncias, de forma comparável à versão sequencial, foram utilizadas as métricas *recall*, precisão e *F-Measure*, calculadas a partir dos algoritmos ANN, KNN, Random Forest e SVM.

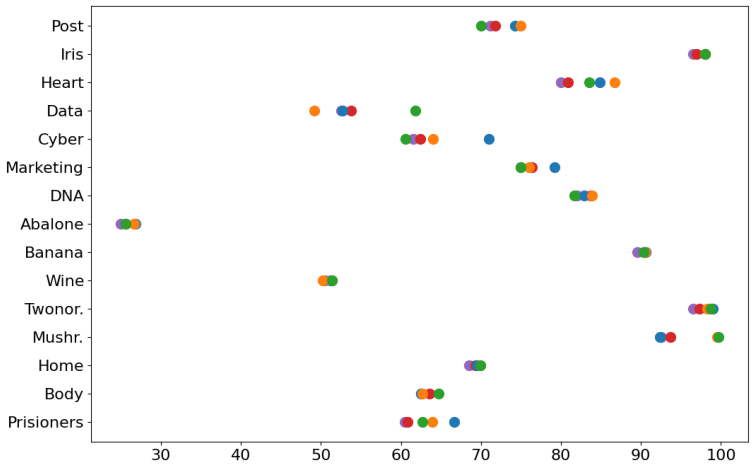
A Figura 11 apresenta os resultados da métrica *F-Measure*, adotada neste estudo por representar a média harmônica entre precision e *recall* em cada uma das bases de dados analisadas. A figura ilustra o desempenho das quatro versões do ACO aplicadas aos diferentes algoritmos de Aprendizado de Máquina, destacando tendências claras. Os

resultados completos das métricas F-Measure, precision e recall, bem como o desvio padrão de cada medida, estão disponíveis no Apêndice A, na página 96.

Figura 11: Resultados obtidos, considerando-se a métrica F-measure, a partir das bases pelos algoritmos de AM

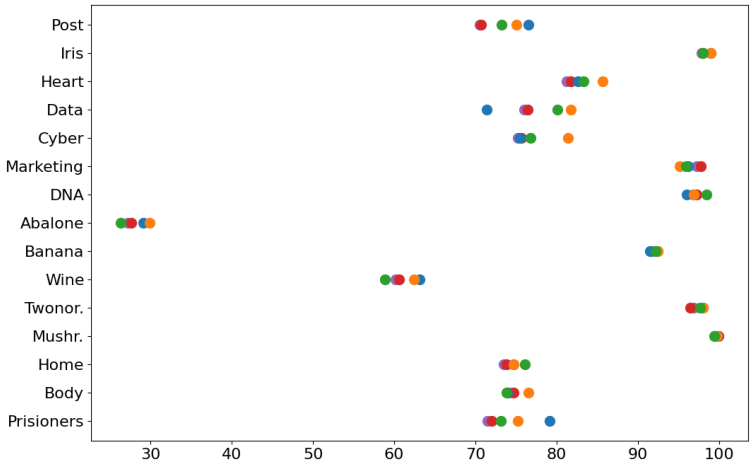


(a) Resultados das bases obtidos pelo ANN

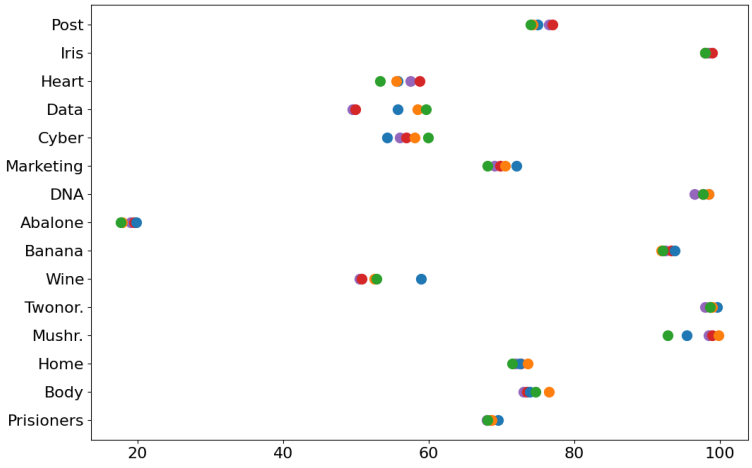


(b) Resultados das bases obtidos pelo KNN

Observa-se que as versões paralelizadas se equiparam à versão sequencial. Exemplificando, no algoritmo ANN, representado pela Figura 11(a), utilizando a base de dados 'Mushroom', os valores do *F-Measure* se sobrepõem entre as quatro versões. De maneira similar, no exemplo do KNN, mostrado na Figura 11(b), com a base de dados "Banana", também há sobreposição dos valores. Já na Figura 11(c), que apresenta o algoritmo *Random Forest*, algumas bases, como a "Isis", apresentam valores praticamente idênticos.



(c) Resultados das bases obtidos pelo *Random Forest*



(d) Resultados das bases obtidos pelo SVM

Fonte: Dados da Pesquisa

Esse comportamento também pode ser observado na Figura 11(d), referente ao algoritmo SVM, onde, por exemplo, a base 'Twonorm' exibe métricas com valores muito próximos.

Com base nas observações feitas nos exemplos de cada uma das subfiguras, é possível perceber que, independentemente do algoritmo de AM e da versão do ACO, seja paralelizada ou sequencial, os valores do *F-Measure* em ambos os algoritmos apresentam-se bastante semelhantes. Além disso, o comportamento das versões do ACO permanece bastante consistente entre os diferentes algoritmos de AM, evidenciando uma convergência nas métricas avaliadas.

Outra análise conduzida foi a comparação estatística entre as quatro versões do ACO e os algoritmos de AM, utilizando o teste T de *Student* para amostras independentes. Esse teste avalia se as melhorias observadas entre as versões do ACO são estatisticamente significativas ou meramente decorrentes de variações aleatórias. Vale destacar que foi adotado o nível de significância de $\alpha = 0.05$, amplamente empregado na literatura.

A Tabela 10 apresenta essa comparação, avaliando se cada versão alcançou desempenho equivalente à versão sequencial ou se há diferença de desempenho entre as versões para cada algoritmo de AM. Por exemplo, na linha "Seq. = Paralelo CPU", indica-se que, em determinadas bases, o desempenho do algoritmo de AM na versão sequencial foi equivalente ao da versão paralelizada na CPU. Já na linha "Seq. $\neq$ Paralelo CPU", significa que os desempenhos das versões sequencial e paralelizada na CPU foram distintos, ou seja, uma delas superou a outra. Os resultados detalhados da comparação entre as versões paralelizadas, indicando equivalência ou diferença de desempenho, podem ser encontrados na Tabela 14, no Apêndice B, página 107.

Tabela 10: Tabela Comparativa de Equivalência e Diferença entre Versões do ACO com o Teste T de *Student*. O número total de base de dados é 15.

	ANN	**KNN**	**RF**	**SVM**
Seq. = Paralelo CPU	7	4	6	11
Seq. ≠ Paralelo CPU	8	11	9	4
Seq. = Vetor. Paralelo CPU	8	13	10	12
Seq. ≠ Vetor. Paralelo CPU	7	2	5	3
Seq. = Paralelo GPU	6	9	8	14
Seq. ≠ Paralelo GPU	9	6	7	1

Fonte: Dados da Pesquisa

Os resultados da Tabela 10 mostram que o desempenho das versões sequencial e paralelizadas varia conforme o algoritmo de AM utilizado. Para o ANN, a equivalência entre a versão sequencial e a paralelizada na CPU ocorreu em 7 bases, enquanto a versão vetorizada na CPU apresentou equivalência em 8 bases. Já na GPU, a equivalência foi menor, observada em 6 bases. No KNN, a versão vetorizada na CPU indicou maior estabilidade, com equivalência em 13 bases, enquanto a versão paralelizada na GPU teve equivalência em 9 bases.

No caso do *Random Forest*, a versão vetorizada na CPU apresentou maior equivalência com a sequencial em 10 bases, enquanto a versão paralelizada na GPU foi equivalente em 8 bases. Para o SVM, a GPU apresentou o melhor desempenho, com equivalência em 14 bases e diferença em apenas 1. De modo geral, a vetorização na CPU mostrou maior estabilidade, enquanto a paralelização na GPU apresentou variações mais expressivas entre os AMs.

Complementando a Tabela 10, a Tabela 11 foi elaborada para evidenciar, entre os casos marcados como divergentes, quais algoritmos e bases de dados apresentaram melhor desempenho, indicando se a versão paralelizada ou sequencial foi superior, representada pelo símbolo “>”. Os resultados detalhados dessa análise entre as versões paralelizadas podem ser encontrados na Tabela 15, Apêndice B, página 107.

Tabela 11: Tabela Comparativa de Equivalência e Diferença entre Versões do ACO com o Teste T de *Student*

	ANN	**KNN**	**RF**	**SVM**
Seq. > Paralelo CPU	3	6	2	2
Paralelo CPU > Seq.	5	5	7	2
Seq. > Vetor. Paralelo CPU	4	1	3	0
Vetor. Paralelo CPU > Seq.	3	1	2	3
Seq. > Paralelo GPU	4	3	1	1
Paralelo GPU > Seq.	5	3	6	0

Fonte: Dados da Pesquisa

Os resultados observados indicam que a versão paralelizada do ACO, especialmente quando executada na CPU e GPU, tende a superar a versão sequencial em muitos casos, especialmente para os algoritmos ANN e *Random Forest*, onde a paralelização mostrou vantagens em maior número de bases. No entanto, para o KNN e SVM, as diferenças entre as versões foram mais equilibradas, com a versão sequencial apresentando desempenho superior em algumas bases. A versão vetorizada na CPU também apresentou ganhos de desempenho em relação à versão sequencial em algumas situações, embora com menor consistência quando comparada à paralelização na GPU.

A análise dos resultados estatísticos obtidos nas Tabelas 10 e 11 revela que a versão paralelizada do ACO, especialmente quando implementada na CPU e GPU, apresentou desempenho superior à versão sequencial na maioria dos casos, destacando-se especialmente nos algoritmos ANN e *Random Forest*. Para esses algoritmos, a paralelização mostrou uma vantagem consistente em um maior número de bases, especialmente quando executada na GPU. No entanto, no KNN e SVM, as diferenças entre as versões foram mais equilibradas, com a versão sequencial superando a paralelizada em algumas bases. A versão vetorizada na CPU, embora tenha mostrado bons resultados, apresentou menor consistência comparada à paralelização na GPU, sendo mais vantajosa em algumas situações e inferior em outras.

A conclusão desta subseção, que aborda os resultados das predições nas bases de dados, reforça as observações feitas nas análises estatísticas e na avaliação dos gráficos de dispersão, ambas indicando uma tendência clara no desempenho das versões paralelizadas do ACO. Nas análises estatísticas, observou-se que, na maioria dos casos, especialmente

para os algoritmos ANN e *Random Forest*, as versões paralelizadas — particularmente quando executadas na CPU e GPU — superaram a versão sequencial, com destaque para o desempenho na GPU. Nos algoritmos KNN e SVM, as diferenças entre as versões mostraram-se mais equilibradas, com algumas bases apresentando uma vantagem para a versão sequencial.

Por outro lado, os gráficos de dispersão evidenciam que, independentemente do algoritmo ou da versão do ACO, as métricas de *F-Measure* entre as versões sequenciais e paralelizadas apresentaram-se muito semelhantes, refletindo uma consistência no comportamento das versões, com resultados praticamente idênticos em várias bases de dados. Além disso, os resultados indicaram que, em muitas situações, as versões paralelizadas e sequenciais tiveram desempenhos praticamente equivalentes, especialmente em bases como as de *'Mushroom'* e *'Banana'*, nos algoritmos ANN e KNN. Isso sugere que, embora a paralelização ofereça vantagens de desempenho para determinados algoritmos e configurações, o comportamento das versões do ACO permanece estável e convergente, independentemente da abordagem adotada.

7 CONSIDERAÇÕES FINAIS

O objetivo central desta dissertação foi o desenvolvimento e a avaliação de três versões paralelizadas do algoritmo *Ant Colony Optimization* (ACO), com a finalidade de reduzir o tempo de execução, preservando a qualidade de predição comparável à versão sequencial do ACO. As versões paralelizadas envolvem a implementação na CPU utilizando paralelização clássica, a versão otimizada com CPU vetorizada e, finalmente, a paralelização via GPU. Para atingir esse propósito, foi conduzida uma análise comparativa entre essas abordagens, considerando o desempenho sob aspectos de tempo de execução, ganho de velocidade e eficiência computacional. Além disso, a investigação teve como foco assegurar que nenhuma das versões paralelizadas comprometesse a qualidade preditiva do algoritmo, mantendo os padrões de desempenho dos modelos de *AM* avaliados através das métricas *F-measure*, *Precision* e *Recall*. Essas métricas foram calculadas com base nos algoritmos de *AM*, incluindo *ANN*, *KNN*, *Random Forest* e *SVM*, garantindo que as versões paralelizadas não apenas preservassem a qualidade preditiva, mas também proporcionassem ganhos substanciais no desempenho computacional.

Os tempos de execução apresentaram reduções substanciais com a paralelização em todas as versões, quando comparadas à versão sequencial. A versão implementada na GPU obteve um desempenho de ganho superior, alcançando, em média, um ganho de 41,84 vezes. As duas versões paralelizadas na CPU também apresentaram melhorias notáveis: a versão vetorizada geralmente superou a versão apenas paralelizada, com um ganho médio de 17,69 vezes, em comparação com 16,63 vezes da versão CPU paralelizada.

A análise dos resultados apresentados nesta dissertação forneceu respostas claras tanto às questões de pesquisa quanto às hipóteses formuladas. A paralelização do algoritmo ACO nas arquiteturas de CPU, CPU vetorizada e GPU resultou em uma redução relevante no tempo de execução em comparação à versão sequencial, indicando que a paralelização é capaz de reduzir o tempo de execução enquanto mantém a qualidade dos modelos preditivos. As versões paralelizadas mantiveram a qualidade dos modelos de AM, evidenciada pelas métricas de AM, o que atesta que os ganhos de desempenho não comprometeram a qualidade da seleção de instâncias. A análise também revelou que a escolha da abordagem mais adequada depende de fatores como o tamanho da base de dados, os recursos computacionais disponíveis e o algoritmo de AM utilizado.

Em cenários com um grande número de instâncias, onde o custo computacional é mais alto, a paralelização tende a ser mais vantajosa. Para bases de dados com um nú-

mero reduzido de instâncias, como “Post Operative” e “Heart Failure”, a versão sequencial pode ser mais adequada, pois o ganho de desempenho com a paralelização é limitado e pode não justificar o custo adicional de recursos computacionais. No entanto, para bases de dados maiores, como “Twonorm” e “Mushrooms”, as versões paralelizadas na CPU e GPU oferecem ganhos notáveis em tempo de execução, sendo a GPU particularmente eficiente para tarefas que exigem alto poder de processamento. A versão vetorizada na CPU também se destaca em cenários com grandes volumes de dados, apresentando ganhos adicionais em algumas bases, como “Abalone” e “Marketing Analysis”, embora sua eficiência não seja uniformemente superior à paralelização tradicional.

Ao analisar os melhores resultados em cada versão dos algoritmos de AM, observa-se o seguinte: No caso do ANN, a versão paralelizada na GPU obteve os melhores resultados em bases como ‘Wine’ e ‘Body Performance’, com ganhos relevantes em *F-Measure* e Precision. A versão vetorizada na CPU teve desempenho competitivo em bases como ‘Banana’ e ‘Abalone’. Para o KNN, a versão vetorizada na CPU destacou-se em bases como ‘Mushrooms’ e ‘Home Equity’, com valores de *F-Measure* superiores, enquanto a versão sequencial se saiu bem em bases menores, como ‘Post Operative’. No caso do *Random Forest*, a versão paralelizada na CPU obteve os melhores resultados em bases como ‘Twonorm’ e ‘DNA’, com ganhos notáveis em *recall* e *precision*, e a versão em GPU também se destacou em bases como ‘Presos’ e ‘Body Performance’. Por fim, no SVM, a versão paralelizada na GPU apresentou os melhores resultados em praticamente todas as bases, com destaque para ‘Wine’ e ‘Mushrooms’, onde alcançou valores de *F-Measure* relevantes. A versão sequencial apresentou bom desempenho em bases menores, como ‘Iris’.

Em síntese, a decisão entre a versão sequencial e a paralelizada do ACO, assim como a escolha dos algoritmos de AM para cada versão, depende do tamanho da base de dados e dos recursos computacionais disponíveis. Para bases menores, a versão sequencial tende a ser mais eficiente, uma vez que o ganho com a paralelização é limitado e os custos com *hardware* podem ser desnecessários.

Em contrapartida, em cenários com grandes volumes de dados, as versões paralelizadas, especialmente na GPU, proporcionam ganhos notáveis em tempo de execução, sendo mais adequadas para tarefas que demandam alto poder de processamento. A versão vetorizada na CPU também pode ser vantajosa em algumas situações, embora nem sempre apresente vantagens claras sobre a paralelização tradicional. No que diz respeito aos algoritmos de AM, a versão paralelizada na GPU obteve os melhores resultados em métricas de desempenho, como *F-Measure* e *precision*, para ANN e SVM, enquanto a versão vetorizada na CPU se destacou no KNN. No *Random Forest*, a versão paralelizada na CPU apresentou os melhores resultados em *recall* e *precision*. Portanto, a escolha da versão ideal do ACO deve considerar o equilíbrio entre o tamanho da base de dados, o

tipo de algoritmo de AM e a infraestrutura computacional disponível.

Por fim, a implementação paralelizada do ACO em Python mostrou-se eficiente, mantendo a flexibilidade e portabilidade da linguagem, o que a torna uma solução viável para cenários de alto desempenho computacional, sem afetar a qualidade dos modelos preditivos.

7.1 Limitações e trabalhos futuros

Uma das principais limitações deste estudo reside no número restrito de bases de dados avaliadas até o momento, justificado pelo tempo considerável exigido para a execução na versão sequencial, conforme já comentado. Além disso, a comparação de desempenho entre núcleos da CPU e núcleos CUDA não foi contemplada, pois a medição dos núcleos CUDA não foi realizada neste estudo, ao contrário das versões de paralelização com CPU, nas quais os experimentos foram limitados pela quantidade de núcleos, variando de 2 a 18 núcleos. Outra limitação relevante é a ausência de referências para comparação, visto que este é o único estudo, até o momento, a implementar a paralelização do ACO. Da mesma forma, não foram encontrados artigos que explorem a paralelização de outros algoritmos de seleção de instâncias, o que impossibilita uma análise comparativa do desempenho e dos ganhos nas métricas de AM.

Como perspectivas para estudos futuros, sugere-se a ampliação do conjunto de bases de dados testadas, a exploração de uma maior variedade de algoritmos de AM para comparação de resultados e a implementação da versão paralelizada do ACO na GPU, analisando o desempenho em função da quantidade de SM, assim como foi feito na paralelização por núcleos da CPU.

8 PUBLICAÇÕES RELACIONADAS AO TRABALHO

A Tabela 12 apresenta os artigos publicados durante o desenvolvimento deste trabalho.

Tabela 12: Publicações relacionadas à pesquisa.

Nº	**Título**	**Periódico/Conferência**	**Ano**
1	Selection of Representative Instances using Ant Colony: A Case Study in a Database of Children and Adolescents with Attention-Deficit/Hyperactivity Disorder	HEALTHINF	2022
2	Avaliação das Técnicas Gulosa e Probabilística no Desempenho do Algoritmo de Otimização de Colônia de Formigas	SSCAD	2024
3	Avaliação de Desempenho e Escalabilidade do Algoritmo de Otimização de Colônia de Formigas em C++ e Python	SSCAD	2024
4	Selection of Representative Instances using Ant Colony Optimization: A Case Study in a Database of Newborns with Congenital Zika in Brazil	HEALTHINF	2025

REFERÊNCIAS

AKINYELU, A. A. Bio-inspired technique for improving machine learning speed and big data processing. In: 2020 International Joint Conference on Neural Networks (IJCNN). [S.l.: s.n.], 2020. p. 1–8.

AL-SHAFEI, A.; ZAREIPOUR, H.; CAO, Y. High-Performance and Parallel Computing Techniques Review: Applications, Challenges and Potentials to Support Net-Zero Transition of Future Grids. Energies, v. 15, n. 22, p. 1–58, November 2022. Disponível em: <https://ideas.repec.org/a/gam/jeners/v15y2022i22p8668-d977148.html>.

ALBAHRA, S. et al. Artificial intelligence and machine learning overview in pathology laboratory medicine: A general review of data preprocessing and basic supervised concepts. Seminars in Diagnostic Pathology, v. 40, n. 2, p. 71–87, 2023. ISSN 0740-2570. Artificial Intelligence (AI) , machine learning ML) and digital pathology integration are the next major chapter in our diagnostic pathology and laboratory medicine arena. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0740257023000138>.

ALEXANDROPOULOS, S.-A. N.; KOTSIANTIS, S. B.; VRAHATIS, M. N. Data preprocessing in predictive data mining. The Knowledge Engineering Review, v. 34, p. e1, 2019.

ALHENAWI, E. et al. Parallel ant colony optimization algorithm for finding the shortest path for mountain climbing. IEEE Access, v. 11, p. 6185–6196, 2023.

ANWAR, I. M.; SALAMA, K. M.; ABDELBAR, A. M. Adr-miner: An ant-based data reduction algorithm for classification. In: 2015 IEEE Congress on Evolutionary Computation (CEC). [S.l.: s.n.], 2015. p. 515–521.

ANWAR, I. M.; SALAMA, K. M.; ABDELBAR, A. M. Instance selection with ant colony optimization. Procedia Computer Science, 2015.

ARNAUTOVIć, M. et al. Parallelization of the ant colony optimization for the shortest path problem using openmp and cuda. In: 2013 36th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). [S.l.: s.n.], 2013. p. 1273–1277.

BADAWY, M.; RAMADAN, N.; HEFNY, H. Healthcare predictive analytics using machine learning and deep learning techniques: a survey. J. of Electric. Syst. and Inform. Techno., v. 10, 08 2023.

BAYDOGMUS, G. K. A parallelization based ant colony optimization for travelling salesman problem. In: 2022 1st International Conference on Information System & Information Technology (ICISIT). Yogyakarta, Indonesia: [s.n.], 2022. p. 166–169.

BENIN, K. R. d. A. Processamento de linguagem natural e a Ciência da Informação: inter-relações e contribuições. 2023. Dissertação (Mestrado) — Universidade Estadual de Londrina, dissertação (Mestado em Ciência da Informação).

BORIN, E. Técnicas para desenvolvimento e aceleração de códigos científicos. 2024. [Online; acessado em 13-Abril-2024]. Disponível em: <https://www.ic.unicamp.br/ edson/disciplinas/lncc14/lab-vetorizacao.html>.

CECILIA, J. M. et al. Enhancing data parallelism for ant colony optimization on gpus. Journal of Parallel and Distributed Computing, v. 73, p. 42–51, 2013. ISSN 0743-7315. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0743731512000032>.

CECILIA, J. M. et al. High-throughput ant colony optimization on graphics processing units. Journal of Parallel and Distributed Computing, v. 113, p. 261–274, 2018. ISSN 0743-7315. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0743731517303337>.

CEKMEZ, U.; OZSIGINAN, M.; SAHINGOZ, O. A uav path planning with parallel aco algorithm on cuda platform. In: . [S.l.: s.n.], 2014. p. 347–354. ISBN 978-1-4799-2376-2.

CENIKJ, G. et al. Selector: selecting a representative benchmark suite for reproducible statistical comparison. In: Proceedings of the Genetic and Evolutionary Computation Conference. ACM, 2022. p. 620–629. Disponível em: <http://dx.doi.org/10.1145/3512290.3528809>.

CHEN, L.; SUN, H.-Y.; WANG, S. Parallel implementation of ant colony optimization on mpp. In: Proceedings of the 7th International Conference on Machine Learning and Cybernetics, ICMLC. [S.l.: s.n.], 2008. v. 2, p. 981–986.

CHENG, F.; CHU, F.; ZHANG, L. A multi-objective evolutionary algorithm based on length reduction for large-scale instance selection. Information Sciences, v. 576, 06 2021.

COLORNI, A.; DORIGO, M.; MANIEZZO, V. Distributed optimization by ant colonies. In: Toward a practice of autonomous systems: proceedings of the First European conference on artificial life. MIT Press. [S.l.: s.n.], 1992. v. 134.

CORDEIRO, T. V. B. Predição de default de empresas: técnicas de machine learning em dados desbalanceados. 2020. Dissertação (Mestrado) — Não Informado pela instituição, dissertação (Mestrado). Disponível em: <https://hdl.handle.net/10438/29873>.

CUNHA, R. L. de F. et al. An argument in favor of strong scaling for deep neural networks with small datasets. CoRR, abs/1807.09161, 2018. Disponível em: <http://arxiv.org/abs/1807.09161>.

DAWSON, L.; STEWART, I. Candidate set parallelization strategies for ant colony optimization on the gpu. In: Lecture Notes in Computer Science. [S.l.: s.n.], 2013. p. 216–225.

DAWSON, L.; STEWART, I. A. Accelerating ant colony optimization-based edge detection on the gpu using cuda. In: 2014 IEEE Congress on Evolutionary Computation (CEC). [S.l.: s.n.], 2014. p. 1736–1743.

DELEVACQ, A. et al. Parallel ant colony optimization on graphics processing units. Journal of Parallel and Distributed Computing, v. 73, p. 52–61, 2013. ISSN 0743-7315. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0743731512000044>.

DELISLE, P.; KRAJECKI, M.; GRAVEL, M. Multi-colony parallel ant colony optimization on smp and multi-core computers. In: 2009 World Congress on Nature Biologically Inspired Computing (NaBIC). [S.l.: s.n.], 2009. p. 318–323.

DELISLE, P. et al. Parallel implementation of an ant colony optimization metaheuristic with openmp. 01 2001.

DELéVACQ, A. et al. Parallel ant colony optimization on graphics processing units. Journal of Parallel and Distributed Computing, v. 73, p. 52–61, 01 2013.

DONGDONG, G. et al. Application of multi-core parallel ant colony optimization in target assignment problem. In: 2010 International Conference on Computer Application and System Modeling (ICCASM 2010). Taiyuan: [s.n.], 2010. p. V3–514–V3–518.

DORIGO, M.; CARO, G. D. Ant colony optimization: a new meta-heuristic. In: Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406). [S.l.: s.n.], 1999. v. 2, p. 1470–1477 Vol. 2.

DORIGO, M.; GAMBARDELLA, L. Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation, v. 1, n. 1, p. 53–66, 1997.

DORIGO, M.; MANIEZZO, V.; COLORNI, A. Positive feedback as a search strategy. Dipartimento di Elettronica, Politecnico di Milano. [S.l.], 1991.

DORIGO, M.; MANIEZZO, V.; COLORNI, A. Ant system: optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), v. 26, n. 1, p. 29–41, 1996.

DOUGLAS, D. L. P.; LEVI, P. A new approach to exploiting parallelism in ant colony optimization. In: Proceedings of 2002 International Symposium on Micromechatronics and Human Science. [S.l.: s.n.], 2002. p. 237–243.

DZALBS, I.; KALGANOVA, T. Accelerating supply chains with ant colony optimization across a range of hardware solutions. Computers Industrial Engineering, v. 147, p. 106610, 2020. ISSN 0360-8352. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0360835220303442>.

ELSAID, A. et al. Optimizing long short-term memory recurrent neural networks using ant colony optimization to predict turbine engine vibration. Applied Soft Computing, v. 73, 2017.

ELSAID, A. et al. Using ant colony optimization to optimize long short-term memory recurrent neural networks. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '18). New York, NY, USA: Association for Computing Machinery, 2018. p. 13–20.

FANG, Y.; WU, L. B. Parallel ant colony algorithm for the logistics scheduling problem. In: 2010 International Conference on Multimedia Communications. [S.l.: s.n.], 2010. p. 116–119.

FANG, Y.; WU, L. B. Parallel ant colony algorithm for the logistics scheduling problem. In: 2010 International Conference on Multimedia Communications. [S.l.: s.n.], 2010. p. 116–119.

FLYNN, M. Very high-speed computing systems. Proceedings of the IEEE, v. 54, p. 1901 – 1909, 01 1966.

FLYNN, M. Some computer organizations and their effectiveness. ieee trans comput c-21:948. Computers, IEEE Transactions on, C-21, p. 948 – 960, 10 1972.

FU, J.; LEI, L.; ZHOU, G. A parallel ant colony optimization algorithm with gpu-acceleration based on all-in-roulette selection. In: Third International Workshop on Advanced Computational Intelligence. Suzhou, Jiangsu: [s.n.], 2010. p. 260–264.

GABR, M. I.; HELMY, Y. M.; ELZANFALY, D. S. Effect of missing data types and imputation methods on supervised classifiers: An evaluation study. Big Data and Cognitive Computing, v. 7, n. 1, 2023. ISSN 2504-2289. Disponível em: <https://www.mdpi.com/2504-2289/7/1/55>.

GAO, J. et al. Gpu implementation of ant colony optimization-based band selections for hyperspectral data classification. In: 2016 8th Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS). [S.l.: s.n.], 2016. p. 1–4.

GENTZSCH, W. Vectorization and parallelization techniques for modern supercomputers. In: _____. Supercomputers and Their Performance in Computational Fluid Dynamics. Wiesbaden: Vieweg+Teubner Verlag, 1993. p. 127–156. ISBN 978-3-322-87863-2. Disponível em: <https://doi.org/10.1007/978-3-322-87863-2$_7$>.

GONCALVES, A. et al. Avaliação das técnicas gulosa e probabilística no desempenho do algoritmo de otimização de colônia de formigas. In: Anais do XXV Simpósio em Sistemas Computacionais de Alto Desempenho. Porto Alegre, RS, Brasil: SBC, 2024. p. 1–12. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/sscad/article/view/30981>.

GUERRERO, G. et al. Comparative evaluation of platforms for parallel ant colony optimization. The Journal of Supercomputing, v. 69, p. 318–329, 2014.

HILL, K. Why is Python the Ideal Programming Language for AI and Data Science? 2020. Accessed: 2024-12-17. Disponível em: <https://towardsai.net/p/l/why-is-python-the-ideal-programming-language-for-ai-and-data-science?form=MG0AV3>.

HWU, W.-M. What is ahead for parallel computing. Journal of Parallel and Distributed Computing, v. 74, n. 7, p. 2574–2581, 2014. ISSN 0743-7315. Special Issue on Perspectives on Parallel and Distributed Processing. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0743731514000331>.

IBIAS, A. et al. Improving Noise Robustness through Abstractions and its Impact on Machine Learning. 2024. Disponível em: <https://arxiv.org/abs/2406.08428>.

INTEL. Hyper-Threading Technology. 2023. `https://www.intel.com.br/content/www/br/pt/support/articles/000100363/processors.html`. Acesso em: 1 de fevereiro de 2025.

JAIN, H. et al. Innovative approach for enhanced pattern extraction utilizing ant colony optimization. In: NAME, E. (Ed.). Title of the Book. [S.l.]: Taylor & Francis, 2023. cap. 56.

JIE, X.; CAIYUN, L.; ZHONG, C. A new parallel ant colony optimization algorithm based on message passing interface. In: 2008 IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application. [S.l.: s.n.], 2008. v. 2, p. 178–182.

JIENING, W.; JIANKANG, D.; CHUNFENG, Z. Implementation of ant colony algorithm based on gpu. In: 2009 Sixth International Conference on Computer Graphics, Imaging and Visualization. [S.l.: s.n.], 2009. p. 50–53.

KAMIRAN, F.; CALDERS, T. Data preprocessing techniques for classification without discrimination. Knowledge and Information Systems, Springer, v. 33, n. 1, p. 1–33, 2012. This article is an extended version of the papers. Disponível em: <https://doi.org/10.1007/s10115-011-0463-8>.

KEELE, S. et al. Guidelines for performing systematic literature reviews in software engineering. [S.l.], 2007.

LECUN, Y. B. . G. H. Y. Deep learning. Nature, Nature Publishing Group, v. 321, p. 436–444, 2015.

LI et al. Realization of parallel ant colony algorithm based on tbb multi-core platform. In: 2010 International Forum on Information Technology and Applications. Kunming, China: [s.n.], 2010. p. 177–180.

LIANG, W. et al. An enhanced ant colony optimization algorithm for global path planning of deep-sea mining vehicles. Ocean Engineering, v. 301, p. 117415, 2024. ISSN 0029-8018. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0029801824007522>.

LIN, S.; WU, J.; BHATTACHARYYA, S. S. Memory-constrained vectorization and scheduling of dataflow graphs for hybrid cpu-gpu platforms. ACM Trans. Embed. Comput. Syst., Association for Computing Machinery, New York, NY, USA, v. 17, n. 2, jan 2018.

LIU, H.; HE, Z. Parallel ant colony optimization algorithms for time series segmentation on a multi-core processor. In: 2012 4th International Conference on Intelligent Human-Machine Systems and Cybernetics. [S.l.: s.n.], 2012. v. 1, p. 340–343.

LIU, H.; HE, Z. Parallel ant colony optimization algorithms for time series segmentation on a multi-core processor. In: 2012 4th International Conference on Intelligent Human-Machine Systems and Cybernetics. [S.l.: s.n.], 2012. v. 1, p. 340–343.

LIU, H.; MOTODA, H. On issues of instance selection. Data Mining and Knowledge Discovery, Springer Nature BV, v. 6, n. 2, p. 115, 2002.

LIU, H.; MOTODA, H. Computational Methods of Feature Selection. [S.l.]: Chapman & Hall/CRC, 2008.

LLOYD, H.; AMOS, M. A highly parallelized and vectorized implementation of max-min ant system on intel® xeon phi™. In: 2016 IEEE Symposium Series on Computational Intelligence (SSCI). Athens, Greece: [s.n.], 2016. p. 1–6.

LUENGO, J. et al. Big Data Preprocessing: Enabling Smart Data. [S.l.: s.n.], 2020. ISBN 978-3-030-39104-1.

MAGALHãES, J. et al. Avaliação de desempenho e escalabilidade do algoritmo de otimização de colônia de formigas em c++ e python. In: Anais do XXV Simpósio em Sistemas Computacionais de Alto Desempenho. Porto Alegre, RS, Brasil: SBC, 2024. p. 97–108. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/sscad/article/view/30989>.

MANFRIN, M.; BIRATTARI, M.; STüTZLE, T. Parallel ant colony optimization for the traveling salesman problem. In: LNCS 4150: Proceedings of the 5th International Workshop on Ant Colony Optimization and Swarm Intelligence. [S.l.: s.n.], 2006. p. 221–234.

MARKVICA, D.; SCHAUER, C.; RAIDL, G. Cpu versus gpu parallelization of an ant colony optimization for the longest common subsequence problem. In: Lecture Notes in Computer Science. [S.l.: s.n.], 2015. p. 401–408.

MCCLANAHAN, C. History and evolution of gpu architecture a paper survey. In: . [s.n.], 2011. Disponível em: <https://api.semanticscholar.org/CorpusID:2337358>.

MEDEIROS, A. C. Análise de Desempenho do Algoritmo Colônia de Formigas para Seleção de Instâncias. 2022. Monografia.

MELO, G. E.; DUITAMA, F.; ARIAS, J. D. An Instance Selection Algorithm for Big Data in High imbalanced datasets based on LSH. 2022. Disponível em: <https://arxiv.org/abs/2210.04310>.

MENEZES, B. A. M. et al. Parallelization strategies for gpu-based ant colony optimization solving the traveling salesman problem. In: 2019 IEEE Congress on Evolutionary Computation (CEC). Wellington, New Zealand: [s.n.], 2019. p. 3094–3101.

MILOUD-AOUIDATE, A.; BABA-ALI, A. R. An efficient ant colony instance selection algorithm for knn classification. International Journal of Applied Metaheuristic Computing (IJAMC), 2013.

OLIVEIRA, B. V. N. d.; MELO, F. T. d. Fundamentos da visão computacional: Arcabouço teórico do reconhecimento artificial de imagens e vídeos. Humanidades & Inovação, v. 10, n. 17, 2023. Disponível em: <https://revista.unitins.br/index.php/humanidadeseinovacao/article/view/9078>.

OLVERA-LÓPEZ, J. A. et al. A review of instance selection methods. Artificial Intelligence Review, v. 34, n. 2, p. 133–143, Aug 2010. Disponível em: <https://doi.org/10.1007/s10462-010-9165-y>.

PAPADIMITRIOU, C. H. Combinatorial optimization. Algorithms and complexity, Prentice-Hall, New Jersey, 1982.

PAPENHAUSEN, E.; MUELLER, K. Coding ants – optimization of gpu code using ant colony optimization. Computer Languages, Systems Structures, v. 54, 2018.

PEAKE, J. et al. Vectorized candidate set selection for parallel ant colony optimization. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '18). New York, NY, USA: Association for Computing Machinery, 2018. p. 1300–1306.

PEDEMONTE, M.; NESMACHNOW, S.; CANCELA, H. A survey on parallel ant colony optimization. Applied Soft Computing, p. 5181–5197, 2011.

PENG, W. et al. A constrained ant colony algorithm for image registration. In: Lecture Notes in Computer Science. [S.l.: s.n.], 2006. v. 4115, p. 1–11.

POLYCHRONIOU, O.; RAGHAVAN, A.; ROSS, K. A. Rethinking SIMD vectorization for in-memory databases. In: . New York, NY, USA: Association for Computing Machinery, 2015.

RAO, K. M.; SAIKRISHNA, G.; SUPRIYA, K. Data preprocessing techniques: emergence and selection towards machine learning models - a practical review using hpa dataset. Multimedia Tools and Applications, v. 82, p. 1–20, 03 2023.

RUKMABHATLA, N. et al. Intel® Performance Hybrid Architecture & Software Optimizations, Part One: Introduction to Intel Performance Hybrid Architecture. [S.l.], 2021. Item Number: 349168. Disponível em: <https://www.intel.com>.

SAHA, S. et al. Cluster-oriented instance selection for classification problems. Information Sciences, v. 602, p. 143–158, 2022. ISSN 0020-0255.

SAILUNAZ, K. et al. A survey of machine learning-based methods for covid-19 medical image analysis. Med. & Biolog. Engineer. & Comput., v. 61, n. 6, p. 1257–1297, Jun 2023. ISSN 1741-0444.

SALAMA, K.; ABDELBAR, A.; ANWAR, I. Data reduction for classification with ant colony algorithms. Intelligent Data Analysis, v. 20, p. 1021–1059, 09 2016.

SATO, M. et al. First results of performance comparisons on many-core processors in solving qap with aco: kepler gpu versus xeon phi. In: Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO Comp '14). New York, NY, USA: Association for Computing Machinery, 2014. p. 1477–1478.

SCHMIDHUBER, J. Deep learning in neural networks: An overview. Neural networks, Elsevier, v. 61, p. 85–117, 2015.

SILVA, A. C. da et al. A review of the main factors, computational methods, and databases used in depression studies. In: HEALTHINF. [S.l.: s.n.], 2022. p. 413–420.

SILVA, E. et al. O uso da inteligência artificial na doença de parkinson. Research, Society and Development, v. 14, p. e7214148011, 01 2025.

SINNOTT-ARMSTRONG, N. A.; GREENE, C. S.; MOORE, J. H. Fast genome-wide epistasis analysis using ant colony optimization for multifactor dimensionality reduction analysis on graphics processing units. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation (GECCO '10). New York, NY, USA: Association for Computing Machinery, 2010. p. 215–216.

SKINDEROWICZ, R. The gpu-based parallel ant colony system. Journal of Parallel and Distributed Computing, v. 98, p. 48–60, 2016. ISSN 0743-7315. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0743731516300284>.

SOARES, F. A. L.; NOBRE, C. N.; FREITAS, H. Cota de. Parallel programming in computing undergraduate courses: a systematic mapping of the literature. IEEE Latin America Transactions, v. 17, n. 08, p. 1371–1381, 2019.

STATISTA. Python - Statistics & Facts. 2024. Accessed: 2024-12-17. Disponível em: <https://www.statista.com/topics/9361/python/?form=MG0AV3topicOverview>.

STEIN, C. M. Programação Paralela para GPU em Aplicações de Processamento de Stream. 2018. Dissertação (Mestrado) — Sociedade Educacional Três de Maio - SETREM, Três de Maio, Brasil, trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação).

STÜTZLE, T.; HOOS, H. Improvements on the ant-system: Introducing the max-min ant system. In: Artificial Neural Nets and Genetic Algorithms. Vienna: Springer Vienna, 1998. p. 245–249. ISBN 978-3-7091-6492-1.

TAE, I. J.; BROILLET-SCHLESINGER, A.; KIM, B. Y. Selection attributes of integrated mobility apps on affecting users' intention to use: A case of republic of korea. Future Transportation, v. 4, n. 4, p. 1205–1222, 2024. ISSN 2673-7590. Disponível em: <https://www.mdpi.com/2673-7590/4/4/58>.

TIWARI, A. K. et al. A novel intuitionistic fuzzy rough instance selection and attribute reduction with kernelized intuitionistic fuzzy c-means clustering to handle imbalanced datasets. Expert Systems with Applications, v. 251, p. 124087, 2024. ISSN 0957-4174. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0957417424009539>.

TSAI, C.-F.; et al. Combining feature selection, instance selection, and ensemble classification techniques for improved financial distress prediction. J. Bus. Res., 2021.

TSUTSUI, S.; COLLET, P. Massively Parallel Evolutionary Computation on GPGPUs. [S.l.: s.n.], 2013. ISBN 978-3-642-37959-8.

TSUTSUI, S.; FUJIMOTO, N. Parallel ant colony optimization algorithm on a multi-core processor. In: Proceedings of the 5th International Workshop on Ant Colony Optimization and Swarm Intelligence. [S.l.: s.n.], 2010. p. 488–495.

TSUTSUI, S.; FUJIMOTO, N. A comparative study of synchronization of parallel aco on multi-core processor. In: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO Companion '15). New York, NY, USA: Association for Computing Machinery, 2015. p. 777–778.

TUFTELAND, T.; ØDESNELTVEDT, G.; GOODWIN, M. Optimizing polyaco training with gpu-based parallelization. In: Lecture Notes in Computer Science. [S.l.: s.n.], 2016. p. 233–240.

TURNER, H.; WHITE, J. Multi-core deployment optimization using simulated annealing and ant colony optimization. In: 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications. Melbourne, VIC, Australia: [s.n.], 2013. p. 1216–1223.

VEDANA, A. B. et al. InteligÊncia artificial na medicina diagnÓstica. Brazilian Journal of Implantology and Health Sciences, v. 6, n. 11, p. 765–794, nov. 2024. Disponível em: <https://bjihs.emnuvens.com.br/bjihs/article/view/4235>.

VIANA, L. H. M. Abordagem e representação gráfica para visualização de hierarquia entre regiões e desequilíbrio de carga de processamento em programas paralelos. Julho 2022. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, Natal, RN, trabalho de Conclusão de Curso de Engenharia de Computação. Disponível em: <https://pascalsuite.imd.ufrn.br>.

WANG, S. Research on grid planning of dual power distribution network based on parallel ant colony optimization algorithm. Journal of Electronic Research and Application, v. 7, p. 32–41, 2023.

WEI, K.-C.; WU, C.-C.; WU, C.-J. Using cuda gpu to accelerate the ant colony optimization algorithm. In: 2013 International Conference on Parallel and Distributed Computing, Applications and Technologies. [S.l.: s.n.], 2013. p. 90–95.

XAVIER, J. et al. Vectorized highly parallel density-based clustering for applications with noise. IEEE Access, PP, p. 1–1, 01 2024.

YANG, Q.; FANG, L.; DUAN, X. Rmaco: a randomly matched parallel ant colony optimization. World Wide Web, v. 19, 2016.

YOUNESS, H. et al. An efficient implementation of ant colony optimization on gpu for the satisfiability problem. In: 2015 23rd Euromicro International Conference

on Parallel, Distributed, and Network-Based Processing. [S.l.: s.n.], 2015. p. 230–235.

ZENG, Z. et al. A study of parallel ant colony optimization with cuda-based implementation. In: 2023 IEEE 6th International Conference on Electronic Information and Communication Technology (ICEICT). [S.l.: s.n.], 2023. p. 869–871.

ZHAI, J.; QI, J.; ZHANG, S. An instance selection algorithm for fuzzy k-nearest neighbor. J. Intell. Fuzzy Syst., IOS Press, NLD, v. 40, n. 1, p. 521–533, jan. 2021. ISSN 1064-1246. Disponível em: <https://doi.org/10.3233/JIFS-200124>.

ZHOU, Y. et al. Parallel ant colony optimization on multi-core simd cpus. Future Generation Computer Systems, v. 79, p. 473–487, 2018. ISSN 0167-739X.

ZHOU, Y.; HE, F.; QIU, Y. Dynamic strategy based parallel ant colony optimization on gpus for tsps. Science China Information Sciences, v. 60, 2017.

ZHOU, Y. et al. Adaptive gradient descent enabled ant colony optimization for routing problems. Swarm and Evolutionary Computation, v. 70, 2022.

ZHU, W.; CURRY, J. Parallel ant colony for nonlinear function optimization with graphics hardware acceleration. In: 2009 IEEE International Conference on Systems, Man and Cybernetics. [S.l.: s.n.], 2009. p. 1803–1808.

APÊNDICE A – DESEMPENHO PREDITIVO DOS MODELOS DAS BASES PÚBLICAS

A Tabela 13 apresenta os valores das três métricas *F-Measure*, *Precision* e *Recall* dos algoritmos de AM ANN, KNN, *Random Forest* e SVM para cada uma das quatro versões do ACO. Além disso, são apresentados os respectivos valores de desvio padrão para cada métrica.

Tabela 13 – Valores de predição dos algoritmos de AM nas Bases de Dados em %. Os valores entre colchetes representam os valores do desvio padrão.

Base de Dados	Versão do ACO	Algoritmo	*F-Measure*	*Precision*	*Recall*
Post Operation	Sequencial	ANN	76.0	77.4	74.6
			[1.2]	[1.1]	[1.3]
	Paralelo CPU	ANN	76.7	78.0	75.4
			[1.1]	[1.0]	[1.2]
	Vetorizado CPU	ANN	75.9	76.3	73.9
			[1.3]	[1.2]	[1.4]
	GPU	ANN	74.1	74.6	73.4
			[1.4]	[1.3]	[1.5]
	Sequencial	k-NN	71.8	72.5	71.0
			[1.5]	[1.4]	[1.6]
	Paralelo CPU	k-NN	74.3	74.9	73.6
			[1.3]	[1.2]	[1.4]
	Vetorizado CPU	k-NN	75.0	75.1	73.9
			[1.2]	[1.1]	[1.3]
	GPU	k-NN	70.0	70.5	69.7
			[1.6]	[1.5]	[1.7]
	Sequencial	*Random Forest*	70.7	71.2	70.1
			[1.4]	[1.3]	[1.5]
	Paralelo CPU	*Random Forest*	76.5	77.0	76.0
			[1.1]	[1.0]	[1.2]
	Vetorizado CPU	*Random Forest*	75.0	75.6	74.4
			[1.2]	[1.1]	[1.3]
	GPU	*Random Forest*	73.3	73.8	72.6
			[1.3]	[1.2]	[1.4]
	Sequencial	SVM	77.0	80.9	73.5
			[1.1]	[1.0]	[1.2]
	Paralelo CPU	SVM	75.0	78.8	71.6

Base de Dados	**Versão do ACO**	**Algoritmo**	***F-Measure***	***Precision***	***Recall***
			[1.2]	[1.1]	[1.3]
	Vetorizado CPU	SVM	74.3	78.0	70.9
			[1.3]	[1.2]	[1.4]
	GPU	SVM	74.0	77.7	70.6
			[1.4]	[1.3]	[1.5]
Iris	Sequencial	ANN	98.9	99.1	98.7
			[0.5]	[0.4]	[0.6]
	Paralelo CPU	ANN	98.1	98.7	97.5
			[0.6]	[0.5]	[0.7]
	Vetorizado CPU	ANN	99.2	99.0	99.4
			[0.4]	[0.3]	[0.5]
	GPU	ANN	97.6	98.1	97.2
			[0.7]	[0.6]	[0.8]
	Sequencial	k-NN	97.0	97.5	96.4
			[0.8]	[0.7]	[0.9]
	Paralelo CPU	k-NN	98.0	98.6	97.9
			[0.7]	[0.6]	[0.8]
	Vetorizado CPU	k-NN	98.0	98.2	97.6
			[0.7]	[0.6]	[0.8]
	GPU	k-NN	98.0	97.9	97.1
			[0.8]	[0.7]	[0.9]
	Sequencial	*Random Forest*	98.0	98.5	97.6
			[0.7]	[0.6]	[0.8]
	Paralelo CPU	*Random Forest*	99.0	99.1	98.3
			[0.5]	[0.4]	[0.6]
	Vetorizado CPU	*Random Forest*	99.0	99.1	98.2
			[0.5]	[0.4]	[0.6]
	GPU	*Random Forest*	98.0	98.4	97.5
			[0.7]	[0.6]	[0.8]
	Sequencial	SVM	99.0	100.0	94.5
			[0.5]	[0.0]	[1.0]
	Paralelo CPU	SVM	98.0	100.0	93.6
			[0.6]	[0.0]	[1.1]
	Vetorizado CPU	SVM	98.0	100.0	93.6
			[0.6]	[0.0]	[1.1]
	GPU	SVM	98.0	100.0	93.6
			[0.7]	[0.0]	[1.2]
Heart Failure	Sequencial	ANN	58.7	59.9	57.6
			[1.8]	[1.7]	[1.9]
	Paralelo CPU	ANN	53.4	54.9	51.9
			[2.0]	[1.9]	[2.1]
	Vetorizado CPU	ANN	52.1	53.8	50.4
			[2.1]	[2.0]	[2.2]
	GPU	ANN	53.3	54.1	52.6
			[2.0]	[1.9]	[2.1]
	Sequencial	k-NN	80.9	81.4	80.2

Base de Dados	**Versão do ACO**	**Algoritmo**	***F-Measure***	***Precision***	***Recall***
			[1.3]	[1.2]	[1.4]
	Paralelo CPU	k-NN	84.8	85.3	84.0
			[1.1]	[1.0]	[1.2]
	Vetorizado CPU	k-NN	86.7	87.1	86.2
			[1.0]	[0.9]	[1.1]
	GPU	k-NN	83.6	84.7	83.0
			[1.2]	[1.1]	[1.3]
	Sequencial	*Random Forest*	81.7	82.1	81.3
			[1.2]	[1.1]	[1.3]
	Paralelo CPU	*Random Forest*	82.6	83.1	82.0
			[1.1]	[1.0]	[1.2]
	Vetorizado CPU	*Random Forest*	85.7	86.5	85.1
			[1.0]	[0.9]	[1.1]
	GPU	*Random Forest*	83.3	84.1	83.0
			[1.2]	[1.1]	[1.3]
	Sequencial	SVM	58.7	61.7	56.1
			[1.8]	[1.7]	[1.9]
	Paralelo CPU	SVM	55.7	58.5	53.2
			[2.0]	[1.9]	[2.1]
	Vetorizado CPU	SVM	55.6	58.3	53.0
			[2.0]	[1.9]	[2.1]
	GPU	SVM	53.3	56.0	50.9
			[2.1]	[2.0]	[2.2]
Data Science	Sequencial	ANN	48.3	49.3	47.4
			[2.2]	[2.1]	[2.3]
	Paralelo CPU	ANN	43.3	44.8	41.8
			[2.4]	[2.3]	[2.5]
	Vetorizado CPU	ANN	47.1	48.2	46.1
			[2.2]	[2.1]	[2.3]
	GPU	ANN	47.5	48.0	47.1
			[2.2]	[2.1]	[2.3]
	Sequencial	k-NN	53.7	54.3	52.9
			[2.0]	[1.9]	[2.1]
	Paralelo CPU	k-NN	52.7	53.1	51.9
			[2.1]	[2.0]	[2.2]
	Vetorizado CPU	k-NN	49.2	49.9	48.6
			[2.3]	[2.2]	[2.4]
	GPU	k-NN	61.8	62.5	60.9
			[1.7]	[1.6]	[1.8]
	Sequencial	*Random Forest*	76.4	76.9	75.9
			[1.3]	[1.2]	[1.4]
	Paralelo CPU	*Random Forest*	71.4	72.5	70.9
			[1.6]	[1.5]	[1.7]
	Vetorizado CPU	*Random Forest*	81.7	82.7	80.9
			[1.1]	[1.0]	[1.2]
	GPU	*Random Forest*	80.1	81.1	79.3

Base de Dados	**Versão do ACO**	**Algoritmo**	***F-Measure***	***Precision***	***Recall***
			[1.2]	[1.1]	[1.3]
	Sequencial	SVM	49.9	50.9	47.6
			[2.2]	[2.1]	[2.3]
	Paralelo CPU	SVM	55.8	58.6	53.2
			[2.0]	[1.9]	[2.1]
	Vetorizado CPU	SVM	58.5	61.4	55.8
			[1.9]	[1.8]	[2.0]
	GPU	SVM	59.6	62.6	56.9
			[1.8]	[1.7]	[1.9]
Cyber Salarie	Sequencial	ANN	57.9	58.9	56.8
			[1.9]	[1.8]	[2.0]
	Paralelo CPU	ANN	58.3	59.5	57.2
			[1.9]	[1.8]	[2.0]
	Vetorizado CPU	ANN	56.3	57.0	55.7
			[2.0]	[1.9]	[2.1]
	GPU	ANN	57.1	58.2	56.1
			[1.9]	[1.8]	[2.0]
	Sequencial	k-NN	62.4	63.1	61.8
			[1.8]	[1.7]	[1.9]
	Paralelo CPU	k-NN	71.0	71.9	70.3
			[1.6]	[1.5]	[1.7]
	Vetorizado CPU	k-NN	64.0	64.0	62.5
			[1.7]	[1.6]	[1.8]
	GPU	k-NN	60.6	61.0	59.8
			[2.0]	[1.9]	[2.1]
	Sequencial	*Random Forest*	75.7	76.2	74.9
			[1.6]	[1.5]	[1.7]
	Paralelo CPU	*Random Forest*	75.5	75.9	74.1
			[1.7]	[1.6]	[1.8]
	Vetorizado CPU	*Random Forest*	81.4	81.9	80.1
			[1.8]	[1.7]	[1.9]
	GPU	*Random Forest*	76.8	77.5	76.2
			[2.0]	[1.9]	[2.1]
	Sequencial	SVM	57.0	59.8	54.4
			[1.8]	[1.7]	[1.9]
	Paralelo CPU	SVM	54.3	57.0	51.8
			[1.7]	[1.6]	[1.8]
	Vetorizado CPU	SVM	58.1	61.0	55.5
			[1.8]	[1.7]	[1.9]
	GPU	SVM	60.0	63.0	57.2
			[1.9]	[1.8]	[2.0]
	Sequencial	ANN	71.7	72.9	70.6
			[1.5]	[1.4]	[1.6]
	Paralelo CPU	ANN	78.3	79.5	77.1
			[1.3]	[1.2]	[1.4]
	Vetorizado CPU	ANN	77.6	78.3	76.9

Base de Dados	**Versão do ACO**	**Algoritmo**	***F-Measure***	***Precision***	***Recall***
			[1.3]	[1.2]	[1.4]
	GPU	ANN	71.5	72.1	71.0
			[1.5]	[1.4]	[1.6]
	Sequencial	k-NN	76.3	77.5	76.0
			[1.4]	[1.3]	[1.5]
Marketing Analysis	Paralelo CPU	k-NN	79.2	80.2	78.3
			[1.2]	[1.1]	[1.3]
	Vetorizado CPU	k-NN	76.0	76.4	74.9
			[1.4]	[1.3]	[1.5]
	GPU	k-NN	75.0	75.3	73.9
			[1.5]	[1.4]	[1.6]
	Sequencial	*Random Forest*	97.8	97.9	96.8
			[0.6]	[0.5]	[0.7]
	Paralelo CPU	*Random Forest*	96.2	96.9	95.4
			[0.7]	[0.6]	[0.8]
	Vetorizado CPU	*Random Forest*	95.1	95.5	94.2
			[0.8]	[0.7]	[0.9]
	GPU	*Random Forest*	95.9	96.1	95.4
			[0.7]	[0.6]	[0.8]
	Sequencial	SVM	69.9	73.4	66.7
			[1.6]	[1.5]	[1.7]
	Paralelo CPU	SVM	72.1	75.7	68.8
			[1.5]	[1.4]	[1.6]
	Vetorizado CPU	SVM	70.5	74.0	67.3
			[1.6]	[1.5]	[1.7]
	GPU	SVM	68.1	71.5	65.0
			[1.7]	[1.6]	[1.8]
	Sequencial	ANN	97.2	97.9	96.6
			[0.6]	[0.5]	[0.7]
	Paralelo CPU	ANN	94.6	95.4	93.9
			[0.8]	[0.7]	[0.9]
	Vetorizado CPU	ANN	96.9	97.0	96.8
			[0.6]	[0.5]	[0.7]
	GPU	ANN	96.2	96.5	95.9
			[0.7]	[0.6]	[0.8]
	Sequencial	k-NN	83.7	84.3	83.2
			[1.2]	[1.1]	[1.3]
DNA	Paralelo CPU	k-NN	82.9	83.4	82.4
			[1.3]	[1.2]	[1.4]
	Vetorizado CPU	k-NN	83.9	84.7	83.5
			[1.2]	[1.1]	[1.3]
	GPU	k-NN	81.7	82.1	81.0
			[1.3]	[1.2]	[1.4]
	Sequencial	*Random Forest*	97.3	97.5	96.9
			[0.6]	[0.5]	[0.7]
	Paralelo CPU	*Random Forest*	96.0	96.6	95.5

Base de Dados	**Versão do ACO**	**Algoritmo**	***F-Measure***	***Precision***	***Recall***
			[0.7]	[0.6]	[0.8]
	Vetorizado CPU	*Random Forest*	96.8	97.1	96.5
			[0.6]	[0.5]	[0.7]
	GPU	*Random Forest*	98.5	98.3	97.1
			[0.5]	[0.4]	[0.6]
	Sequencial	SVM	97.7	100.0	93.2
			[0.6]	[0.0]	[1.0]
	Paralelo CPU	SVM	98.5	100.0	94.0
			[0.5]	[0.0]	[1.0]
	Vetorizado CPU	SVM	98.4	100.0	94.0
			[0.5]	[0.0]	[1.0]
	GPU	SVM	97.7	100.0	93.3
			[0.6]	[0.0]	[1.0]
Abalone	Sequencial	ANN	26.5	27.7	25.3
			[2.5]	[2.4]	[2.6]
	Paralelo CPU	ANN	27.0	28.3	25.9
			[2.5]	[2.4]	[2.6]
	Vetorizado CPU	ANN	29.2	30.0	28.5
			[2.4]	[2.3]	[2.5]
	GPU	ANN	30.1	31.0	29.3
			[2.4]	[2.3]	[2.5]
	Sequencial	k-NN	25.6	26.1	25.1
			[2.5]	[2.4]	[2.6]
	Paralelo CPU	k-NN	26.8	27.3	26.4
			[2.5]	[2.4]	[2.6]
	Vetorizado CPU	k-NN	26.6	26.8	25.9
			[2.5]	[2.4]	[2.6]
	GPU	k-NN	25.6	25.8	24.9
			[2.5]	[2.4]	[2.6]
	Sequencial	*Random Forest*	27.6	28.1	27.1
			[2.5]	[2.4]	[2.6]
	Paralelo CPU	*Random Forest*	29.1	29.5	28.3
			[2.4]	[2.3]	[2.5]
	Vetorizado CPU	*Random Forest*	29.9	30.2	29.0
			[2.4]	[2.3]	[2.5]
	GPU	*Random Forest*	26.3	27.6	26.1
			[2.5]	[2.4]	[2.6]
	Sequencial	SVM	19.5	20.5	18.6
			[2.6]	[2.5]	[2.7]
	Paralelo CPU	SVM	19.8	20.8	18.9
			[2.6]	[2.5]	[2.7]
	Vetorizado CPU	SVM	17.9	18.8	17.1
			[2.7]	[2.6]	[2.8]
	GPU	SVM	17.7	18.6	16.9
			[2.7]	[2.6]	[2.8]
	Sequencial	ANN	95.2	96.0	94.3

Base de Dados	**Versão do ACO**	**Algoritmo**	***F-Measure***	***Precision***	***Recall***
			[0.7]	[0.6]	[0.8]
	Paralelo CPU	ANN	92.9	93.7	92.1
			[0.9]	[0.8]	[1.0]
	Vetorizado CPU	ANN	91.5	92.5	90.7
			[1.0]	[0.9]	[1.1]
	GPU	ANN	93.5	94.2	92.9
			[0.8]	[0.7]	[0.9]
	Sequencial	k-NN	90.6	91.5	89.8
			[1.0]	[0.9]	[1.1]
Banana	Paralelo CPU	k-NN	90.6	91.4	90.0
			[1.0]	[0.9]	[1.1]
	Vetorizado CPU	k-NN	90.5	90.9	89.6
			[1.0]	[0.9]	[1.1]
	GPU	k-NN	90.3	90.5	89.0
			[1.0]	[0.9]	[1.1]
	Sequencial	*Random Forest*	92.4	92.9	91.9
			[0.9]	[0.8]	[1.0]
	Paralelo CPU	*Random Forest*	91.5	91.9	90.5
			[1.0]	[0.9]	[1.1]
	Vetorizado CPU	*Random Forest*	92.4	92.5	91.1
			[0.9]	[0.8]	[1.0]
	GPU	*Random Forest*	92.2	92.1	90.8
			[0.9]	[0.8]	[1.0]
	Sequencial	SVM	93.4	98.0	89.1
			[0.8]	[0.5]	[1.0]
	Paralelo CPU	SVM	93.8	98.5	89.6
			[0.8]	[0.5]	[1.0]
	Vetorizado CPU	SVM	92.0	96.6	87.8
			[0.9]	[0.6]	[1.1]
	GPU	SVM	92.1	96.7	87.9
			[0.9]	[0.6]	[1.1]
	Sequencial	ANN	55.3	56.2	54.5
			[2.0]	[1.9]	[2.1]
	Paralelo CPU	ANN	53.5	54.0	52.1
			[2.1]	[2.0]	[2.2]
	Vetorizado CPU	ANN	53.2	53.6	51.9
			[2.1]	[2.0]	[2.2]
	GPU	ANN	52.5	52.9	51.2
			[2.1]	[2.0]	[2.2]
	Sequencial	k-NN	51.4	52.5	51.0
			[2.2]	[2.1]	[2.3]
Wine	Paralelo CPU	k-NN	51.2	51.8	50.1
			[2.2]	[2.1]	[2.3]
	Vetorizado CPU	k-NN	50.2	50.6	49.4
			[2.3]	[2.2]	[2.4]
	GPU	k-NN	51.3	51.9	50.3

Base de Dados	**Versão do ACO**	**Algoritmo**	***F-Measure***	***Precision***	***Recall***
			[2.2]	[2.1]	[2.3]
	Sequencial	*Random Forest*	60.6	61.2	60.0
			[1.9]	[1.8]	[2.0]
	Paralelo CPU	*Random Forest*	63.1	63.9	62.1
			[1.8]	[1.7]	[1.9]
	Vetorizado CPU	*Random Forest*	62.4	62.5	61.1
			[1.8]	[1.7]	[1.9]
	GPU	*Random Forest*	58.9	59.1	58.0
			[2.0]	[1.9]	[2.1]
	Sequencial	SVM	50.8	53.4	48.5
			[2.2]	[2.1]	[2.3]
	Paralelo CPU	SVM	59.0	61.9	56.3
			[2.0]	[1.9]	[2.1]
	Vetorizado CPU	SVM	52.5	55.1	50.1
			[2.2]	[2.1]	[2.3]
	GPU	SVM	52.9	55.5	50.5
			[2.2]	[2.1]	[2.3]
Twonorm	Sequencial	ANN	97.5	98.5	96.5
			[0.6]	[0.5]	[0.7]
	Paralelo CPU	ANN	98.3	99.1	97.6
			[0.5]	[0.4]	[0.6]
	Vetorizado CPU	ANN	98.7	99.0	97.9
			[0.5]	[0.4]	[0.6]
	GPU	ANN	98.4	98.3	96.9
			[0.5]	[0.4]	[0.6]
	Sequencial	k-NN	97.3	97.9	96.8
			[0.6]	[0.5]	[0.7]
	Paralelo CPU	k-NN	99.0	99.3	98.1
			[0.6]	[0.5]	[0.7]
	Vetorizado CPU	k-NN	98.3	98.6	97.4
			[0.6]	[0.5]	[0.7]
	GPU	k-NN	98.7	98.9	97.6
			[0.6]	[0.5]	[0.7]
	Sequencial	*Random Forest*	96.5	96.9	95.8
			[0.6]	[0.5]	[0.7]
	Paralelo CPU	*Random Forest*	98.0	95.8	97.5
			[0.6]	[0.5]	[0.7]
	Vetorizado CPU	*Random Forest*	98.0	98.1	96.9
			[0.6]	[0.5]	[0.7]
	GPU	*Random Forest*	97.7	97.5	96.2
			[0.6]	[0.5]	[0.7]
	Sequencial	SVM	98.7	100.0	94.2
			[0.6]	[0.5]	[0.7]
	Paralelo CPU	SVM	99.7	100.0	95.2
			[0.6]	[0.5]	[0.7]
	Vetorizado CPU	SVM	99.0	100.0	94.5

Base de Dados	**Versão do ACO**	**Algoritmo**	***F-Measure***	***Precision***	***Recall***
			[0.6]	[0.5]	[0.7]
	GPU	SVM	98.7	100.0	94.2
			[0.6]	[0.5]	[0.7]
Mushrooms	Sequencial	ANN	99.9	99.7	99.1
			[0.7]	[0.6]	[0.8]
	Paralelo CPU	ANN	99.6	99.6	99.0
			[0.7]	[0.6]	[0.8]
	Vetorizado CPU	ANN	99.6	99.6	99.1
			[0.7]	[0.6]	[0.8]
	GPU	ANN	99.9	99.8	99.1
			[0.7]	[0.6]	[0.8]
	Sequencial	k-NN	93.7	94.2	92.9
			[0.9]	[0.8]	[1.0]
	Paralelo CPU	k-NN	92.3	92.5	91.0
			[1.0]	[0.9]	[1.1]
	Vetorizado CPU	k-NN	99.5	91.0	99.1
			[0.4]	[1.0]	[0.5]
	GPU	k-NN	99.7	99.4	98.6
			[0.3]	[0.4]	[0.5]
	Sequencial	*Random Forest*	99.9	99.7	99.1
			[0.3]	[0.4]	[0.5]
	Paralelo CPU	*Random Forest*	99.9	99.4	98.9
			[0.3]	[0.4]	[0.5]
	Vetorizado CPU	*Random Forest*	99.8	99.4	98.8
			[0.3]	[0.4]	[0.5]
	GPU	*Random Forest*	99.4	99.1	98.3
			[0.4]	[0.4]	[0.5]
	Sequencial	SVM	99.0	100.0	94.5
			[0.5]	[0.0]	[1.0]
	Paralelo CPU	SVM	95.5	100.0	91.1
			[0.8]	[0.0]	[1.1]
	Vetorizado CPU	SVM	99.8	100.0	95.3
			[0.3]	[0.0]	[1.0]
	GPU	SVM	92.9	97.5	88.6
			[1.0]	[0.6]	[1.2]
Home Equity	Sequencial	ANN	74.0	75.1	72.9
			[1.5]	[1.4]	[1.6]
	Paralelo CPU	ANN	74.4	74.9	72.3
			[1.5]	[1.4]	[1.6]
	Vetorizado CPU	ANN	73.6	74.0	71.2
			[1.5]	[1.4]	[1.6]
	GPU	ANN	72.6	73.5	71.0
			[1.6]	[1.5]	[1.7]
	Sequencial	k-NN	69.3	70.4	68.9
			[1.7]	[1.6]	[1.8]
	Paralelo CPU	k-NN	69.5	70.9	69.1

Base de Dados	**Versão do ACO**	**Algoritmo**	***F-Measure***	***Precision***	***Recall***
			[1.7]	[1.6]	[1.8]
	Vetorizado CPU	k-NN	69.9	70.5	68.6
			[1.7]	[1.6]	[1.8]
	GPU	k-NN	69.9	71.0	69.2
			[1.7]	[1.6]	[1.8]
	Sequencial	*Random Forest*	73.9	74.3	73.2
			[1.5]	[1.4]	[1.6]
	Paralelo CPU	*Random Forest*	74.7	75.0	74.1
			[1.5]	[1.4]	[1.6]
	Vetorizado CPU	*Random Forest*	74.7	74.9	73.4
			[1.5]	[1.4]	[1.6]
	GPU	*Random Forest*	76.1	76.1	75.0
			[1.4]	[1.3]	[1.5]
	Sequencial	SVM	72.6	76.3	69.3
			[1.6]	[1.5]	[1.7]
	Paralelo CPU	SVM	72.5	76.1	69.2
			[1.6]	[1.5]	[1.7]
	Vetorizado CPU	SVM	73.6	77.3	70.2
			[1.6]	[1.5]	[1.7]
	GPU	SVM	71.5	75.1	68.3
			[1.7]	[1.6]	[1.8]
Body Performance	Sequencial	ANN	61.5	62.7	60.4
			[1.8]	[1.7]	[1.9]
	Paralelo CPU	ANN	63.8	64.3	62.0
			[1.7]	[1.6]	[1.8]
	Vetorizado CPU	ANN	59.9	61.0	58.4
			[1.9]	[1.8]	[2.0]
	GPU	ANN	64.0	64.8	62.2
			[1.7]	[1.6]	[1.8]
	Sequencial	k-NN	63.5	64.0	62.9
			[1.8]	[1.7]	[1.9]
	Paralelo CPU	k-NN	62.5	62.9	61.3
			[1.9]	[1.8]	[2.0]
	Vetorizado CPU	k-NN	62.7	63.5	61.9
			[1.9]	[1.8]	[2.0]
	GPU	k-NN	64.7	65.3	64.0
			[1.8]	[1.7]	[1.9]
	Sequencial	*Random Forest*	74.7	75.5	74.1
			[1.5]	[1.4]	[1.6]
	Paralelo CPU	*Random Forest*	73.9	74.7	73.2
			[1.6]	[1.5]	[1.7]
	Vetorizado CPU	*Random Forest*	76.6	77.5	76.2
			[1.4]	[1.3]	[1.5]
	GPU	*Random Forest*	73.9	74.9	73.0
			[1.6]	[1.5]	[1.7]
	Sequencial	SVM	73.5	77.2	70.2

Base de Dados	**Versão do ACO**	**Algoritmo**	***F-Measure***	***Precision***	***Recall***
			[1.6]	[1.5]	[1.7]
	Paralelo CPU	SVM	73.9	77.6	70.5
			[1.6]	[1.5]	[1.7]
	Vetorizado CPU	SVM	76.6	80.4	73.1
			[1.5]	[1.4]	[1.6]
	GPU	SVM	74.7	78.4	71.3
			[1.6]	[1.5]	[1.7]
Presos	Sequencial	ANN	65.4	66.1	64.8
			[1.8]	[1.7]	[1.9]
	Paralelo CPU	ANN	64.9	65.3	63.9
			[1.8]	[1.7]	[1.9]
	Vetorizado CPU	ANN	69.3	69.8	68.5
			[1.7]	[1.6]	[1.8]
	GPU	ANN	67.2	68.3	66.1
			[1.8]	[1.7]	[1.9]
	Sequencial	k-NN	60.8	61.4	60.2
			[1.9]	[1.8]	[2.0]
	Paralelo CPU	k-NN	66.7	67.2	66.1
			[1.7]	[1.6]	[1.8]
	Vetorizado CPU	k-NN	63.9	64.1	62.9
			[1.8]	[1.7]	[1.9]
	GPU	k-NN	62.7	63.9	62.4
			[1.9]	[1.8]	[2.0]
	Sequencial	*Random Forest*	72.0	72.4	71.1
			[1.6]	[1.5]	[1.7]
	Paralelo CPU	*Random Forest*	79.2	79.9	78.1
			[1.3]	[1.2]	[1.4]
	Vetorizado CPU	*Random Forest*	75.2	75.9	74.0
			[1.5]	[1.4]	[1.6]
	GPU	*Random Forest*	73.1	73.6	72.0
			[1.6]	[1.5]	[1.7]
	Sequencial	SVM	68.6	72.0	65.5
			[1.7]	[1.6]	[1.8]
	Paralelo CPU	SVM	69.6	73.0	66.4
			[1.7]	[1.6]	[1.8]
	Vetorizado CPU	SVM	68.7	72.1	65.6
			[1.7]	[1.6]	[1.8]
	GPU	SVM	68.1	71.5	65.0
			[1.8]	[1.7]	[1.9]

Fonte: Dados da Pesquisa

APÊNDICE B – DESEMPENHO ESTATÍSTICO DOS MODELOS PELO TESTE T *STUDENT*

A Tabela 14 apresenta uma análise comparativa por meio do teste T, avaliando se cada versão alcançou um desempenho equivalente à versão sequencial e entre as versões paralelizadas, ou se há diferenças significativas entre elas para cada algoritmo de AM. Por exemplo, na linha "Paralelo CPU = Paralelo GPU", indica-se que, em determinadas bases, o desempenho do algoritmo de AM na versão paralelizada para CPU foi estatisticamente equivalente ao da versão para GPU. Por outro lado, na linha "Paralelo CPU ≠ Paralelo GPU", sinaliza-se que os desempenhos das versões paralelas diferiram, ou seja, uma delas obteve um resultado superior à outra.

Tabela 14: Tabela comparativa entre versões do ACO. O número total de base de dados é 15.

	ANN	**KNN**	**RF**	**SVM**
Seq. = Paralelo CPU	7	4	6	11
Seq. ≠ Paralelo CPU	8	11	9	4
Seq. = Vetor. Paralelo CPU	8	13	10	12
Seq. ≠ Vetor. Paralelo CPU	7	2	5	3
Seq. = Paralelo GPU	6	9	8	14
Seq. ≠ Paralelo GPU	9	6	7	1
Paralelo CPU = Vetor. Paralelo CPU	13	5	12	14
Paralelo CPU ≠ Vetor. Paralelo CPU	2	10	3	1
Paralelo CPU = Paralelo GPU	8	5	3	9
Paralelo CPU ≠ Paralelo GPU	7	10	12	6
Vetor. Paralelo CPU = Paralelo GPU	10	8	9	2
Vetor. Paralelo CPU ≠ Paralelo GPU	5	7	6	13

Fonte: Dados da Pesquisa

Complementando a Tabela 14, a Tabela 15 foi elaborada para evidenciar, entre os casos marcados como divergentes, quais algoritmos e bases de dados apresentaram melhor desempenho, indicando se a versão paralelizada ou sequencial foi superior, representada pelo símbolo ">".

Tabela 15: Tabela Comparativa de Equivalência e Diferença entre Versões do ACO com o Teste T de *Student*

	ANN	**KNN**	**RF**	**SVM**
Seq. > Paralelo CPU	3	6	2	2
Paralelo CPU > Seq.	5	5	7	2
Seq. > Vetor. Paralelo CPU	4	1	3	0
Vetor. Paralelo CPU > Seq.	3	1	2	3
Seq. > Paralelo GPU	4	3	1	1
Paralelo GPU > Seq.	5	3	6	0
Paralelo CPU > Vetor. Paralelo CPU	2	5	1	1
Vetor. Paralelo CPU > Paralelo CPU	0	5	2	0
Paralelo CPU > Paralelo GPU	4	5	7	3
Paralelo GPU > Paralelo CPU	3	5	5	3
Paralelo GPU > Vetor. Paralelo CPU	3	3	4	10
Vetor. Paralelo CPU > Paralelo GPU	2	4	2	3

Fonte: Dados da Pesquisa