UNIVERSIDADE FEDERAL DO PAMPA

Natiele Lucca

Uso de Operações SIMD em uma Biblioteca de Algoritmos Bio-inspirados

Alegrete
2019

Natiele Lucca

Uso de Operações SIMD em uma Biblioteca de Algoritmos Bio-inspirados

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Claudio Schepke

Alegrete
2019

Ficha catalográfica elaborada automaticamente com os dados fornecidos
pelo(a) autor(a) através do Módulo de Biblioteca do
Sistema GURI (Gestão Unificada de Recursos Institucionais) .

L934u Lucca, Natiele
Uso de Operações SIMD em uma Biblioteca de Algoritmos Bio-inspirados / Natiele Lucca.
85 p.

Trabalho de Conclusão de Curso(Graduação)-- Universidade Federal do Pampa, CIÊNCIA DA COMPUTAÇÃO, 2019.
"Orientação: Claudio Schepke".

1. Algoritmos Bio-inspirado. 2. Biblioteca. 3. Instrução Vetorial. 4. OpenMP. 5. Otimização. I. Título.

Natiele Lucca

Uso de Operações SIMD em uma Biblioteca de Algoritmos Bio-inspirados

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Trabalho de Conclusão de Curso defendido e aprovado em 25 de novembro de 2019.
Banca examinadora:

Prof. Dr. Claudio Schepke
Orientador
UNIPAMPA

Prof. Dr. Arthur Francisco Lorenzon
UNIPAMPA

Me. Gabriella Lopes Andrade
PUCRS

AGRADECIMENTOS

Agradeço ao apoio e paciência da minha família e do meu orientador durante a realização desse trabalho. Agradeço também as bolsas PDA Iniciação Científica/Unipampa 2018 e PROBIC/FAPERGS 2018 e 2019.

RESUMO

Algoritmos de otimização buscam resolver problemas complexos. Uma estratégia para modelar algoritmicamente um problema é utilizar conceitos de computação natural, também conhecida como computação bio-inspirada. Os algoritmos bio-inspirados simulam a capacidade de aprendizado ou adaptação de um grupo de indivíduos inteligentes submetidos a um ambiente complexo e assim otimizam uma solução inicial com o objetivo de encontrar soluções ótimas. Dessa forma, neste trabalho é proposto uma biblioteca *open source* bio-inspirada e paralela. Constituem essa biblioteca os algoritmos *Artificial Bee Colony* (Colônia Artificial de Abelhas - ABC), *Ant Colony Optimization* (Otimização por Colônia de Formigas - ACO) e *Particle Swarm Optimization* (Otimização por Enxame de Partículas - PSO), desenvolvidos na linguagem de programação C++. O trabalho testa diretivas de execução paralela e vetorial, comparando o desempenho das diretivas individualmente e combinadas. A diretiva de execução vetorial obteve o maior número de testes com ganho de desempenho, sendo essa diretiva aplicada nos algoritmos selecionados para a biblioteca. Após a implementação paralela, os algoritmos foram submetidos às funções clássicas de teste *Alpine*, *Booth*, *Easom*, *Griewank*, *Rastrigin*, *Rosenbrock* e *Sphere*, sendo aplicados a uma população de 10 até 100 agentes. Os resultados obtidos mostraram que as versões sequencial e paralela obtiveram a mesma solução média e que para pelo menos um teste de população todas as funções obtiveram ganho de desempenho. Com isso, comprovou-se que a versão paralela dos algoritmos desenvolvidos mantém a qualidade da solução e reduz o tempo de execução.

Palavras-chave: Algoritmos Bio-inspirados. Biblioteca. Instrução Vetorial. OpenMP. Otimização. Artificial Bee Colony. Ant Colony Optimization. Particle Swarm Optimization.

ABSTRACT

Optimization algorithms seeks to resolve complex problems. A strategy to algorithmically model a problem is to use concepts of Natural Computing, also known as Bio-inspired Computing. Bio-inspired algorithms simulate the ability of learn or adapt a smart group of individuals subjected to a complex environment and thus optimize initial solution in order to find optimal solutions in a polynomial time. In this way, this work proposes an open source, bio-inspired, and parallel library. This library is composed by Artificial Bee Colony (ABC), Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO) algorithms developed in C++ programming language. The OpenMP API was used for the parallelization of algorithms. The work tests parallel and vectorization directives execution by comparing the performance of individual and combined directives. The vectorization directive execution obtained the highest number of tests with performance gain. This directive was applied to the selected algorithms of the library. After the parallel implementation, the algorithms were submitted to the classic test functions Alpine, Booth, Easom, Griewank, Rastrigin, Rosenbrock and Sphere, being applied to a population of 10 to 100 agents. The results showed that the sequential and parallel versions obtained the same average solution and that, for at least one population test, all functions obtained performance gains. With this, we prove that the parallel version of the developed algorithms maintains the quality of the solution and reduces the execution time.

Key-words: Bioinspired Algorithms. Library. OpenMP. Vectorization instruction. Optimization. Ant Colony Optimization. Artificial Bee Colony. Particle Swarm Optimization.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

ABC *Artificial Bee Colony*

ACO *Ant Colony Optimization*

API *Application Programming Interface*

AVX Advanced Vector Extensions

AVX-512 Advanced Vector Extensions 512

BCO *Bee colonies Optimization*

DE *Differential Evolution*

ES *Evolution Strategies*

ESA *Evolutionary Simulated Annealing*

EvA2 *Evolutionary Algorithms Framework*

FOM *Framework for Metaheuristic Optimization*

GA *Genetic Algorithms*

GPU *Graphics Processing Unit* - Unidade de Processamento Gráfico

IDE *Integrated Development Environment*

LBMAS *Learning-Based Multi-agent System*

MMX Multimedia Extension

MPI *Message Passing Interface*

OpenGL *Open Graphics Library*

OpenMP *Open Multi-Processing*

PaGMO *Parallel Global Multiobjective Optimizer*

ParadisEO *Parallel and Distributed Evolving Objects*

PSO *Particle Swarm Optimization*

Pthreads *POSIX Threads*

SA *Simulated Annealing*

SIMD *Single Instruction Multiple Data*

SMA *Swarms of Metaheuristic Agents*

SSE Streaming SIMD Extensions

SSE2 Streaming SIMD Extensions 2

SSE3 Streaming SIMD Extensions 3

SSE4 Streaming SIMD Extensions 4

TS *Tabu Search*

ULA Unidade Lógica e Aritmética

LISTA DE SÍMBOLOS

η_{ij}	Heurística local
ρ	Taxa de evaporação do feromônio
BN	Quantidade de abelhas campeiras
D	Tamanho do espaço
S	Speedup
SN	Número de fontes de alimento

SUMÁRIO

1 INTRODUÇÃO

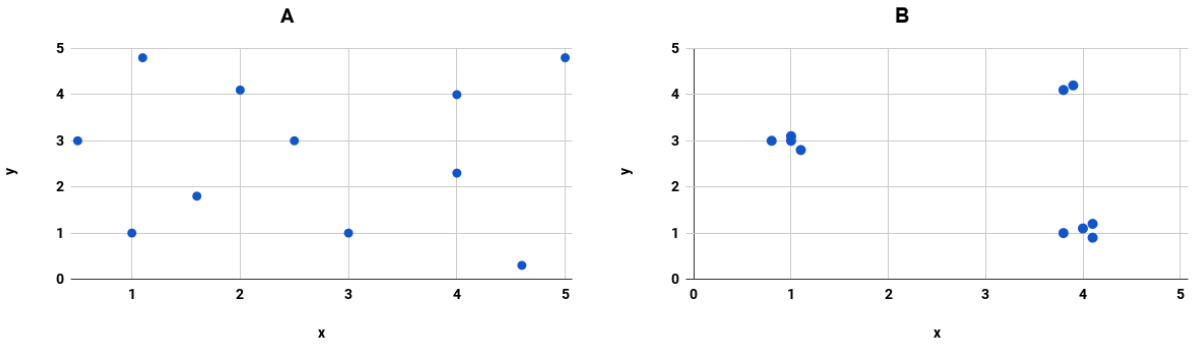
Problemas reais de engenharia, ciência e economia por exemplo, não podem ser resolvidos de forma exata e sequencial devido ao elevado tempo de computação para encontrar a solução ótima (IGNÁCIO; FERREIRA, 2002). A capacidade de processamento de um computador não é suficiente para realizar o esforço matemático exigido para encontrar a solução desses problemas. Para isso é necessário que os algoritmos utilizem abordagens que otimizem a execução das tarefas, assim podem solucionar esses problemas de forma eficiente (NIEVERGELT et al., 1995).

A computação natural é uma estratégia de otimização que aplica conceitos da biologia na solução de problemas complexos (CASTRO et al., 2004). Esses problemas possuem um grande número de variáveis ou soluções potenciais, sendo que nem sempre é possível garantir que uma solução encontrada pelo algoritmo seja ótima (CASTRO, 2007).

São exemplos de problemas complexos, os clássicos caixeiro viajante e roteamento de veículos. Outras aplicações são: o trabalho de Lin e Lucas (2015) que apresenta um modelo de evacuação de aeronaves sob emergência com emoções para simular o efeito de passageiros com medo ou pânico; a pesquisa de Diciolla et al. (2015) que classifica e gera uma previsão para pacientes com doença renal terminal (DRC); e o trabalho de Carvalho et al. (2016) que apresenta um algoritmo bio-inspirado para controlar o tráfego em sistemas de elevadores.

Os algoritmos bio-inspirados aplicam técnicas de inteligência de enxames ou inteligência de colônias, ou ainda inteligência coletiva, que simulam o comportamento coletivo de sistemas auto-organizados, distribuídos, autônomos, flexíveis e dinâmicos (SERAPIAO, 2009). Um enxame ou colônia é uma população de elementos que interagem e são capazes de otimizar um objetivo global através da busca colaborativa em um espaço seguindo regras específicas.(KENNEDY; EBERHART, 2001).

Figura 1 – Malha Aleatória x Malha Convergindo



Fonte: Autora

A simulação é a capacidade de transformar uma solução inicial aleatória em uma solução factível e ótima, evoluindo de acordo com as iterações e seguindo as regras específicas de cada modelo (SERAPIAO, 2009). A Figura 1-A mostra uma malha aleatória e a Figura 1-B uma malha convergindo. A primeira malha representa a inicialização do algoritmo onde os indivíduos ou agentes são dispostos de forma aleatória pelo espaço de busca. Já a malha convergindo expressa a solução quando os indivíduos estão aglomerados em um ou mais grupos que representam as melhores soluções encontradas.

Na execução sequencial uma tarefa é executada após a outra. Assim que uma tarefa encerra a seguinte da fila de execução inicia. Já a execução concorrente possibilita que tarefas possam ser executadas simultaneamente, aumentando o desempenho, através de replicas das unidades vetoriais ou Unidade Lógica e Aritmética (ULA). Isso permite que ao mesmo tempo duas ou mais tarefas possam ser executadas em unidades distintas.

Além da replicação de unidades vetoriais e ULA, as unidades de processamento também foram replicadas. Dessa forma, o paralelismo expresso por múltiplos núcleos em processadores *multicore* permite a execução de instruções diferentes e simultâneas. Essa estratégia possui restrições físicas como consumo energético e dissipação de calor que limitam a replicação de cores.

A arquitetura *Single Instruction Multiple Data* (SIMD) compreende um ou mais processadores com n cores que executam uma instrução sobre múltiplos dados (FLYNN; RUDD, 1996). Nesse trabalho foi desenvolvida uma biblioteca de algoritmos bio-inspirados que aplica a concorrência entre unidades vetoriais. A interface de programação paralela *Open Multi-Processing* (OpenMP)[1] é utilizada para melhorar o desempenho dos algoritmos bio-inspirados.

1.1 Objetivos

O objetivo geral deste trabalho é implementar uma biblioteca paralela na linguagem de programação C++ para algoritmos bio-inspirados, a fim de auxiliar os desenvolvedores na solução de problemas. Os objetivos específicos são os seguintes:

- Implementar os algoritmos: Enxame de Partículas (PSO – *Particle Swarm Optimization*), Colônias de Formigas (ACO — *Ant Colony Optimization*) e Colônia Artificial de Abelhas (ABC — *Artificial Bee Colony*).
- Implementar duas versões de cada algoritmo, uma sequencial e outra paralela.
- Submeter os algoritmos a casos de teste.
- Verificar o desempenho paralelo para cada um dos casos de teste.

[1] Disponível em <https://www.openmp.org/>

1.2 Justificativa

Bibliotecas específicas são consolidadas tanto na indústria como na academia, pois implementam características de baixo nível que facilitam a programação de aplicações. Os algoritmos desenvolvidos com bibliotecas abstraem elementos, o que permite ao desenvolvedor aprimorar a aplicação e agilizar o processo de desenvolvimento. *Open Graphics Library* (OpenGL)[2] é um exemplo de biblioteca utilizada na área de computação gráfica que fornece recursos como animação, tratamento de imagens e texturas. Com esse recurso o desenvolvedor dispõem de rotinas implementadas para a exibição e o processamento de imagens ou objetos (RIEDER; BRANCHER, 2002). Entretanto, não é dispensável o conhecimento sobre o funcionamento dos métodos.

Neste trabalho foram escolhidos para implementação os seguintes algoritmos: Enxame de Partículas (PSO – *Particle Swarm Optimization*); Colônias de Formigas (ACO – *Ant Colony Optimization*); e Colônia Artificial de Abelhas (ABC – *Artificial Bee Colony*). De acordo com os trabalhos relacionados (seção 2.5) esses são os algoritmos de inteligência de enxame que as bibliotecas pesquisadas mais implementam.

Os algoritmos de enxame disponíveis na literatura são implementados de diversas formas. Alguns utilizam bibliotecas para facilitar a manipulação de vetores e matrizes enquanto outros usam para operar expressões e para gerar valores aleatórios. Os desenvolvedores utilizam os recursos da linguagem de programação adotada para explorar e facilitar a implementação, o que destaca a importância da existência de uma biblioteca que permita a implementação de qualquer problema e abstraia a manipulação dos objetos.

Os algoritmos bio-inspirados possibilitam encontrar soluções ótimas para problemas complexos, mas não são simples de implementar, pois, exigem conhecimentos de programação como a manipulação de estruturas de dados. Visto que os problemas complexos compreendem áreas distintas, uma biblioteca paralela de algoritmos bio-inspirados permite que diferentes profissionais implementem soluções minimizando o tempo de desenvolvimento e maximizando o desempenho.

1.3 Organização deste trabalho

Este trabalho está organizado em 5 capítulos. O Capítulo 2 apresenta uma descrição da computação natural e suas áreas com enfase na inteligência de enxame. São detalhados os algoritmos ABC, ACO e o PSO e as respectivas estruturas. Também é apresentada uma visão geral de aplicações paralelas, bem como a descrição dos trabalhos relacionados. O Capítulo 3 apresenta os métodos que serão aplicados na execução desta pesquisa, descrevendo como serão realizados os testes e a forma de análise dos resultados. O Capítulo 4 contém os detalhes da implementação, os resultados obtidos nos testes

[2] Disponível em <https://www.opengl.org/>

realizados conforme descrito no capítulo anterior e a análise dos resultados. Por fim, o Capítulo 5 disserta as considerações finais sobre o trabalho e as perspectivas futuras.

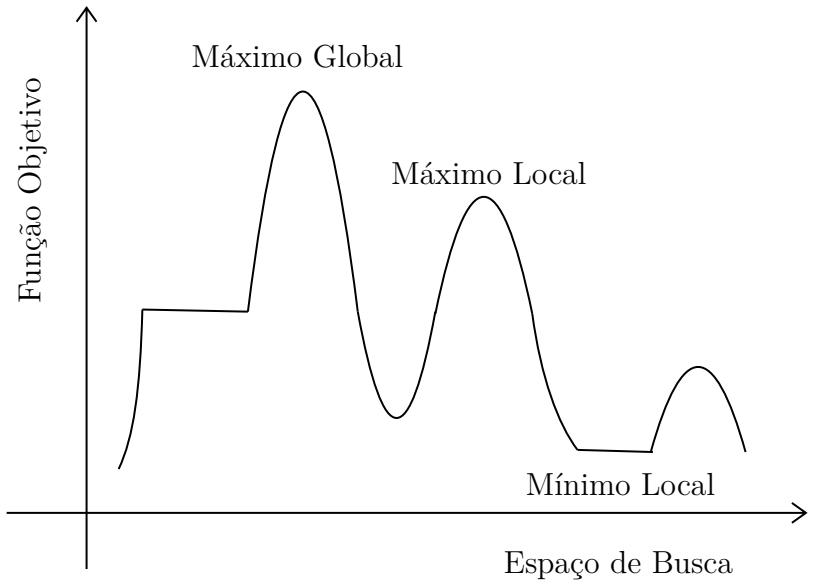
2 ASPECTOS CONCEITUAIS

Este capítulo apresenta uma visão geral da teoria da computação natural e da inteligência de enxames. Ele demonstra a estrutura e a aplicação dos algoritmos selecionados para compor a biblioteca. O capítulo contém a especificação sobre como a interface paralela funciona e os trabalhos relacionados identificados na literatura.

2.1 Computação Inspirada na Natureza

Na natureza as espécies usam a sociabilidade de diversas formas. O comportamento imprevisível de certas espécies beneficia as populações contra predadores, na coleta de alimento e na busca por parceiros ou abrigo. Além do comportamento aleatório, outra característica observável nas espécies é forma com que combinam os indivíduos, a fim de garantir a sobrevivência da população com as características mais favoráveis (KENNEDY; EBERHART, 2001). Esses comportamentos são simulados em algoritmos de computação natural, com o objetivo de encontrar as melhores soluções para um problema.

Figura 2 – Soluções de uma Função Objetivo



Fonte: Adaptado de Norvig e Russell (2014)

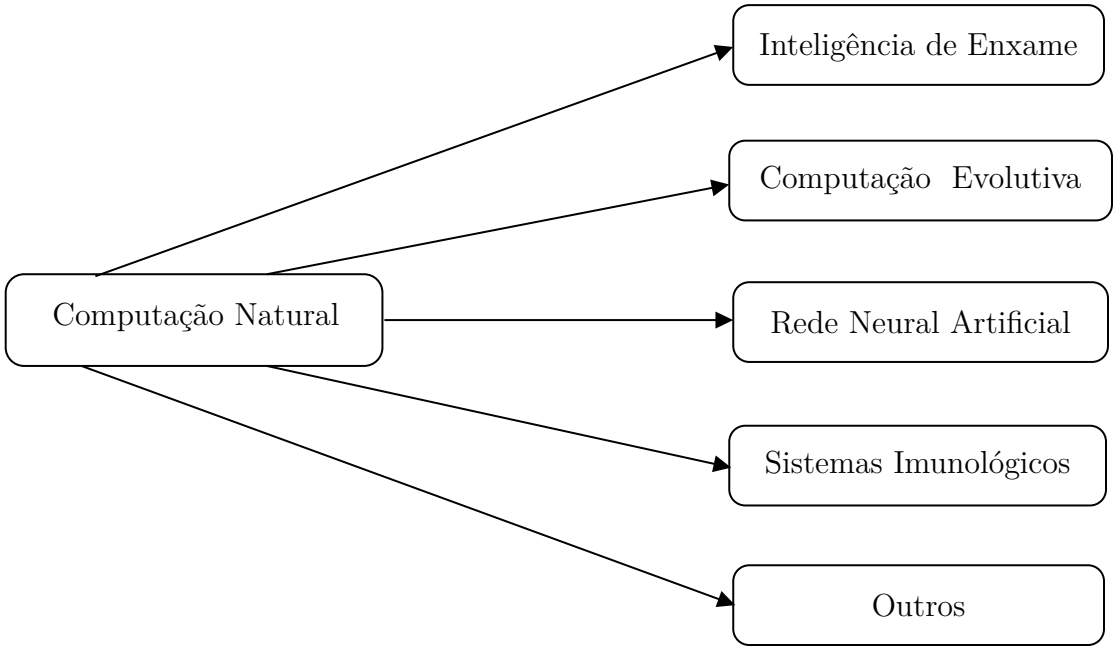
Dada um problema de otimização (maximização ou minimização) sobre uma função $y = f(x)$. O problema é de otimização onde o objetivo será minimizar por exemplo, o custo da viagem como no problema do caixeiro viajante, ou maximizar o lucro de algo. O problema pode ser resolvido algebricamente testando todas as possibilidades do espaço de busca. Entretanto, existem estratégias como a computação natural que visa minimizar os custos dessa computação. Essa estratégia permite que uma população disposta aleato-

riamente no espaço de busca sofra combinações e ao fim de m iterações o ponto máximo ou mínimo da função seja encontrado.

A Figura 2 apresenta o gráfico de uma função $y = f(x)$, onde tem-se um problema de maximização com o objetivo de encontrar o ponto máximo da função. Nessa figura são destacados os termos mínimo local, máximo local e máximo global. Para uma função de maximização os mínimos globais e locais são pontos em que a solução é ruim. Enquanto os máximos locais são soluções factíveis e o máximo global é a solução ótima para o problema. Entretanto, é notável que classificar uma solução como máximo local ou global é uma tarefa difícil, pois a solução ótima não é conhecida. Para resolver esse problema cada algoritmo de computação natural aplica uma técnica. Alguns aplicam o conceito de variação aleatória que altera a população a cada iteração mesmo que a solução piore, visto que, às vezes é preciso priora-lá para obter soluções melhores. Outros algoritmos modificam a população de acordo com uma equação probabilística. Essas técnicas são descritas no decorrer do trabalho.

Nesse contexto, a computação inspirada na natureza é subdivida em algoritmos inteligentes que incluem redes neurais artificiais, computação evolutiva, inteligência de enxame, sistemas imunológicos artificiais e outros (ENGELBRECHT, 2007). A Figura 3 apresenta um fluxograma que representa as áreas da computação bio-inspirada.

Figura 3 – Áreas da Computação Inspirada na Natureza.



Fonte: Adaptado de Engelbrecht (2007)

A inteligência de enxame simula o comportamento de agentes diante da colônia. Esses são inteligentes e autônomos não só realizam tarefas, mas tem a capacidade de

tomar decisões (KENNEDY; EBERHART, 2001). São exemplos ACO, ABC, *Bee colonies Optimization* (BCO) e PSO.

A computação evolutiva é baseada na evolução natural das espécies, onde a população se reproduz transferindo e combinando aos outros indivíduos suas características genéticas (BANZHAF et al., 1998). São exemplos de algoritmos dessa área da computação bio-inspirada o *Genetic Algorithms* (GA), *Differential Evolution* (DE), *Simulated Annealing* (SA), *Evolutionary Simulated Annealing* (ESA) e *Evolution Strategies* (ES).

As redes neurais artificiais simulam a capacidade cognitiva do ser humano, através dos neurônios que interconectados permitem a troca de informação (RAUBER, 2005).

Os sistemas imunológicos simulam a capacidade humana de identificar e distinguir os componentes que pertencem ao sistema (próprios) dos estranhos ou invasores (não-próprios), principalmente, elementos estranhos que possam causar problemas (CASTRO, 2001).

2.2 Inteligência de Enxames

Serapiao (2009) define a inteligência de enxames ou inteligência de colônias, ou ainda inteligência coletiva, é um conjunto de técnicas baseadas no comportamento coletivo de sistemas auto-organizados, distribuídos, autônomos, flexíveis e dinâmicos. Os algoritmos que aplicam essa técnica simulam a forma com que o enxame (grupo de agentes) interagem e a forma com que tomam decisões.

A Figura 4 apresenta um algoritmo de inteligência de enxame genérico. Inicialmente o algoritmo gera uma população inicial que normalmente é aleatória (critério do desenvolvedor). Em seguida essa população é avaliada de acordo com uma função de qualidade, analisada e em caso de ótimo local armazenada. O critério de parada é uma maneira de controlar a quantidade de iterações que serão realizadas no algoritmo, caso não satisfeito as alterações são realizadas sob a população de acordo com a estratégia de cada algoritmo a fim de melhorar e alcançar a função objetivo; ao fim das iterações a melhor solução é retornada.

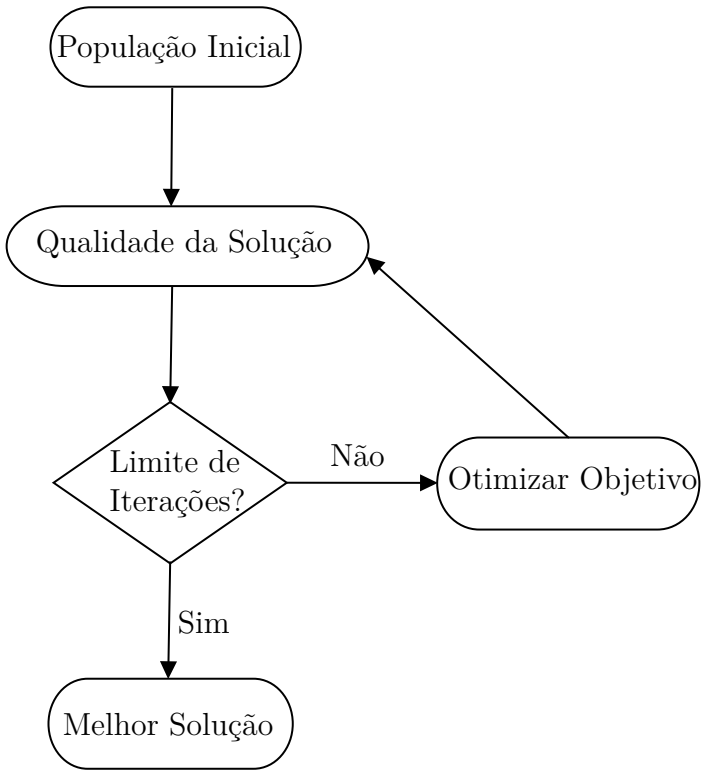
2.3 Algoritmos

Esta seção discute os algoritmos ABC, ACO e PSO, que foram selecionados para o estudo. Também são apresentados a estrutura lógica do funcionamento e as características de cada algoritmo, assim como o respectivo pseudocódigo.

2.3.1 Colônia Artificial de Abelhas - ABC

Uma colônia de abelhas é formada por abelhas campeiras, seguidoras e escudeiras. Metade do enxame são abelhas campeiras e a outra metade são abelhas seguidoras. As abelhas campeiras são responsáveis por explorar o espaço de busca e encontrar fontes de

Figura 4 – Algoritmo de Inteligência de Enxame Genérico



Fonte: Autora

alimento que são possíveis soluções para o problema. As seguidoras são abelhas recrutadas pelas campeiras de acordo com a qualidade da fonte encontrada. Já as escudeiras são abelhas campeiras que encontraram uma fonte ruim e as abandonaram (KARABOGA; BASTURK, 2007).

A população de abelhas inicial é composta por possíveis soluções para o problema. Cada solução x_i é gerada randomicamente a partir da Equação 2.1, sendo que min e max são os limites do espaço de busca e $rand(0,1)$ é um valor aleatório entre o intervalo [0,1] (KARABOGA; BASTURK, 2007).

$$x_{i,j} = x_{i,j}^{min} + rand(0,1) * (x_{i,j}^{max} - x_{i,j}^{min}) \tag{2.1}$$

Após a inicialização das abelhas a população é avaliada, ou seja, a qualidade da solução é verificada de acordo com a Equação 2.2. Sendo que $f(i)$ é o valor da função objetivo para a solução x_i.

$$fit_i = \begin{cases} \frac{1}{(1+f_i)}, & \text{se } f_i >= 0 \\ 1 + |f_i|, & \text{se } f_i < 0 \end{cases} \tag{2.2}$$

Cada abelha campeira busca através da Equação 2.3 encontrar uma nova fonte de alimento. Sendo que $v_{i,j}$ é uma solução candidata a substituir $x_{i,j}$, onde k $\in$ {1, 2,...,

BN[1]} e j ∈ {1, 2, ..., D[2]}, de modo que k ≠ i e $\phi_{i,j}$ é um número aleatório entre [-1, 1] (SERAPIAO, 2009). Se a solução candidata for melhor que a solução atual, a solução $x_{i,j}$ é atualizada para a solução $v_{i,j}$, caso a solução candidata for pior, a solução $x_{i,j}$ é mantida.

$$v_{i,j} = x_{i,j} + \phi_{i,j} * (x_{i,j} - -x_{k,j}) \tag{2.3}$$

As abelhas seguidoras buscam explorar as solução encontradas pelas abelhas campeiras. Dessa forma as abelhas seguidoras escolhem uma solução de acordo com uma probabilidade. Essa probabilidade é dada pela Equação 2.4, onde x_i é uma solução encontrada pelas abelhas campeiras, SN é o número de fontes de limento que é igual a quantidade de abelhas campeiras e $fit(x_i)$ representa a aptidão ou qualidade de uma solução x_i (SERAPIAO, 2009).

$$P(i) = \frac{fit(x_i)}{\sum_{n=1}^{SN} fit(x_n)} \tag{2.4}$$

Em sequência o limite de tentativas de melhorar uma solução é verificado, dessa forma soluções que estão estagnadas por um limite de iterações são abandonadas e então as abelhas escudeiras geram uma nova solução inicial de acordo com a Equação 2.1.

Após a verificação de fontes estagnadas a melhor solução encontrada pela população é armazenada. Caso o critério de parada for satisfeito, o algoritmo retorna a melhor solução encontrada, caso contrário retorna ao passo de exploração das abelhas campeiras.

O pseudocódigo do algoritmo clássico ABC segundo Serapiao (2009) está representado no algoritmo 1:

2.3.2 Otimização por Colônias de Formigas - ACO

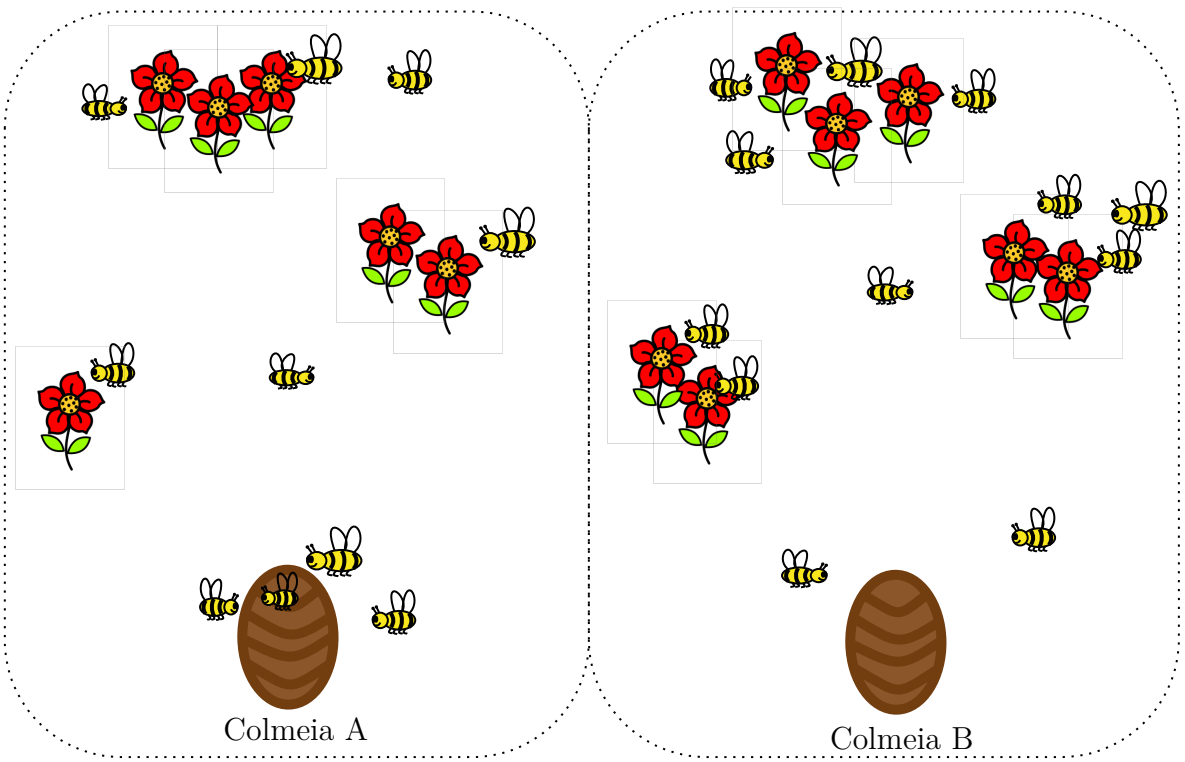
O algoritmo de colônias de formigas foi proposto por Dorigo e Birattari (2010) e inspirado no comportamento de formigas na busca por alimento. As formigas partem do formigueiro em busca de fontes de alimento, ou seja, elas exploram o espaço de busca liberando feromônio pelo caminho percorrido. Durante um determinado período de tempo as formigas buscam por uma fonte de alimento, mas aquelas que encontram fonte mais próximas do formigueiro concentram uma quantidade maior de feromônio, pois elas percorrem o caminho mais de uma vez. Portanto, quanto mais química uma trilha for, ou seja, quanto maior for a concentração de feromônio mais formigas serão atraídas para ela. Dessa forma, a maioria das formigas tendem a percorrem o mesmo caminho (SERAPIAO, 2009).

[1] Quantidade de abelhas campeiras
[2] Espaço D-dimensional

Algoritmo 1: Pseudocódigo ABC

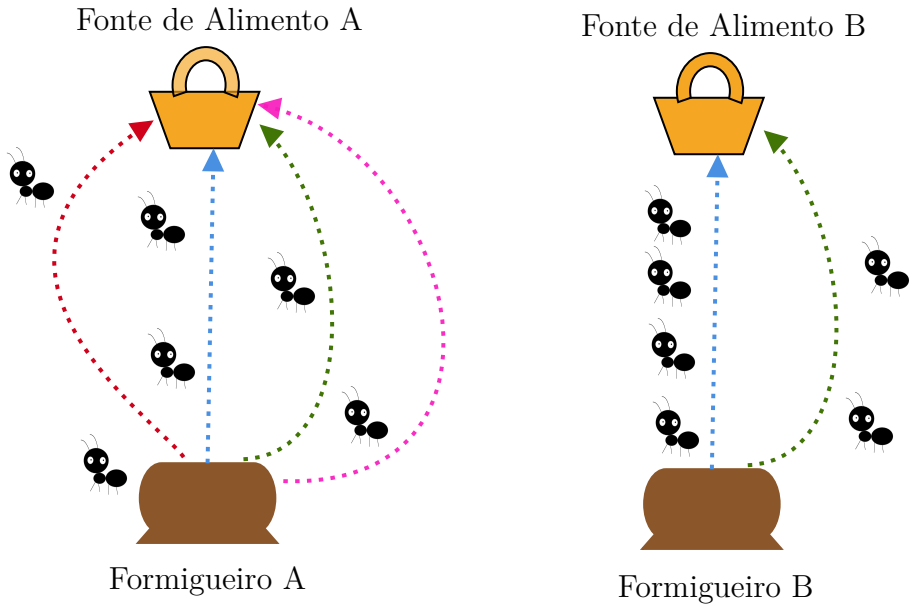
```
1  início
2      Inicializar a população ;                                  // solução inicial
3      Avaliar a aptidão da população inicial;
4      Memorizar a melhor solução ;
5      enquanto critério de parada não for satisfeito faça
6          Gerar novas soluções ;                                 // abelhas campeiras
7          Calcular a probabilidade de escolha das soluções;
8          Explorar as soluções de acordo com a probabilidade ;   // abelhas
            seguidoras
9          Interromper a exploração das piores fontes ;
10         Gerar novas soluções aleatórias ;                      // abelhas escudeiras
11         Memorizar a melhor solução ;
12     fim
13     retorna Melhor Solução Encontrada ;
14 fim
```

Figura 5 – ABC - Representação da Coleta de Néctar



Fonte: Autora

Figura 6 – ACO - Representação da Busca por Alimento



Fonte: Autora

Na Figura 6, o Formigueiro A representa a inicialização do algoritmo, onde cada formiga busca um caminho até a fonte de alimento; já o Formigueiro B representa o menor caminho entre o formigueiro e a fonte de alimento. Esse caminho foi seguido pelas outras formigas devido à tendência das formigas escolherem o trecho com a maior quantidade de feromônio.

O algoritmo ACO é utilizado para solucionar problemas como o caixeiro viajante, pois o caminho que acumula mais feromônio é consequentemente o mesmo que o agente leva menos tempo para percorrer. Outros exemplos de aplicações para o ACO são otimização de roteamento em redes de telecomunicações e coloração de grafos (ENGELBRECHT, 2007).

Para construir uma solução cada formiga *(k)* utiliza iterativamente uma função probabilística para decidir se incluirá ou não determinada aresta na solução, expressa na Equação 2.5. Para atualizar a trilha de feromônio a Equação 2.6 é calculada. Onde $\tau_{i,j}$ é a trilha de feromônio da combinação (i, j), $\eta_{i,j}$ é a heurística local da combinação (i, j), α é a importância relativa da trilha de feromônio, β é a importância relativa da heurística local, ρ é a persistência da trilha (taxa de evaporação) e $\Delta\ \tau_{i,j}$ é a quantidade de feromônio que será depositada na aresta (i, j). Se $(i, j) \in$ S a Equação 2.7 deve ser usada, caso contrário Equação 2.8.

$$p_{i,j}^{k}(t) = \begin{cases} \frac{\tau_{i,j}(t) * \eta_{i,j}^{\beta}}{\sum_{j \in J^k} \tau_{i,j}^{\alpha}(t) * \eta_{i,j}^{\beta}}, & \text{se } j \in J^k \\ 0, & \text{caso contrário} \end{cases} \tag{2.5}$$

$$\tau_{i,j}(t+1) = (1-(-\rho)) * \tau_{i,j}(t) + \rho * \Delta\tau_{i,j} \tag{2.6}$$

$$\Delta\tau_{i,j} = \frac{1}{f(s)} \tag{2.7}$$

$$\Delta\tau_{i,j} = 0 \tag{2.8}$$

O algoritmo ACO é aplicado principalmente em problemas que envolvem domínios discretos, mas a estratégia adotada pelo algoritmo também pode ser aplicada em domínios contínuos. O ACO pode ser adaptado para domínios contínuos alterando a forma de distribuição da probabilidade (DARIANE; MORADI, 2009).

Para calcular a distribuição da probabilidade é utilizada uma soma ponderada de k funções, de acordo com a Equação 2.9 (SOCHA; DORIGO, 2008). Onde ω é o vetor de pesos, μ é o vetor de médias e σ o vetor de desvio padrão. Sendo que o tamanho dos vetores é igual, ou seja, $|\omega| = |\sigma| = |\mu| =$ k.

$$P^i(x) = G^i(x) = \sum_{l=1}^{k} \omega_l^i \frac{1}{\sigma_l^i\sqrt{2\pi}} \exp \frac{1}{2\sigma_l^i}(x-\mu_l^i)^2 \tag{2.9}$$

O ACO discreto armazena as informações do feromônio em uma tabela, já o ACO para domínios contínuos armazena as soluções geradas e os respectivos valores da função objetivo em um arquivo ou estrutura de dados.

O peso ω_l de cada solução é dado pela Equação 2.10 (SOCHA; DORIGO, 2008), sendo q um parâmetro entre [0.0001; 1] para valores baixos as melhores soluções são selecionadas o que gera pouca diversidade na população e, para valores próximos a 1 a probabilidade de escolha é uniforme gerando diversidade. O componente μ é determinado pela Equação 2.11 (SOCHA; DORIGO, 2008).

$$\omega_l = \frac{1}{\sigma_l^2\sqrt{2\pi}} \exp \frac{-(1-1)^2}{2q^2k^2} \tag{2.10}$$

$$\mu^i = \{\mu_1^i, ..., \mu_k^i\} = \{s_1^i, ..., s_k^i\} \tag{2.11}$$

O componente σ é calculado pela Equação 2.12 (SOCHA; DORIGO, 2008), onde ξ é a taxa de evaporação do feromônio e s^i é o conjunto de soluções do algoritmo.

$$\sigma_l^i = \sum_{e=1}^{k} \xi \frac{|s_e^i - s_l^i|}{k-1} \tag{2.12}$$

O pseudocódigo do algoritmo ACO para domínios contínuos, segundo Serapiao (2009) é representado no algoritmo 2:

Algoritmo 2: Pseudocódigo ACO

```
1  início
2      Inicializar as formigas ;                          // Soluções iniciais
3      Calcular o peso das soluções ;                                  // ω_l
4      Calcular a probabilidade de escolha de soluções ;
5      Inicializar o arquivo de soluções ;
6      Memorizar a melhor solução inicial ;
7      enquanto critério de parada não for satisfeito faça
8          Gerar novas soluções ;      // Seleção de acordo com a probabilidade
9          Calcular o peso das novas soluções ;
10         Calcular a probabilidade de escolha das novas soluções ;
11         Inserir as novas soluções no arquivo ;
12         Remover do arquivo as piores soluções ;        // Evaporar o feromônio
13         Memorizar a melhor solução ;
14     fim
15     retorna Melhor Solução Encontrada
16 fim
```

2.3.3 Otimização por Enxame de Partículas - PSO

O algoritmo otimização por Enxame de Partículas (PSO) foi modelado a partir do comportamento social do bando de aves (ENGELBRECHT, 2007). No PSO, a população de indivíduos ou partículas é agrupada em um enxame (conjunto de soluções). Essas partículas simulam o comportamento de pássaros através do aprendizado próprio (componente cognitivo) e o aprendizado do bando (componente social), ou seja, imitam o seu próprio sucesso e o de indivíduos vizinhos (TAVARES; NEDJAH; MOURELLE, 2015). A partir desse comportamento as partículas podem definir novas posições que as direcionam para soluções ótimas.

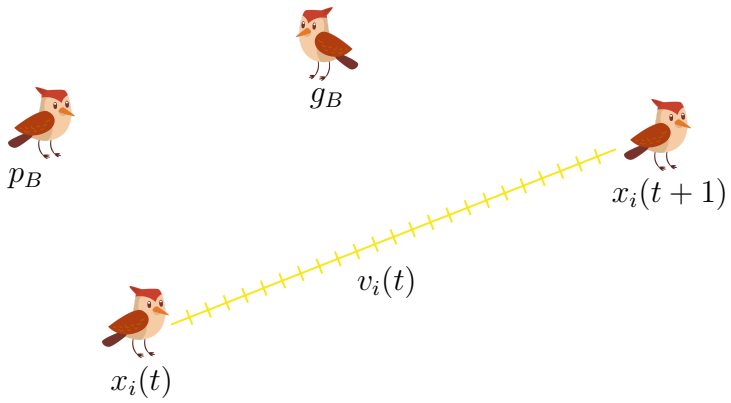
As posições das partículas são inicializadas aleatoriamente pelo espaço de busca com velocidade de descocamento igual a zero (ENGELBRECHT, 2007). Após a inicialização a qualidade ou aptidão das soluções são calculadas pela função objetivo.

Segundo Serapiao (2009) uma partícula realiza movimentos em uma determinada direção influenciado pela velocidade da partícula, de forma que a velocidade não seja constante. A velocidade de uma partícula é obtida pela Equação 2.13, sendo $x_i(t)$ é a posição atual da partícula, v_i a velocidade atual da partícula, (p_B) a melhor solução da partícula e (g_B) a melhor solução do enxame.

$$v_i(t+1) = v_i(t) + \varphi 1 * (p_B - x_i(t)) + \varphi 2 * (g_B - x_i(t)) \tag{2.13}$$

As constantes $\varphi 1$ e $\varphi 2$ representam a componente cognitivo e o social respectivamente, elas também podem ser identificadas como fatores *gbest* e *lbest* que influenciam no movimento das partículas (CHIACHIA; PENTEADO; MARANA, 2003). O *gbest* representa o melhor global sendo a melhor posição encontrada pelo enxame, o *lbest* é o melhor

Figura 7 – PSO - Representação do Voo dos Pássaros



Fonte: Autora

local e é a melhor posição encontrada pela partícula i.

A nova posição de uma partícula i é obtida a partir da sua posição atual e pela velocidade atualizada, sendo expressa na Equação 2.14 (SERAPIAO, 2009). A Figura 7 representa o deslocamento ou a nova posição de um pássaro.

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{2.14}$$

A velocidade das partículas são limitadas por v_{max} e v_{min}, de acordo com a Equação 2.15. A velocidade calculada que não está entre os limites é ajustada para o limite correspondente. Dessa forma, as partículas são mantida dentro do espaço de busca estabelecido (SERAPIAO, 2009).

$$\begin{cases} \text{Se } v_i > v_{max} \text{ então } v_i = v_{max} \\ \text{Senão se } v_i < v_{min} \text{ então } v_i = v_{min} \end{cases} \tag{2.15}$$

Após realizar as atualizações de velocidade e posição, se o critério de parada for satisfeito a melhor solução encontrada pelo enxame é retornada. Caso não for satisfeito o algoritmo retorna para o passo do cálculo de aptidão.

O pseudocódigo do algoritmo clássico PSO segundo Serapiao (2009) está apresentado no algoritmo 3:

2.4 Paralelismo

Um algoritmo sequencial é formado por uma série de instruções que são executadas uma após a outra pela unidade de processamento. Em programação paralela um problema é dividido em áreas que podem ser executadas simultaneamente pois não possuem dependências de dados ou de instruções. Para permitir que mais de uma instrução

Algoritmo 3: Pseudocódigo PSO

```
1  início
2      Inicializar as partículas ;                          // Soluções iniciais
3      Inicializar a velocidade das partículas ;
4      enquanto critério de parada não for satisfeito faça
5          Calcular a aptidão das soluções;
6          Memorizar a melhor solução local ;                    // Atualizar p_B
7          Memorizar a melhor solução global ;                   // Atualizar g_B
8          Atualizar a velocidade das partículas ;
9          Atualizar a posição das partículas ;      // De acordo com a velocidade
10     fim
11     retorna Melhor Solução Encontrada
12 fim
```

seja executada simultaneamente o hardware é replicado, ou seja, há fisicamente mais unidades de processamento (BARNEY, 2018).

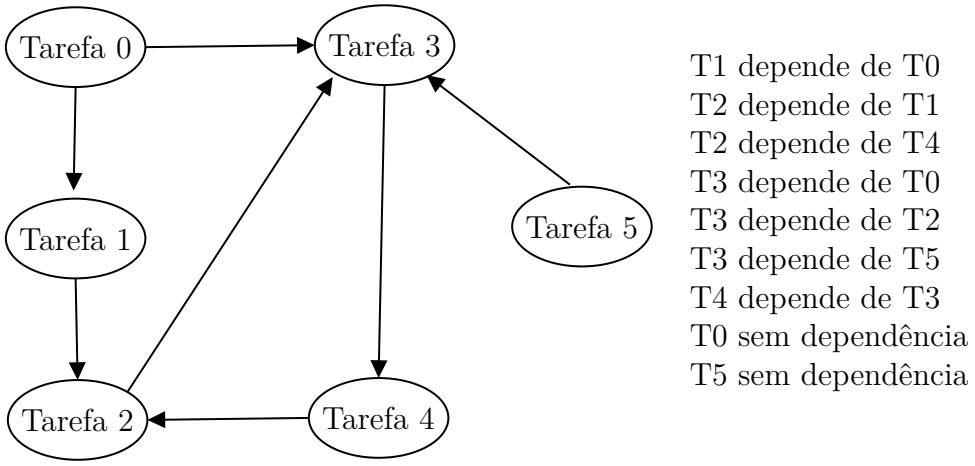
> Uma arquitetura paralela fornece uma estrutura explícita e de alto nível para o desenvolvimento de soluções utilizando o processamento paralelo, através da existência de múltiplos processadores que cooperam para resolver problemas através de execução concorrente (DUNCAN, 1990).

Nem todas as tarefas podem ser paralelizadas devido a dependência que existe entre elas, certas atividades somente podem ser executadas após outra determinada tarefa ser concluída. Algumas dessas tarefas possuem dependência de dados quando necessitam de informações que serão calculadas por outras tarefas, outras possuem dependência de instruções quando operam sobre registradores que também serão utilizados por outras tarefas (UFRJ, 2011). A Figura 8 apresenta um grafo de tarefas, ou seja, uma representação gráfica que compreende 5 tarefas e as dependências que existem entre elas.

Uma aplicação paralela pode explorar a estratégia de organização do algoritmo para ganho de desempenho. A forma de programar pode influenciar o tempo de execução de um algoritmo pois, de acordo com o volume de dados manipulados, a cache pode encher exigindo assim uma sequência de substituições que necessitam de acesso a memória RAM ou até mesmo ao disco rígido (*swap*). Por exemplo, na multiplicação de matrizes, o ideal é organizar o código de forma que os dados sejam carregados em cache o mínimo de vezes possível.

Paralelizar uma aplicação significa identificar áreas que podem ser executadas simultaneamente de forma que a solução seja mantida ou melhorada em casos de otimização. A Figura 9 apresenta a execução de um algoritmo genérico de maneira sequencial e paralela. Na execução sequencial todas as tarefas são executadas em série, já na paralela as tarefas são distribuídas entre *threads* concorrentes, por consequência o tempo de execução da versão paralela é inferior, pois os recursos de hardware disponíveis são melhor explorados.

Figura 8 – Grafo de Tarefas



Fonte: Autora

A computação paralela tem sido usada para modelar, simular e compreender problemas complexos que envolvam: atmosfera, terra, física aplicada, partículas, alta pressão, fusão, biotecnologia, genética, química, engenharia mecânica, engenharia elétrica, microeletrônica, ciência da computação, matemática entre outros (BARNEY, 2018).

2.4.1 Arquitetura de Processadores Vetoriais

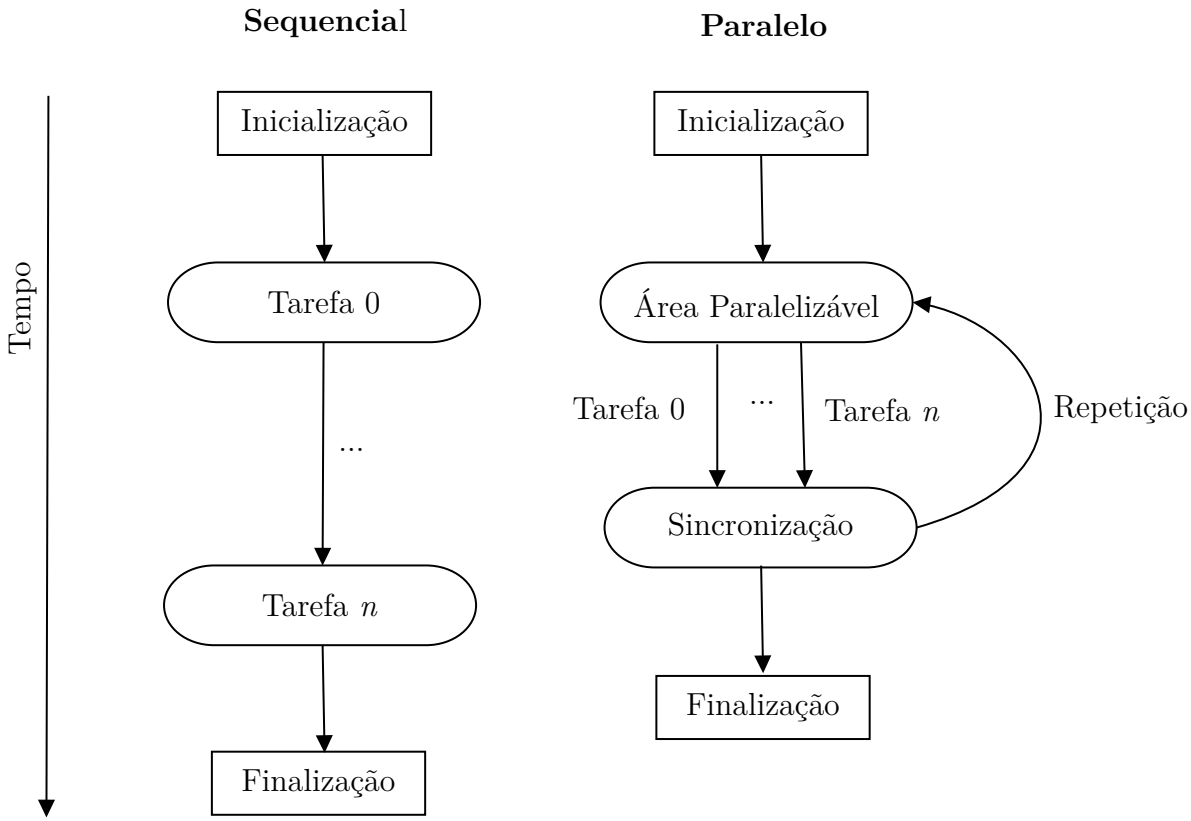
De acordo com Flynn e Rudd (1996), os processadores vetoriais são classificados como arquitetura SIMD. Esses processadores realizam uma única instrução sobre múltiplos dados, sendo que os dados devem estar organizados em um vetor. Dessa forma, o paralelismo a nível de dados é explorado, o qual permite reduzir o tempo de execução.

A Figura 10 apresenta a soma do vetor A com o vetor B. A soma dos vetores em uma arquitetura sem registradores vetoriais acontece sequencialmente através de um laço de repetição que leva 8 ciclos para realizar a soma. Em um cenário hipotético com arquitetura SIMD e 4 registradores vetoriais, a soma acontece em 2 ciclos, pois são realizadas 4 somas simultâneas por ciclo.

Um processador vetorial é composto por: registradores vetoriais, unidades funcionais, unidades de *load/store* e registradores escalares. Os registradores vetoriais são registradores em que os dados são organizados em forma de vetor, ou seja, em sequência. Dessa forma em uma arquitetura que não possui unidades vetoriais, o processamento de uma operação é realizado sobre um escalar, enquanto em em uma arquitetura SIMD a unidade de processamento opera sobre um vetor (ALVES et al., 2015).

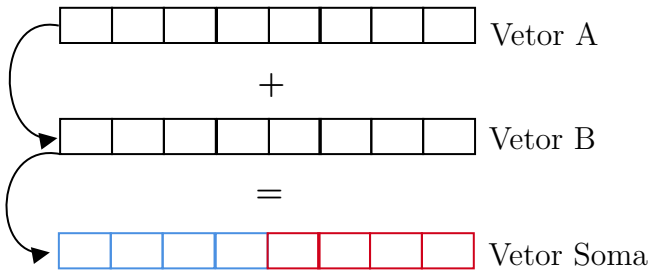
As unidades funcionais ou ULA são responsáveis pelas operações lógicas, de inteiros e ponto flutuante. As ULAs podem ser organizadas em paralelo ou implementadas com pipeline. As unidades de *load/store* realizam a comunicação entre os registradores e a

Figura 9 – Representação da execução de um algoritmo sequencial e paralelo



Fonte: Autora

Figura 10 – Representação da soma de vetores na Arquitetura SIMD



memória. Os registradores escalares são responsáveis por realizar operações sequenciais, como por exemplo testes condicionais e gerenciamento de entrada e saída.

Os processadores de propósito geral possuem além da ULA, unidades vetoriais (SIMD) que variam de tipo, dependendo do modelo do processador. As extensões da arquitetura SIMD que podem estar presentes em um processador são Multimedia Extension (MMX), Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), Streaming SIMD Extensions 4 (SSE4), Advanced Vector Extensions (AVX) e Advanced Vector Extensions 512 (AVX-512). Essas extensões compreendem um conjunto de instruções para o processador operar múltiplos dados em uma única instrução (INTEL, 2019). A tecnologia MMX opera sobre dados inteiros e empacotados. A extensão possui 8 registradores (MMX0 - MMX7) de 64 bits cada (ALVES et al., 2015). A extensão SSE foi projetada para substituir o MMX. Essa tecnologia permitiu processar dados inteiros e de ponto flutuante com 8 registradores vetoriais de 128 bits (XMMX0 - XMMX). O SSE2 substituiu MMX e o SSE. As instruções de inteiros estendem a tecnologia MMX com atualização de 64 para 128 bits. As instruções de ponto flutuante de precisão dupla permitem a execução simultânea de duas operações de ponto flutuante. A sequência do SSE2 foi o SSE3, que inclui instruções para melhorar a sincronização entre as *threads* e aplicações de mídia e jogos (INTEL, 2019). O SSE4 incluiu instruções que melhoravam o desempenho de aplicações, aumentavam o suporte a cálculos e o acesso a memória (INTEL, 2007). A tecnologia AVX é utilizada para aplicações que fazem uso de múltiplas operações de ponto flutuante. Essa extensão dispõe de registradores com 256 bits, contribuindo para processamento de imagem, áudio, vídeo simulações e modelagem 3D. O AVX-512 permite o processamento do dobro de elementos que o AVX e pode processar, em uma instrução, o que o SSE processaria em quatro. Dessa forma o AVX-512 é uma instrução SIMD aplicada em tarefas que demandam muitas instruções (INTEL, 2019).

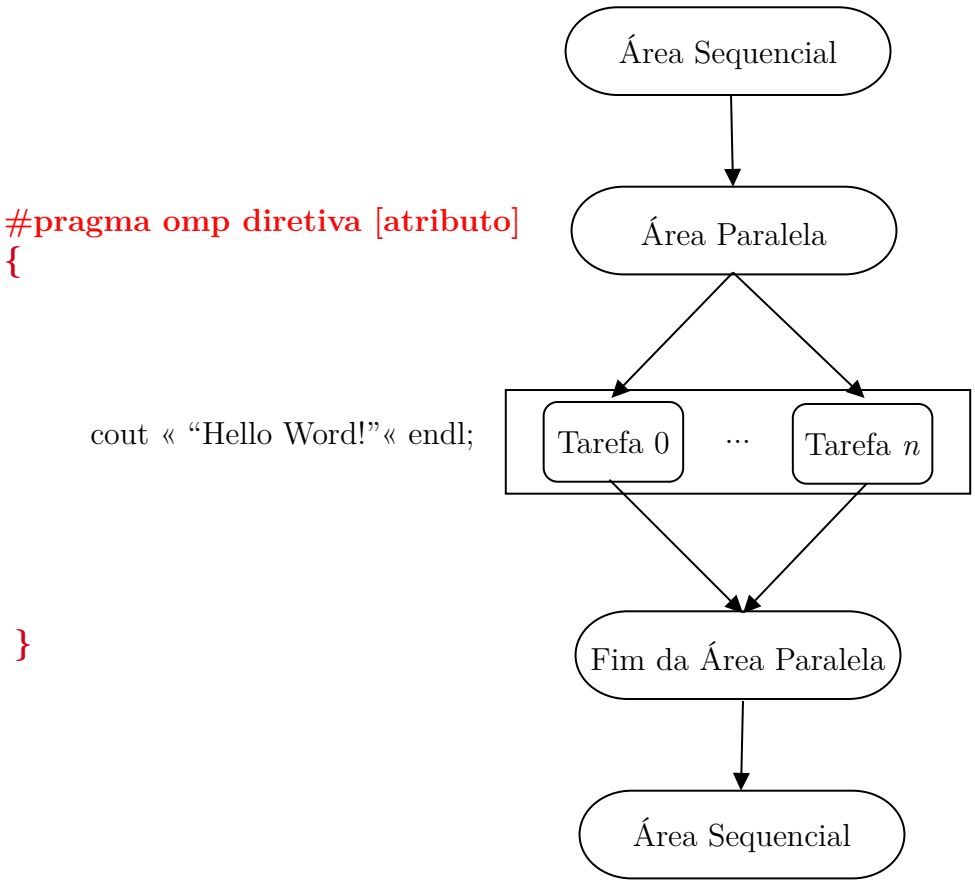
2.4.2 OpenMP

O OpenMP é uma *Application Programming Interface* (API) para programação paralela de memória compartilhada e multiplataforma disponível em C/C++ e Fortran (OPENMP, 2018). Essa API é baseada no modelo de execução *fork-join*, no qual uma *thread* mestre inicia a execução e gera *threads* de trabalho para executar em paralelo, conforme necessário ou especificado (CHAPMAN; MEHROTRA; ZIMA, 1998). A Figura 11 apresenta o fluxo de execução de um algoritmo que possui uma instrução OpenMP.

A API possui um conjunto de diretivas de compilação, funções e variáveis de ambiente para a programação paralela (TORELLI; BRUNO, 2004). Uma diretiva é precedida obrigatoriamente por `#pragma omp` e seguida por `[atributos]`, sendo que os atributos são opcionais.

Seguem algumas diretivas que compõem a API OpenMP (OPENMP, 2018).

Figura 11 – Fluxo de execução em OpenMP



Fonte: Adaptado de OpenMP (2008)

- `parallel`: Essa diretiva descreve que a uma área do código será executada por n *threads*, sendo n o número de *threads* especificados por um atributo e/ou variável de ambiente.
- `for`: Essa diretiva especifica que as iterações do laço de repetição serão executadas em paralelo por n *threads*.
- `parallel for`: Especifica a construção de um laço paralelo, sendo que o laço será executado por n *threads*.
- `simd`: Essa diretiva descreve que algumas iterações de um laço de repetição podem ser executadas simultaneamente por unidades vetoriais.
- `for simd`: Essa diretiva especifica que um laço pode ser dividido em n *threads* que executam algumas iterações simultaneamente por unidades vetoriais.

Em sequência são apresentados alguns atributos da API OpenMP (OPENMP, 2018). Para todos os casos, `lista` representa uma ou mais variáveis.

- `private (lista)`: Esse atributo informa que o bloco paralelo possui variáveis privadas para cada uma das *n threads*. As variáveis do bloco que não são informadas na lista, são públicas.

- `shared (lista)`: O atributo especifica que as variáveis são públicas e compartilhadas entre as *n threads*.

- `num_threads(int)`: Esse atributo determina o número *n* de *threads* utilizadas no bloco paralelo. Sendo que o valor de *n* é válido apenas para o bloco em que foi definido.

- `reduction (operador: lista)`: A redução é utilizada para executar cálculos em paralelos. Sendo que cada *thread* tem seu valor parcial, ao final da região paralela o valor final da variável é atualizado com os cálculos parciais. O operador pode ser, por exemplo $+$, $-$, $*$, max e min.

- `nowait`: Uma diretiva OpenMP possui uma barreira implícita ao seu fim, com o objetivo de garantir a sincronização. Entretanto a diretiva `nowait` omite a existência dessa barreira. Dessa forma, as *threads* não ficam em espera até que as demais também terminem o trabalho.

As variáveis de ambiente do OpenMP especificam características que afetam a execução dos programas. Seguem algumas variáveis (OPENMP, 2018):

- `OMP_NUM_THREADS`: Especifica o número *n* de *threads* utilizados nos blocos paralelos do algoritmo.

- `OMP_THREAD_LIMIT`: Descreve o número máximo de *threads*.

- `OMP_NESTED`: Permite ativar ou desativar o paralelismo aninhado.

- `OMP_STACKSIZE`: Especifica o tamanho da pilha para *threads*.

A API OpenMP é aplicada neste trabalho para paralelizar os algoritmos ABC, ACO e PSO. As diretivas utilizadas neste trabalho são apresentadas na seção 4.2.

2.5 Trabalhos Relacionados

Para a pesquisa e coleta de material foram utilizadas as bases de dados *IEEE Xplore*, *Springer Link* e *Science Direct*, pois são as principais bases indexadoras da área de computação. Os termos *parallel* (paralelo); *library* (biblioteca); *swarm* (enxame); *algorithm* (algoritmo); *optimization* (otimização); *swarm intelligence* (inteligência de enxame); *framework*; *bioinspired* (bio-inspirado) compõem a string de busca aplicada nas bases selecionadas. As questões de pesquisa que conduzem o estudo são as seguintes:

1. Quais são as bibliotecas bio-inspiradas disponíveis?
2. Quais algoritmos as bibliotecas implementam?
3. Os algoritmos são paralelos?
4. Os algoritmos paralelos em C++ são desenvolvidos em qual API?

A pesquisa nas bases de dados através da string de busca identificou algumas bibliotecas, em seguida o GitHub[3] foi explorado com o objetivo de encontrar trabalhos relacionados. Para filtrar esses materiais foram estabelecidos critérios de exclusão com o objetivo de selecionar trabalhos atuais e que façam parte de congressos relevantes para a área de pesquisa, são eles:

1. Data de publicação inferior aos anos 2000;
2. Publicações em línguas distintas de português ou inglês;
3. Fora do escopo da pesquisa;
4. Duplicados;
5. Indisponíveis nas bases de dados.

A string de busca utilizada nas bases indexadoras selecionadas foi *(optimization OR parallel OR algorithm OR framework) AND (library) AND (swarm OR bioinspired OR "swarm intelligence")*. Aplicando os critérios de exclusão aos 384 artigos e livros retornados pelas bases indexadoras restaram 4 para a revisão de literatura.

O artigo Silva et al. (2018) apresenta uma análise comparativa entre um conjunto de 25 bibliotecas de otimização disponíveis na literatura, identifica os pontos fortes e fracos de cada uma. Verifica possíveis carências e possibilidades de trabalhos futuros, uma vez que, busca disponibilizar um material completo sobre as ferramentas de otimização para pesquisadores que investigam o estado da arte.

> A busca por novos métodos, assim como identificar as melhores técnicas para resolver problemas de otimização, sem refazer a aplicação, tem orientado pesquisadores para o desenvolvimento de estruturas como *frameworks*. Os *frameworks* fornecem uma estrutura de recursos genéricos para a solução de problemas de um domínio específico, facilitando o desenvolvimento de novas aplicações (SILVA et al., 2018).

Silva et al. (2018) amplia a comparação apresentada em Parejo et al. (2012), com questões de pesquisa associadas a estruturas multiagentes usadas em meta-heurísticas. Com o objetivo de identificar as características necessárias para o desenvolvimento e os *frameworks* disponíveis na literatura. O artigo destaca como carências na literatura

[3] Disponível em: https://github.com/

ferramentas de otimização, como análise estatística, autoajuste de parâmetros e interfaces gráficas.

Das vinte e cinco ferramentas avaliadas em Silva et al. (2018) apenas oito possuem implementações bio-inspiradas, são elas ECJ[4], *Evolutionary Algorithms Framework* (EvA2)[5], *Framework for Metaheuristic Optimization* (FOM)[6], *Learning-Based Multi-agent System* (LBMAS), *Parallel and Distributed Evolving Objects* (ParadisEO)[7], MALLBA[8], Opt4[9], e *Swarms of Metaheuristic Agents (SMA).*

O *framework* ECJ foi desenvolvido por White (2012) na linguagem de programação Java para solucionar problemas através da computação evolucionária. Essa biblioteca dispõem de interface gráfica, suporte para computação distribuída e paralelismo. Esse *framework* implementa os algoritmos GA, PSO, DE e outros (LUKE, 2010).

A biblioteca EvA2 foi desenvolvida por Kronfeld, Planatscher e Zell (2010) a partir da ferramenta JavaEvA. Essa biblioteca é implementada na linguagem de programação Java, com interface gráfica e arquitetura cliente-servidor. Os algoritmos implementados são de computação evolutiva, por exemplo ES, SA, DE e PSO.

FOM é uma ferramenta orientada a objeto para otimização usando meta-heurísticas. Essa ferramenta possui interface gráfica e possibilita que os parâmetros de um algoritmo possam ser ajustados durante cada iteração. São exemplos de algoritmos implementados pela ferramenta SA, ACO e ES (PAREJO et al., 2003).

LBMAS é um sistema para solucionar problemas de otimização combinatória. Esse sistema permite a colaboração entre agentes de uma população e dois arquivos públicos - um para armazenar soluções factíveis e outro para soluções geradas pelos agentes no espaço de busca. Apesar da população compartilhar experiências, a execução é sequencial. Logo para alcançar boas soluções o algoritmo deve iterar durante um longo período de tempo. São alguns exemplos de implementações GA, DE, SA, *Tabu Search* (TS) e ACO, entretanto não há uma versão para download disponível.

A biblioteca MALLBA soluciona problemas de otimização através de métodos implementados em C++. Ela também oferece paralelismo (*POSIX Threads* (Pthreads) e OpenMP) e computação distribuída (ALBA et al., 2002). A biblioteca é *open source* e possui implementação dos algoritmos PSO, ACO, GA, TS e outros (ASTORGA et al., 2018).

Opt4j é um *framework* para computação evolucionária desenvolvido em linguagem Java. O Opt4j permite a execução paralela e distribuída dos algoritmos DE, PSO e SA (SILVA et al., 2018).

[4] Disponível em http://cs.gmu.edu/∼eclab/projects/ecj/
[5] Disponível em http://www.ra.cs.unituebingen.de/software/EvA2/
[6] Disponível em http://www.isa.us.es/fom
[7] Disponível em http://paradiseo.gforge.inria.fr/
[8] Disponível em http://neo.lcc.uma.es/mallba/easy-mallba/
[9] Disponível em http://opt4j.sourceforge.net/

A biblioteca *Parallel Global Multiobjective Optimizer* (PaGMO)[10] é *open source*, implementa os algoritmos ABC, PSO, SA, GA e outros na linguagem C++. A API *Message Passing Interface* (MPI) é utilizada para paralelizar a execução (BISCANI; STROE, 2018).

O *framework* ParadisEO possui diversas técnicas para solucionar um problema, como por exemplo possibilidade de uso da *Graphics Processing Unit* - Unidade de Processamento Gráfico (GPU) que não é citada em outros *frameworks* identificados na literatura. Esse *framework* utiliza as bibliotecas paralelas MPI e PThreads. Para explorar os recursos dessa biblioteca é preciso possuir licença. ParadisEO é organizado em cinco módulos (SILVA et al., 2018):

1. ParadisEO-EO: Desenvolvimento de meta-heurísticas de base populacional;
2. ParadisEO-MO: Para meta-heurísticas de trajetória ou baseado em um único ponto;
3. ParadisEO-MOEO: Técnicas evolucionárias para otimização multiobjetivo;
4. ParadisEO-PEO: Meta-heurísticas paralelas e distribuídas;
5. ParadiseEO-MO-GPU: Explora a GPU, focando em meta-heurísticas de trajetória.

PyGMO[11] é uma biblioteca *open source* que fornece algoritmos como o PSO, ABC e outros na linguagem de programação Python. Os algoritmos e métodos não são paralelos, as topologias de rede *Hypercube, Ring, Barabasi-Albert,Watts-Strogatz e Erdos-Renyi*, são usadas para definir as rotas de migração de boas soluções entre os cores (IZZO; BISCANI, 2015).

A proposta SMA apresenta os algoritmos de inteligência de enxame ABC, PSO e ESA, esses são desenvolvidos na linguagem de programação C++ (SILVA et al., 2018). Entretanto não há uma versão disponível para download.

A Tabela 1 apresenta as bibliotecas que foram identificadas na literatura, a linguagem de programação, os algoritmos que elas implementam, se aplicam paralelismo e se necessitam de licença, assim respondendo as questões de pesquisa 1, 2 e 3. A quarta questão é apresentada na Tabela 2 que lista os *frameworks* implementados na linguagem C++ e as respectivas interfaces de programação paralela.

2.6 Discussão dos Trabalhos Relacionados

A revisão da literatura sobre o estado da arte apresentada na seção 2.5 mostra materiais atuais sobre as bibliotecas de computação natural, disponíveis para a comunidade de desenvolvedores.

[10] Disponível em: http://esa.github.io/pagmo/index.html
[11] Disponível em: http://esa.github.io/pygmo/index.html

Tabela 1 – Bibliotecas de Computação Natural.

Framework	**Linguagem**	**Algoritmos**	**Paralelismo**	**Licença**
ECJ	Java	GA, PSO e DE	Sim	Open Source
Eva2	Java	ES, DE e PSO	Sim	LGPL
FOM	Java	SA, ACO e ES	Não	LGPL
LBMAS	-	GA, DE, SA e ACO	-	-
MALLBA	C++	SA, GA, PSO e ACO	Sim	Open Source
Opt4	Java	DE, PSO e SA	Sim	LGPL
ParadisEO	C++	PSO	Sim	CECILL
PaGMO	C++	ABC, PSO, SA e GE	Sim	Open Source
PyGMO	Python	PSO, ABC, GA e ACO	Não	Open Source
SMA	C++	ABC, PSO e ESA	-	-

Fonte: Silva et al. (2018)

Tabela 2 – Interfaces Paralelas.

Framework	**MPI**	**OpenMP**	**PThreads**
MALLBA	Não	Sim	Sim
PaGMO	Sim	Não	Não
ParadisEO	Sim	Não	Sim

Fonte: Silva et al. (2018)

As bibliotecas bio-inspiradas identificadas na literatura são ECJ, Eva2, FOM, MALLBA, Opt4, ParadisEO, PaGMO e PyGMO respectivamente descritas e caracterizadas na seção 2.5. Essas bibliotecas possuem algoritmos evolutivos e de inteligência de enxame são eles *Artificial Bee Colony* (ABC), *Ant Colony Optimization(ACO)*, *Bee colonies Optimization* (BCO), *Differential Evolution (DE)*, *Evolution Strategies* (ES), *Evolutionary Simulated Annealing* (ESA), *Genetic Algorithms* (GA), *Particle Swarm Optimization* (PSO), *Simulated Annealing* (SA) e *Swarms of Metaheuristic Agent* (SMA).

No estudo buscou-se identificar quais destas bibliotecas bio-inspiradas possuem algoritmos paralelos. Na Tabela 1 são descritas todas as bibliotecas identificadas na literatura sendo que as paralelas são apontadas na quarta coluna. Independente de linguagem de programação as bibliotecas paralelas são ECJ, Eva2, MALLBA, Opt4, ParadisEO e PaGMO.

O objetivo da revisão também foi identificar a API utilizada somente nas bibliotecas desenvolvidas na linguagem de programação C++. A Tabela 2 apresenta as bibliotecas e as respectivas interfaces utilizadas. A biblioteca MALLBA possui algoritmos implementados nas APIs OpenMP e PThreads, já na biblioteca PaGMO a interface é MPI e para a ParadisEO os recursos são as APIs MPI e Pthread.

3 ESTRATÉGIAS METODOLÓGICAS

Este capítulo descreve as estratégias definidas para alcançar os objetivos do trabalho. Ele detalha a implementação dos algoritmos, apresentando as ferramentas utilizadas para auxiliar o desenvolvimento. Os testes são descritos, a fim de mostrar como ambas as versões sequencial e paralela foram comparadas. Por fim, apresentamos a forma com que os resultados serão exibidos.

3.1 Desenvolvimento da Biblioteca

A biblioteca bio-inspirada foi desenvolvida na linguagem de programação C++, com o objetivo de facilitar a programação e dispor de recursos que permitam solucionar problemas de forma eficiente. Essa linguagem é orientada a objetos logo, possibilita abstração, encapsulamento, herança e polimorfismo o que é essencial para as boas práticas de geração de código. A linguagem também oferece bibliotecas de manipulação de vetores e matrizes e para geração de valores aleatórios. Esses contribuem com a implementação dos algoritmos de inteligência de enxame. Outro recurso disponível é a compatibilidade com a API OpenMP.

Os algoritmos de computação bio-inspirada selecionados para este estudo pertencem a classe de inteligência de enxame, logo simulam o comportamento de agentes diante da colônia. Foram escolhidos os algoritmos ABC, ACO e PSO, pois possuem características desejáveis para a otimização de problemas que não podem ser resolvidos ou solucionados em tempo polinomial. A estrutura dos algoritmos permite que certos blocos de código possam ser paralelizados aumentando o desempenho da aplicação. O paralelismo dos algoritmos da biblioteca foi implementado pela API OpenMP, pois é compatível com C++ e possui uma interface simples para o desenvolvimento de aplicações paralelas.

A *Integrated Development Environment* (IDE) utilizada para programação é o Atom[1] pois possui integração com o GitHub, sendo essa uma ferramenta para controle de versão. Atom também é multiplataforma, gratuito e amplamente difundido pela comunidade de programadores.

3.2 Funções de Teste

Para validar a implementação é preciso verificar as características do algoritmo através de testes. Os testes são realizados por funções que possuem propriedades diversas, para garantir que o algoritmo testado possa ou não resolver certos tipos de otimização com eficiência (YANG, 2010).

A qualidade das funções de otimização pode ser avaliada pelas propriedades: unimodal e multimodal; bidimensional e N-dimensional; contínua e não-contínua; e convexa, e não-convexa (YANG, 2010). Uma função que possui apenas um ótimo local é chamada

[1] Disponível em: https://atom.io/

de unimodal e as funções com dois ou mais ótimos locais são chamadas de multimodal. Funções sobre o espaço de duas dimensões são classificadas como bidimensional. Já as demais funções são classificadas neste trabalho como multidimensionais ou N-dimensional.

Uma função é contínua quando todos os pontos p do seu domínio D seguem a Equação 3.1 (GUIDORIZZI, 2001).

$$\lim_{x \to p} f(x) = f(p) \tag{3.1}$$

Uma função é convexa quando a reta secante que une dois pontos está acima do gráfico ou se qualquer ponto $m \in D$ segue a Equação 3.2 (GUIDORIZZI, 2001).

$$f''(m) > 0 \tag{3.2}$$

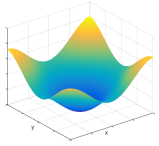
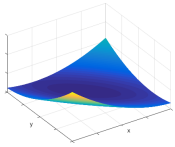
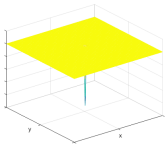
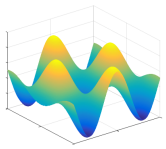
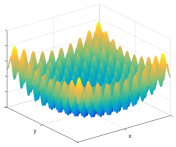
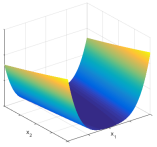
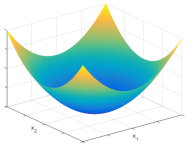
Dessa forma para testar os algoritmos implementados na biblioteca bio-inspirada foi desenvolvido um *benchmark* como uma aplicação base composta por sete funções de teste. São elas: *Alpine*, *Booth*, *Easom*, *Griewank*, *Rastrigin*, *Rosenbrock* e *Sphere*. Essas funções são apresentadas na Tabela 3. As características das funções como dimensão do agente, espaço de busca, número máximo de iterações e mínimo global foram estabelecidos de acordo com Serapiao (2009) e Ardeh (2016), sendo descritas Tabela 4.

As funções de minimização utilizadas nos testes são contínuas, as demais características são descritas na Tabela 4, de acordo com Ardeh (2016) e Molga e Smutnicki (2005).

Os resultados finais compreendem a média de 30 execuções, assim como adotado no trabalho de Couto, Silva e Barsante (2015). Uma execução compreende a geração dos arquivos de entrada, a execução sequencial e, por último, a execução paralela. Os arquivos de entrada são gerados através do método *runDetails*. Esse método cria uma pasta chamada *Inputs*, que armazena os arquivos de entrada gerados durante a execução do método. Em seguida a versão sequencial é executada pelo método *run*, sendo que a população e os demais valores aleatórios do código são inicializados com os valores dos arquivos gerados anteriormente no método *runDetails*. Por fim, a versão paralela é calculada pelo método *runParallel*. A versão paralela contém as mesmas instruções da versão sequencial, tendo apenas o acréscimo das diretivas OpenMP. Os métodos *runDetails*, *run* e *runParallel* retornam a melhor solução encontrada pelo algoritmo e o tempo de execução em segundos.

A Figura 12 apresenta o fluxo de execução das opções de execução. A execução detalhada gera o enxame aleatório, grava-o em um arquivo, executa as otimizações que são as especifidades de cada algoritmo, armazenando os demais valores aleatórios em arquivos, ao fim retorna a melhor solução encontrada. Enquanto a versão sequencial e paralela inicializam o enxame a partir do arquivo gerado anteriormente, fazem as otimizações para cada algoritmo, sendo que os valores aleatórios utilizados durante a execução são lidos de arquivos gerados pelo método anterior, ao fim retornam a melhor solução.

Tabela 3 – Funções Benchmark

Benchmark	**Função**	**Gráfico**
Alpine	$f(\mathbf{x}) = \sum i = 1^n \lvert x_i sin(x_i) + 0.1x_i \rvert$	
Booth	$f(x, y) = (x + 2y - 7)^2 + (2x + y - 5)^2$	
Easom	$f(x, y) = (-1) * \cos(x) * \cos(y) * e^{-[(x-\pi)^2-(y-\pi)^2]}$	
Griewank	$f(\mathbf{x}) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}})$	
Rastrigin	$f(\mathbf{x}) = 10n + \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i))$	
Rosenbrock	$f(\mathbf{x}) = \sum_{i=1}^{n}[b(x_{i+1} - x_i^2)^2 + (a - x_i)^2]$	
Sphere	$f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2$	

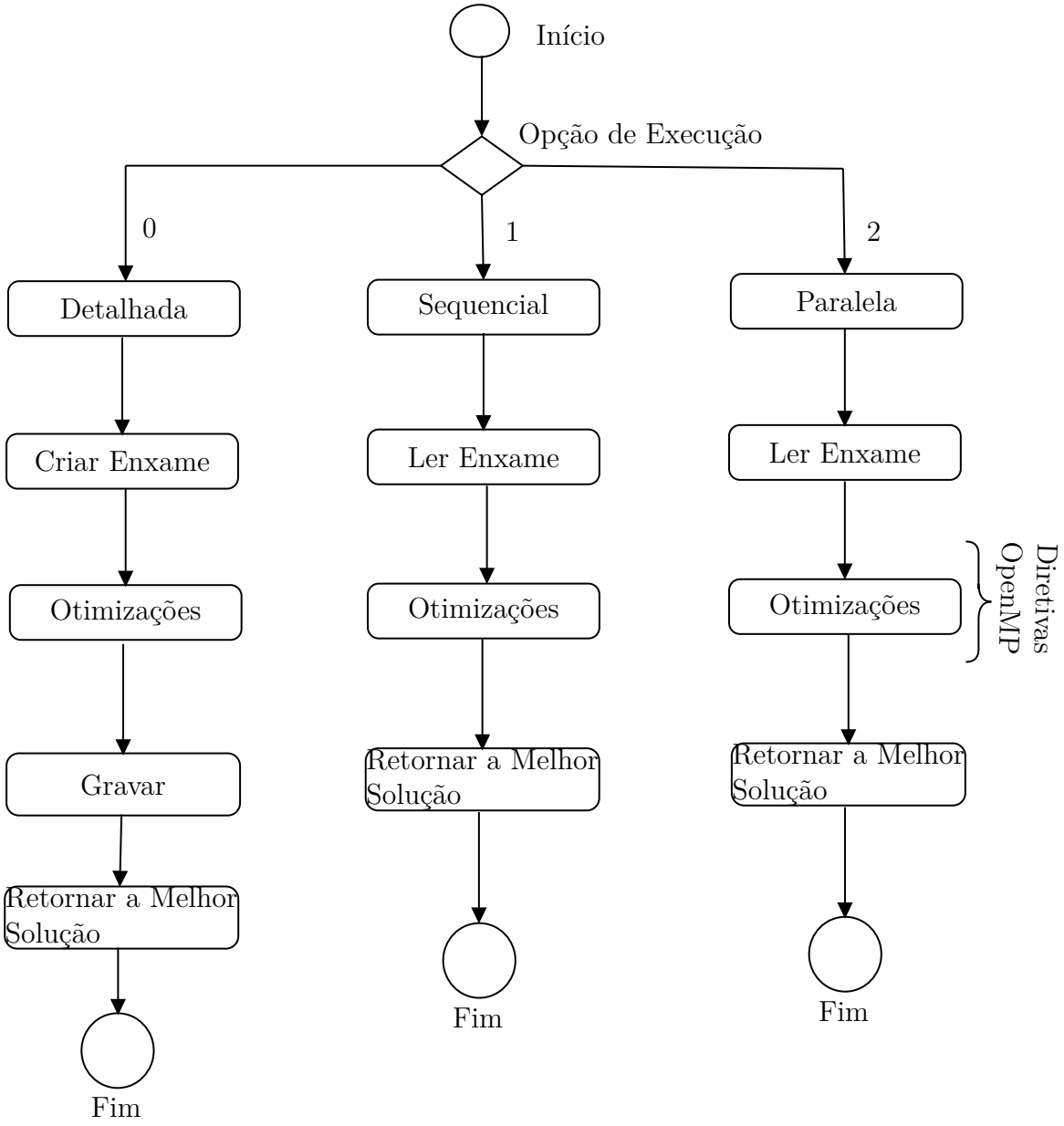
Fonte: Ardeh (2016)

Tabela 4 – Parâmetros das Funções *Benchmark*

Benchmark	**Dimensões**	**Limite de Busca**	**Iterações**	**Mínimo Global**	**Multimodal**	**Convexo**
Alpine	10	[0,10]	3000	f(0,...,0) = 0	Sim	Não
Booth	2	[-10,10]	1000	f(1,3)=0	Não	Sim
Easom	2	[-100,100]	1000	$f(\pi, \pi)$ = -1	Sim	Não
Griewank	50	[-100,100]	5000	f(0,...,0) = 0	Não	Não
Rastrigin	50	[-5.12,5.12]	5000	f(0,...,0) = 0	Sim	Sim
Rosenbrock	50	[-5,10]	5000	f(1,...,1)=0	Sim	Não
Sphere	6	[-5.12,5.12]	3000	f(0,...,0)=0	Não	Sim

Fonte: Ardeh (2016)

Figura 12 – Representação das Opções de Execução



Fonte: Autora

O lançamento dos testes são automatizados através de instruções em *shell script*. Essa linguagem permite a criação de pastas para armazenar os arquivos de saída de cada teste e a criação de *loops* sobre uma linha de comando, permitindo que ocorra o lançamento de 30 repetições sobre cada passo da execução. A linguagem também possibilita o redirecionamento da saída dos métodos para um arquivo. Dessa forma os resultados das execuções são armazenados em um arquivo do tipo `.csv`, com o nome da função, onde cada linha do arquivo corresponde a solução e o tempo de execução mensurado.

Os resultados obtidos são expressos em gráficos de colunas para representar o tempo de execução em suas respectivas versões, sequencial e paralela. O gráfico apresenta sob cada par de colunas o ganho de desempenho em porcentagem, dado pela equação Equação 3.4.

De acordo com os estudos de Talukder (2011) e Abraham, Sanyal e Sanglikar (2010) o número de agentes responsáveis por determinar o ótimo global de uma função corresponde ao intervalo de 20 a 60 agentes. Dessa forma, as funções são testadas para uma população de 10 até 100 agentes variando de 10 em 10 unidades.

Todos os testes foram realizados na workstation da Universidade Federal do Pampa (UNIPAMPA) campus Alegrete que possui a configuração descrita na Tabela 5:

Tabela 5 – Ambiente de execução: CPU

Características	**Xeon E5-2650 (×2)**
Frequência	2.00 GHz
Núcleos	8 (×2)
Threads	16 (×2)
Cache L1	32 KB
Cache L2	256 KB
Cache L3	20 MB
Memória RAM	128 GB

3.3 Implementação do PSO

O algoritmo PSO segue os parâmetros descritos na Tabela 6. O número de iterações de cada teste é estabelecido de acordo com as dimensões da função. Funções bidimensionais utilizam 1000 iterações. Funções N-dimensionais com enxame menor que 50 adotam 3000 iterações e para funções com enxame maior ou igual a 50 foram definidas 5000 iterações (SERAPIAO, 2009).

Para a execução do algoritmo PSO são necessários 11 parâmetros, são eles: *population*, min, max, c1, c2, *steps*, *function*, *inertia*, *dimension*, [min|max] e [0|1|2]. Esses atributos representam o número de partículas da população, o valor mínimo e máximo do espaço de busca, os coeficientes social e cognitivo, o número de iterações, o nome da

Tabela 6 – Parâmetros do PSO

Variável	Valor
α_1	2
α_2	2
Inércia	0.7

Fonte: Serapiao (2009)

função que será executada, o peso da inércia, a dimensão de cada partícula, o tipo da função(minimização ou maximização) e a opção de execução, respectivamente.

O último parâmetro sinaliza se a execução será detalhada com a opção 0, se será sequencial com a opção 1 e se será paralela com a opção 2. A opção detalhada gera arquivos conforme especificados em seção 3.2. A opção detalhada cria uma pasta chamada *Inputs*, que armazena os arquivos *valuesParticleInit.txt* e *valuesPSO.txt*. O primeiro arquivo armazena as posições iniciais para todas as partículas da população definida, enquanto o segundo arquivo armazena os coeficientes social e cognitivo gerados aleatoriamente em um intervalo entre 0.0 e 1.0. A opção sequencial e paralela inicializam a população e os outros valores aleatórios a partir dos valores dos arquivos gerados anteriormente e retornam a melhor solução encontrada pelo algoritmo e o tempo de execução em segundos.

As instruções OpenMP no algoritmo PSO foram inseridas na função objetivo e na função de atualização do enxame. As funções Booth e Easom são bidimensionais, logo o laço de repetição foi substituído por uma instrução que compreende o somatório das duas dimensões, as demais funções N-dimensionais possuem um laço de repetição. Esse laço é precedido pela diretiva OpenMP `#pragma omp simd reduction(+: sum)`. O outro segmento de código paralelizado é o método que atualiza a posição de cada partícula e em seguida atualiza a velocidade de cada partícula. A diretiva OpenMP aplicada é `#pragma omp simd`.

3.4 Implementação do ABC

Para a execução do algoritmo ABC são necessários 11 parâmetros, são eles: *population*, min, max, *dimension*, *limit*, *steps*, *function*, [min|max] e [0|1|2]. Esses atributos representam o número de abelhas da população, o valor mínimo e máximo do espaço de busca, a dimensão de cada abelha, o limite de tentativas de melhorar uma solução, o número de iterações, nome da função que será executada, o tipo da função(minimização ou maximização) e a opção de execução, respectivamente. O limite de tentativas para melhorar uma solução de acordo com (SERAPIAO, 2009) é 100. O número de iterações de cada teste é estabelecido da mesma forma que seção 3.3.

O último parâmetro sinaliza se a execução será detalhada com a opção 0, se será sequencial com a opção 1 e se será paralela com a opção 2. A opção detalhada cria um diretório chamado *Inputs*, que armazena os arquivos *valuesBeeInit.txt*, *sendBees.txt*,

Tabela 7 – Parâmetros do ACO

Variável	Valor	Descrição
k	50	Tamanho do Arquivo de Solução
q	0.01	Coeficiente da Equação ω
μ	0.85	Taxa de Evaporação do Feromônio

Fonte: Dorigo et al. (1996)

sendTrackBees.txt, *scoutBee.txt*. O primeiro arquivo armazena as posições iniciais para todas as abelhas da população definida, enquanto o demais arquivo armazenam os valores sorteados aleatoriamente durante a execução dos métodos.

Os métodos paralelizados no algoritmo ABC são as funções objetivos, o cálculo de custo e *fitness* para a população inicial e o cálculo de probabilidades. O paralelismo das funções objetivos ocorrem da mesma forma que no algoritmo PSO. O laço de repetição que atualiza para todo o enxame inicial o custo e o *fitness*, assim como o laço de repetição que atualiza para o enxame a nova probabilidade são paralelizados pela instrução `#pragma omp simd`.

3.5 Implementação do ACO

Para a execução do algoritmo ABC são necessários 12 parâmetros, são eles: *population*, min, max, *dimension*, k, q, *mu*, *steps*, *function*, [min|max] e [0|1|2]. Esses atributos representam o número de formigas da população, o valor mínimo e máximo do espaço de busca, a dimensão de cada formiga, o tamanho do arquivo de solução, o $q \in [0.0001; 1]$, a taxa de evaporação, o número de iterações, o tipo da função(minimização ou maximização) e a opção de execução, respectivamente. O número de iterações de cada teste é 100 (DORIGO; BIRATTARI, 2010).

A opção detalhada cria uma pasta chamada *Inputs*, que armazena os arquivos *valuesAntInit.txt*, *valuesSolution.txt*, *selectionAnt.txt*, *advanceAnt.txt*. Os dois primeiros arquivos armazenam as posições iniciais para todas as formigas da população definida e as posições iniciais do arquivo de soluções. Enquanto, os demais arquivos armazenam os valores gerados aleatoriamente durante a execução dos métodos.

O paralelismo do algoritmo ACO foi estabelecido na função objetivo, no cálculo do parâmetro ω, no cálculo de probabilidades, no método que seleciona uma formiga de acordo com a probabilidade e o método de avanço da formiga onde é realizado o somatório de σ.

3.6 Caracterização de Sucesso

Para medir o desempenho das versões dos algoritmos paralelos, foi utilizado o conceito de *speedup*(S). O *speedup* é definido como a razão entre o tempo de computação

do algoritmo serial (T_{serial}) e o tempo de computação do algoritmo paralelo ($T_{paralelo}$), dado pela Equação 3.3. O *speedup* mostra o ganho efetivo do tempo de processamento do algoritmo paralelo sobre o algoritmo serial (MORAES, 2011).

$$S = \frac{T_{serial}}{T_{paralelo}} \tag{3.3}$$

Quando o tempo paralelo é exatamente igual ao tempo sequencial, o *speedup* é igual a 1. Neste caso não há ganho de desempenho. Uma outra forma de mensurar o quanto uma versão paralela é melhor que a versão sequencial é considerar o percentual de ganho de desempenho apresentado na Equação 3.4.

$$S = \frac{T_{serial} - T_{paralelo}}{T_{paralelo}} * 100 = (\frac{T_{serial}}{T_{paralelo}} - 1) * 100 \tag{3.4}$$

3.7 Considerações do Capítulo

Nesse capítulo foram apresentados os aspectos metodológicos para o desenvolvimento do trabalho. Foram descritos os métodos adotados para a implementação e para a realização dos testes, as ferramentas utilizadas, a máquina e a forma com que os resultados serão expressos para permitir a reprodutibilidade dos experimentos.

4 ANÁLISE EXPERIMENTAL

Esse capítulo tem por objetivo expor e analisar os resultados obtidos pelos testes realizados. Os resultados serão apresentados em forma tabular e gráfica, sendo respectivamente comentados.

4.1 Implementação da Biblioteca Bio-inspirada

O código-fonte dos algoritmos ABC, ACO e PSO está organizado em três diretórios, *include*, *main* e *src*. O primeiro diretório contém os arquivos de extensão *hpp*, ou seja, os arquivos de cabeçalho que definem as classes e métodos. O diretório *main* contém o arquivo de inicialização da execução, a implementação das equações de benchmark utilizadas nos testes e um arquivo auxiliar que contém os métodos para a execução das versões sequencial (métodos sem as diretivas OpenMP) e paralela (que contém as diretivas OpenMP). O diretório *src* contém os códigos-fonte implementados. Além dos diretórios, a biblioteca contém um script de lançamento para os testes, chamado de *run.sh* e um arquivo *README*, documento de texto com informações e detalhes de utilização da biblioteca. A Tabela 8 mostra a estrutura de arquivos relacionada para cada algoritmo.

Tabela 8 – Estrutura da Implementação

Diretório	**ABC**	**ACO**	**PSO**
include	abc.hpp	aco.hpp	pso.hpp
	bee.hpp	ant.hpp	particle.hpp
	benchmarking.hpp	benchmarking.hpp	benchmarking.hpp
main	main.cpp	main.cpp	main.cpp
	benchmarking.cpp	benchmarking.cpp	benchmarking.cpp
	auxiliar.cpp	auxiliar.cpp	auxiliar.cpp
src	abc.cpp	aco.cpp	pso.cpp
	bee.cpp	ant.cpp	particle.cpp
.	run.sh	run.sh	run.sh
	README.md	README.md	README.md

Para utilizar a biblioteca, o usuário deve instanciar um objeto que corresponde a classe do algoritmo escolhido. Esse objeto pode ser executado pelos métodos *run*, *runParallel* e *runDetails*, os quais são, respectivamente, a execução sequencial, a execução paralela e a execução detalhada. A versão disponível da biblioteca não lê dados de arquivos. A biblioteca apenas pode gerá-los quando a execução detalhada for selecionada.

O arquivo *README* é disponibilizado para exemplificar como compilar e testar o código. O documento destaca a necessidade da *flag* `-fopenmp` para a execução paralela, detalhando para cada algoritmo as variáveis necessárias para instanciar os objetos de cada classe, assim como as opções de execução e as opções de testes. Os arquivos *README* dos

algoritmos PSO, ABC e ACO são apresentados no Apêndice A, Apêndice B e Apêndice C, respectivamente.

Após a implementação sequencial da biblioteca e verificação do código, foram feitos testes para avaliar o tempo de execução das implementações. Desta forma, foi possível obter um *baseline* o qual foi tomado como referência para efeito de comparação dos tempos de execução das implementações paralelas.

Um dos objetivos específicos ao propor uma biblioteca bio-inspirada é que a mesma explorasse algum tipo de paralelismo. Inicialmente, avaliamos a possibilidade da exploração de diretivas OpenMP em laços paralelos usando `#pragma omp parallel for`. Ao analisar os laços, observou-se que a granularidade das tarefas tenderia a ser baixa, uma vez que havia pouca computação para cada iteração. Na grande maioria dos casos, a computação restringia-se a algumas operações de soma e multiplicação em uma única linha de código. Avaliando experimentalmente, os resultados paralelos mostraram-se piores que os sequenciais usando laços paralelos. Diante disso, optou-se em explorar o paralelismo das instruções SIMD, através de `#pragma omp simd`. Os resultados de performance são apresentados na próxima seção.

4.2 Avaliação Preliminar das Soluções

Essa seção apresenta os resultados obtidos nos testes de performance dos algoritmos da biblioteca. Depois de analisar os códigos-fontes e identificar segmentos do mesmo com potencial de paralelismo, foram realizados testes envolvendo a diretiva `#pragma omp parallel for`, com 2 *threads*. Testes com 4, 6 e 8 *threads* também foram realizados, mas os resultados foram ainda piores.

Os testes realizados com a diretiva anteriormente citada apresentaram resultados ruins, sendo que para alguns casos a versão paralela é cerca de 90% pior que a sequencial. A diretiva `#pragma omp parallel for` apresentou apenas um resultado positivo no teste do algoritmo PSO na função *Alpine*.

Os resultados da Tabela 9, Tabela 10 e Tabela 11 apresentam o ganho de desempenho em porcentagem para os testes descritos nesta seção, sendo que os valores destacados em negrito representam a melhor taxa de ganho. A população utilizada nos testes é de 20 agentes.

Devido aos resultados ruins, outras diretivas foram investigadas a fim de obter uma redução no tempo de execução. A diretiva `#pragma omp simd` foi utilizada, pois, essa diretiva permite usar unidades vetoriais, onde um bloco de instruções é executado simultaneamente. Como uma instrução é executada sobre diferentes dados, para garantir a eficiência desse método de paralelismo os endereços acessados no laço de repetição devem estar consecutivos em memória. Desse modo a alocação de memória contígua é aplicada na implementação dos algoritmos sobre os vetores dos dados. Dos resultados obtidos pela diretiva `#pragma omp simd` apenas 3 testes obtiveram resultados negativos, ou seja, piora

Tabela 9 – Implementação Paralela - ABC.

Função	simd	for simd	simd + parallel for	parallel for
Alpine	0,21	**1,19**	-0,23	-99,40
Booth	6,84	0,34	**6,98**	-3,89
Easom	5,67	-2,35	**6,86**	-18,71
Griewank	**0,99**	-0,11	0,83	-99,21
Rastrigin	**1,74**	0,20	1,28	-99,21
Rosenbrock	**3,41**	-0,36	2,67	-99,30
Sphere	-1,13	**2,19**	0,76	-99,57

Fonte: Autora

Tabela 10 – Implementação Paralela - ACO.

Função	simd	for simd	simd + parallel for	parallel for
Alpine	**0,04**	-0,43	-94,37	-92,96
Booth	**0,12**	-2,59	-94,39	-91,92
Easom	**0,08**	-0,66	-94,50	-92,11
Griewank	**0,31**	0,16	-96,60	-96,71
Rastrigin	0,38	**0,82**	-97,24	-97,47
Rosenbrock	**4,85**	0,09	-97,25	-97,37
Sphere	**0,04**	-0,16	-93,77	-91,97

Fonte: Autora

no tempo de execução paralelo, sendo 1 deles no algoritmo ABC na função Sphere e os outros dois no algoritmo PSO para as funções Booth e Easom.

Em sequência a diretiva `#pragma omp for simd` foi testada, essa diretiva permite vetorizar e paralelizar um laço do código. A diretiva foi testada e diversos resultados foram obtidos. Para alguns casos, o tempo de execução paralelo foi pior que o sequencial e para outros casos a diretiva resultou na melhor solução obtida diante de todos os testes realizados com diferentes diretivas.

A fim de verificar o desempenho das funções, outro teste foi realizado com a diretiva `#pragma omp simd` nas funções objetivo e nos demais métodos paralelizáveis de cada algoritmo foi aplicada a diretiva `#pragma omp parallel for` com 2 *threads*. Esse teste obteve resultados positivos quanto ao tempo de execução paralelo apenas para as funções no algoritmo ABC, com exceção da função *Alpine* que obteve resultados negativos. Os demais testes obtiveram como resultados a piora no tempo de execução paralelo.

Como é possível observar na Tabela 9, na Tabela 10 e na Tabela 11 os testes com a diretiva `#pragma omp simd` obtiveram os melhores resultados para os três algoritmos. Dessa forma essa diretiva foi utilizada para paralelizar os algoritmos desenvolvidos neste trabalho.

Tabela 11 – Implementação Paralela - PSO.

Função	simd	for simd	simd + parallel for	parallel for
Alpine	**1,12**	-0,19	-0,63	1,10
Booth	-2,39	**-2,05**	-86,83	-83,22
Easom	**-0,16**	-5,20	-81,95	-81,00
Griewank	**0,84**	0,53	-58,14	-61,04
Rastrigin	1,70	**7,81**	-60,08	-61,56
Rosenbrock	**0,66**	0,31	-59,12	-62,69
Sphere	**7,19**	-11,75	-77,77	-84,97

Fonte: Autora

4.3 Tempos de Execução

Após a implementação e os testes referentes as diretivas *OpenMP* da biblioteca, foram realizados testes para avaliar o desempenho das versões sequencial e paralela submetidas a diferentes populações. Na Tabela 12, Tabela 13 e Tabela 14 são apresentados, para os 7 casos de teste, o tempo da execução sequencial e paralelo, em segundos, o ganho de desempenho (Ganho de Desemp. (%)) em porcentagem e a média das soluções para cada população (Pop.). Além disso, para os tempos de execução, são mostrados, entre parênteses, o desvio padrão. Como a versão sequencial e a versão paralela são submetidas ao mesmo caso de teste, a solução obtida é igual. Os valores de ganho de desempenho destacados em negrito representam o melhor ganho efetivo para o caso de teste em questão. Os valores na coluna média das soluções destacados em itálico correspondem a solução ótima, enquanto que os destacados em negrito correspondem a melhor solução obtida para a função.

Tabela 12 – PSO - Resultados Obtidos nos Testes

Função	Pop.	Sequencial(s) (Desvio Padrão)	Paralelo(s) (Desvio Padrão)	Ganho de Desemp.(%)	Média das Soluções
Alpine	10	0,09867 (0.002)	0,10532 (0.003)	-6,31	3,66E+02
	20	0,11408 (0.015)	0,11225 (0.009)	1,63	8,22E+01
	30	0,17399 (0.023)	0,16602 (0.015)	**4,80**	3,68E+00
	40	0,22202 (0.024)	0,21623 (0.012)	2,68	3,59E+00
	50	0,27602 (0.025)	0,27274 (0.016)	1,20	4,12E+00
	60	0,32494 (0.029)	0,32132 (0.020)	1,12	7,07E+01
	70	0,37462 (0.034)	0,36141 (0.011)	3,65	4,74E+02
	80	0,43086 (0.022)	0,42613 (0.024)	1,11	**2,53E+00**
	90	0,47107 (0.024)	0,48474 (0.045)	-2,82	2,86E+00
	100	0,53140 (0.029)	0,53244 (0.027)	-0,19	3,09E+00

Booth	10	0,03163 (0.001)	0,03302 (0.001)	-4,19	1,37E-02
	20	0,02156 (0.002)	0,02218 (0.001)	-2,78	***0,0E+00***
	30	0,02952 (0.002)	0,03098 (0.002)	-4,72	3,88E-04
	40	0,03646 (0.003)	0,03708 (0.003)	-1,67	1,88E-02
	50	0,04674 (0.003)	0,04667 (0.003)	0,16	2,81E-02
	60	0,05468 (0.004)	0,05602 (0.004)	-2,39	4,83E-03
	70	0,06359 (0.005)	0,06348 (0.005)	0,17	6,02E-04
	80	0,07150 (0.005)	0,07081 (0.005)	**0,97**	8,02E-05
	90	0,07828 (0.007)	0,07832 (0.007)	-0,04	3,63E-04
	100	0,08709 (0.007)	0,08705 (0.007)	0,04	6,59E-03
Easom	10	0,03466 (0.001)	0,03843 (0.0008)	-9,81	-1,67E-01
	20	0,02662 (0.002)	0,02669 (0.001)	-0,27	-5,67E-01
	30	0,03612 (0.003)	0,03533 (0.003)	2,23	-5,33E-01
	40	0,04642 (0.004)	0,04653 (0.003)	-0,25	-6,65E-01
	50	0,05572 (0.005)	0,05546 (0.005)	0,48	-7,67E-01
	60	0,06610 (0.006)	0,06621 (0.006)	-0,16	-7,77E-01
	70	0,07702 (0.006)	0,07701 (0.006)	0,01	-8,00E-01
	80	0,08862 (0.005)	0,08832 (0.005)	0,34	**-9,33E-01**
	90	0,09488 (0.008)	0,09493 (0.008)	-0,06	-9,01E-01
	100	0,10261 (0.012)	0,07815 (0.006)	**31,29**	-9,29E-01
Griewank	10	0,44853 (0.029)	0,44200 (0.023)	1,48	2,71E+02
	20	0,89392 (0.059)	0,88235 (0.055)	1,31	1,89E+02
	30	1,28730 (0.081)	1,28037 (0.092)	0,54	1,96E+02
	40	1,73427 (0.114)	1,69519 (0.124)	**2,31**	1,20E+02
	50	2,12005 (0.115)	2,08841 (0.116)	1,52	1,46E+02
	60	2,53792 (0.172)	2,51682 (0.171)	0,84	1,62E+02
	70	2,99390 (0.176)	2,93386 (0.177)	2,05	1,24E+02
	80	3,39981 (0.24)	3,35639 (0.25)	1,29	**9,11E+01**
	90	3,80303 (0.304)	3,73154 (0.274)	1,92	1,03E+02
	100	4,22303 (0.259)	4,16001 (0.278)	1,52	1,11E+02
Rastrigin	10	0,40481 (0.30)	0,39833 (0.21)	1,63	4,91E+02
	20	0,79317 (0.033)	0,78213 (0.020)	1,41	4,16E+02
	30	1,19233 (0.054)	1,16023 (0.047)	**2,77**	4,17E+02
	40	1,58993 (0.088)	1,57249 (0.087)	1,11	3,99E+02
	50	1,99148 (0.115)	1,94771 (0.110)	2,25	3,55E+02
	60	2,37402 (0.108)	2,33439 (0.110)	1,70	3,59E+02
	70	2,77582 (0.162)	2,72237 (0.174)	1,96	**3,28E+02**
	80	3,15126 (0.143)	3,08086 (0.133)	2,28	3,61E+02
	90	3,53387 (0.180)	3,46843 (0.183)	1,89	3,55E+02

	100	3,93809 (0.256)	3,87599 (0.245)	1,60	3,28E+02
Rosenbrock	10	0,35422 (0.042)	0,34690 (0.036)	2,11	3,34E+02
	20	0,73364 (0.076)	0,71447 (0.078)	2,68	2,45E+02
	30	1,03145 (0.088)	1,01020 (0.090)	2,10	1,59E+02
	40	1,44094 (0.167)	1,41395 (0.163)	1,91	1,07E+02
	50	1,77912 (0.168)	1,72936 (0.167)	**2,88**	1,25E+02
	60	2,13645 (0.214)	2,12248 (0.275)	0,66	1,10E+02
	70	2,47694 (0.231)	2,43000 (0.236)	1,93	9,42E+01
	80	2,81688 (0.290)	2,76468 (0.286)	1,89	7,15E+01
	90	3,18033 (0.302)	3,12816 (0.315)	1,67	**6,02E+01**
	100	3,43905 (0.292)	3,36878 (0.300)	2,09	9,13E+01
Sphere	10	0,04806 (0.004)	0,04783 (0.004)	0,49	7,07E-01
	20	0,08090 (0.015)	0,07247 (0.010)	11,64	3,84E-01
	30	0,11294 (0.014)	0,09987 (0.008)	**13,09**	4,56E-01
	40	0,13766 (0.022)	0,13684 (0.023)	0,60	7,80E-02
	50	0,16845 (0.168)	0,14953 (0.149)	12,65	6,39E-02
	60	0,20110 (0.029)	0,18760 (0.020)	7,19	8,32E-02
	70	0,22268 (0.021)	0,21511 (0.019)	3,52	3,98E-04
	80	0,24892 (0.034)	0,24083 (0.022)	3,36	1,05E-01
	90	0,28778 (0.027)	0,28199 (0.024)	2,05	7,86E-02
	100	0,31722 (0.025)	0,30856 (0.023)	2,80	**3,77E-05**

Fonte: Autora

Tabela 13 – ABC - Resultados Obtidos nos Testes

Função	**Pop.**	**Sequencial(s) (Desvio Padrão)**	**Paralelo(s) (Desvio Padrão)**	**Ganho de Desemp.(%)**	**Média das Soluções**
Alpine	10	0,07202 (0,012)	0,08578 (0,002)	-16,05	3,92E-17
	20	0,13443 (0,009)	0,13144 (0,008)	2,27	2,25E-17
	30	0,19341 (0,011)	0,19775 (0,011)	-2,20	1,34E-17
	40	0,25718 (0,011)	0,25571 (0,013)	0,58	1,55E-17
	50	0,32459 (0,014)	0,31603 (0,011)	2,71	1,48E-17
	60	0,38308 (0,018)	0,38412 (0,014)	-0,27	1,49E-17
	70	0,44636 (0,014)	0,44072 (0,013)	1,28	**7,60E-18**
	80	0,51364 (0,018)	0,50907 (0,015)	0,90	1,31E-17
	90	0,58021 (0,015)	0,56382 (0,019)	**2,91**	1,15E-17
	100	0,63482 (0,015)	0,62877 (0,019)	0,96	1,21E-17

Booth	10	0,03163 (0,002)	0,03302 (0,0007)	-4,19	3,37E-15
	20	0,04578 (0,001)	0,04620 (0,001)	-0,91	1,43E-15
	30	0,08750 (0,003)	0,08757 (0,004)	-0,07	2,94E-17
	40	0,08673 (0,005)	0,10025 (0,014)	-13,49	1,18E-15
	50	0,10961 (0,012)	0,12703 (0,019)	-13,72	1,84E-16
	60	0,12904 (0,012)	0,12493 (0,008)	**3,29**	2,02E-17
	70	0,14666 (0,009)	0,14693 (0,011)	-0,19	1,76E-17
	80	0,16429 (0,003)	0,16786 (0,011)	-2,13	1,92E-16
	90	0,18695 (0,010)	0,18451 (0,006)	1,32	1,34E-17
	100	0,21016 (0,015)	0,20751 (0,015)	1,28	**1,12E-17**
Easom	10	0,03466 (0,004)	0,03843 (0,0009)	-9,81	-8,51E-01
	20	0,05537 (0,007)	0,06852 (0,001)	-19,19	-9,97E-01
	30	0,07294 (0,002)	0,07483 (0,004)	-2,53	-9,99E-01
	40	0,09976 (0,011)	0,09701 (0,009)	2,83	***-1,00E+00***
	50	0,12178 (0,014)	0,11699 (0,007)	**4,10**	***-1,00E+00***
	60	0,14223 (0,013)	0,14090 (0,009)	0,95	***-1,00E+00***
	70	0,16436 (0,010)	0,16355 (0,009)	0,50	***-1,00E+00***
	80	0,18267 (0,009)	0,18566 (0,013)	-1,61	***-1,00E+00***
	90	0,20965 (0,010)	0,20923 (0,014)	0,20	***-1,00E+00***
	100	0,23697 (0,019)	0,22963 (0,014)	3,20	***-1,00E+00***
Griewank	10	0,25695 (0,007)	0,25346 (0,011)	1,38	4,31E-03
	20	0,50932 (0,011)	0,49589 (0,011)	2,71	4,17E-04
	30	0,76533 (0,013)	0,75079 (0,018)	1,94	1,21E-03
	40	1,02243 (0,018)	0,99308 (0,023)	2,96	1,66E-06
	50	1,27300 (0,018)	1,24509 (0,032)	2,24	7,44E-10
	60	1,53226 (0,018)	1,48139 (0,023)	**3,43**	4,37E-12
	70	1,78255 (0,024)	1,78694 (0,171)	-0,25	2,86E-13
	80	2,04233 (0,036)	1,97896 (0,027)	3,20	1,10E-12
	90	2,28918 (0,023)	2,22817 (0,029)	2,74	5,53E-12
	100	2,54341 (0,030)	2,47914 (0,050)	2,59	**8,83E-14**
Rastrigin	10	0,24118 (0,013)	0,23323 (0,017)	3,41	6,68E-01
	20	0,46506 (0,011)	0,45546 (0,014)	2,11	2,93E-06
	30	0,70519 (0,023)	0,67762 (0,012)	4,07	1,26E-07
	40	0,94294 (0,057)	0,89953 (0,014)	**4,83**	1,92E-08
	50	1,15307 (0,011)	1,12196 (0,011)	2,77	8,62E-11
	60	1,39739 (0,020)	1,36587 (0,020)	2,31	4,15E-11
	70	1,64471 (0,064)	1,59407 (0,022)	3,18	1,13E-11
	80	1,87645 (0,032)	1,82897 (0,022)	2,60	**1,38E-12**

	90	2,08836 (0,023)	2,03712 (0,029)	2,52	1,73E-12
	100	2,33784 (0,064)	2,27843 (0,022)	2,61	1,83E-12
Rosenbrock	10	0,21579 (0,011)	0,20573 (0,008)	**4,89**	2,21E+03
	20	0,41981 (0,011)	0,40343 (0,010)	4,06	2,45E+01
	30	0,62739 (0,016)	0,60784 (0,017)	3,22	1,33E+03
	40	0,84039 (0,023)	0,81256 (0,023)	3,43	1,36E+01
	50	1,04267 (0,026)	0,99894 (0,17)	4,38	4,80E+02
	60	1,24508 (0,022)	1,19830 (0,017)	3,90	1,12E+01
	70	1,45066 (0,023)	1,39949 (0,023)	3,66	1,08E+01
	80	1,65306 (0,027)	1,59453 (0,036)	3,67	**1,03E+01**
	90	1,85822 (0,030)	1,80006 (0,038)	3,23	1,09E+01
	100	2,05958 (0,025)	1,99118 (0,055)	3,44	9,18E+00
Sphere	10	0,07202 (0,007)	0,08578 (0,010)	-16,05	3,92E-17
	20	0,13443 (0,010)	0,13144 (0,009)	2,27	2,25E-17
	30	0,19341 (0,005)	0,19775 (0,015)	-2,20	1,34E-17
	40	0,25718 (0,011)	0,25571 (0,010)	0,58	1,55E-17
	50	0,32459 (0,018)	0,31603 (0,009)	2,71	1,48E-17
	60	0,38308 (0,013)	0,38412 (0,017)	-0,27	1,49E-17
	70	0,44636 (0,015)	0,44072 (0,013)	1,28	**7,60E-18**
	80	0,51364 (0,031)	0,50907 (0,030)	0,90	1,31E-17
	90	0,58021 (0,022)	0,56382 (0,014)	**2,91**	1,15E-17
	100	0,63482 (0,018)	0,62877 (0,022)	0,96	1,21E-17

Fonte: Autora

Tabela 14 – ACO - Resultados Obtidos nos Testes

Função	**Pop.**	**Sequencial(s) (Desvio Padrão)**	**Paralelo(s) (Desvio Padrão)**	**Ganho de Desemp.(%)**	**Média das Soluções**
Alpine	10	0,78847 (0,018)	0,78790 (0,019)	0,07	1,78E-06
	20	1,66105 (0,018)	1,65422 (0,016)	0,41	2,43E-14
	30	2,64996 (0,048)	2,64940 (0,041)	0,02	7,20E-20
	40	3,65952 (0,021)	3,65178 (0,023)	0,21	2,28E-22
	50	4,77599 (0,144)	4,73180 (0,051)	0,93	4,35E-16
	60	5,84553 (0,018)	5,85400 (0,017)	-0,14	1,55E-23
	70	7,07270 (0,579)	6,92281 (0,262)	**2,17**	1,13E-02
	80	8,05204 (0,388)	8,05613 (0,395)	-0,05	2,30E-01
	90	9,15291 (0,596)	9,13137 (0,585)	0,24	6,39E-01
	100	10,60449 (0,427)	10,50709 (0,019)	0,93	**9,45E-26**

Booth	10	1,31359 (0,043)	1,30611 (0,043)	**0,57**	1,02E+00
	20	2,87138 (0,082)	2,86386 (0,084)	0,26	8,06E-01
	30	4,58436 (0,136)	4,66987 (0,419)	-1,83	**2,33E-01**
	40	6,37379 (0,170)	6,39180 (0,303)	-0,28	6,60E-01
	50	3,41968 (0,113)	3,42916 (0,126)	-0,28	5,68E-01
	60	4,19850 (0,125)	4,19678 (0,126)	0,04	9,68E-01
	70	5,03402 (0,215)	5,02415 (0,150)	0,20	4,48E-01
	80	5,89722 (0,197)	5,87612 (0,171)	0,36	8,86E-01
	90	6,79006 (0,253)	6,75888 (0,201)	0,46	5,87E-01
	100	7,63613 (0,229)	7,60099 (0,215)	0,46	5,39E-01
Easom	10	0,52652 (0,058)	0,53320 (0,057)	-1,25	-8,10E-01
	20	1,17133 (0,050)	1,16929 (0,045)	0,17	-8,68E-01
	30	1,82753 (0,085)	1,82508 (0,079)	0,13	-8,63E-01
	40	2,56079 (0,096)	2,56539 (0,094)	-0,18	-8,80E-01
	50	3,33772 (0,101)	3,35935 (0,130)	-0,64	-9,00E-01
	60	4,12799 (0,113)	4,13068 (0,108)	-0,07	-9,19E-01
	70	5,00640 (0,173)	4,98475 (0,168)	**0,43**	**-9,59E-01**
	80	5,75939 (0,167)	5,78596 (0,172)	-0,46	-9,27E-01
	90	6,64365 (0,253)	6,62960 (0,201)	0,21	-9,57E-01
	100	7,51052 (0,200)	7,50886 (0,203)	0,02	**-9,59E-01**
Griewank	10	0,78847 (0,019)	0,79258 (0,016)	-0,52	2,77E-15
	20	1,50064 (0,014)	1,49634 (0,014)	0,29	5,47E-07
	30	2,18518 (0,027)	2,18338 (0,030)	0,08	***0,00E+00***
	40	2,88998 (0,092)	2,91635 (0,158)	-0,90	5,75E-02
	50	3,68380 (0,376)	3,60668 (0,219)	**2,14**	3,29E-16
	60	4,33944 (0,047)	4,33144 (0,049)	0,18	3,32E-02
	70	5,27361 (0,373)	5,27561 (0,370)	-0,04	4,39E-02
	80	6,24209 (0,255)	6,24011 (0,263)	0,03	1,73E-01
	90	6,98669 (0,477)	6,97177 (0,389)	0,21	3,23E-02
	100	7,86294 (0,390)	7,86217 (0,385)	0,01	8,17E-02
Rastrigin	10	0,58587 (0,028)	0,58993 (0,030)	-0,69	3,87E+03
	20	1,27531 (0,110)	1,27069 (0,107)	0,36	3,44E+03
	30	2,04009 (0,170)	2,03434 (0,175)	0,28	3,63E+03
	40	2,83503 (0,223)	2,81759 (0,213)	0,62	5,32E+01
	50	3,73149 (0,345)	3,71249 (0,349)	0,51	**4,39E+01**
	60	4,46356 (0,235)	4,44859 (0,233)	0,34	5,20E+01
	70	5,31229 (0,342)	5,38248 (0,319)	-1,30	4,90E+01
	80	6,26472 (0,384)	6,25342 (0,383)	0,18	4,63E+01

	90	7,42855 (0,625)	7,37910 (0,556)	**0,67**	4,61E+01
	100	8,07091 (0,348)	8,09436 (0,371)	-0,29	5,19E+01
Rosenbrock	10	0,58711 (0,030)	0,58470 (0,023)	0,41	3,29E+05
	20	1,31370 (0,160)	1,23824 (0,050)	**6,09**	4,49E+05
	30	1,95184 (0,092)	1,95550 (0,103)	-0,19	5,39E+05
	40	2,72391 (0,077)	2,72664 (0,077)	-0,10	**6,91E+02**
	50	3,49656 (0,144)	3,48343 (0,143)	0,38	8,59E+02
	60	4,34323 (0,110)	4,33634 (0,110)	0,16	8,42E+02
	70	5,16949 (0,150)	5,16581 (0,153)	0,07	8,98E+02
	80	6,05081 (0,187)	6,05801 (0,193)	-0,12	7,49E+02
	90	6,92485 (0,247)	6,92708 (0,247)	-0,03	7,50E+02
	100	7,84981 (0,262)	7,90563 (0,343)	-0,71	7,61E+02
Sphere	10	0,78122 (0,019)	0,77833 (0,024)	0,37	1,99E-17
	20	1,63719 (0,018)	1,63726 (0,019)	0,00	1,60E-32
	30	2,60993 (0,017)	2,60712 (0,16)	0,11	1,91E-40
	40	3,61078 (0,018)	3,62109 (0,035)	-0,28	6,33E-48
	50	4,67856 (0,012)	4,69605 (0,053)	-0,37	4,23E-54
	60	5,79508 (0,019)	5,81281 (0,105)	-0,30	1,19E-54
	70	6,88357 (0,019)	6,90073 (0,064)	-0,25	2,62E-55
	80	8,02584 (0,016)	8,02994 (0,074)	-0,05	2,50E-52
	90	9,24908 (0,364)	9,19905 (0,120)	**0,54**	4,44E-56
	100	10,40360 (0,214)	10,36808 (0,017)	0,34	**2,89E-56**

Fonte: Autora

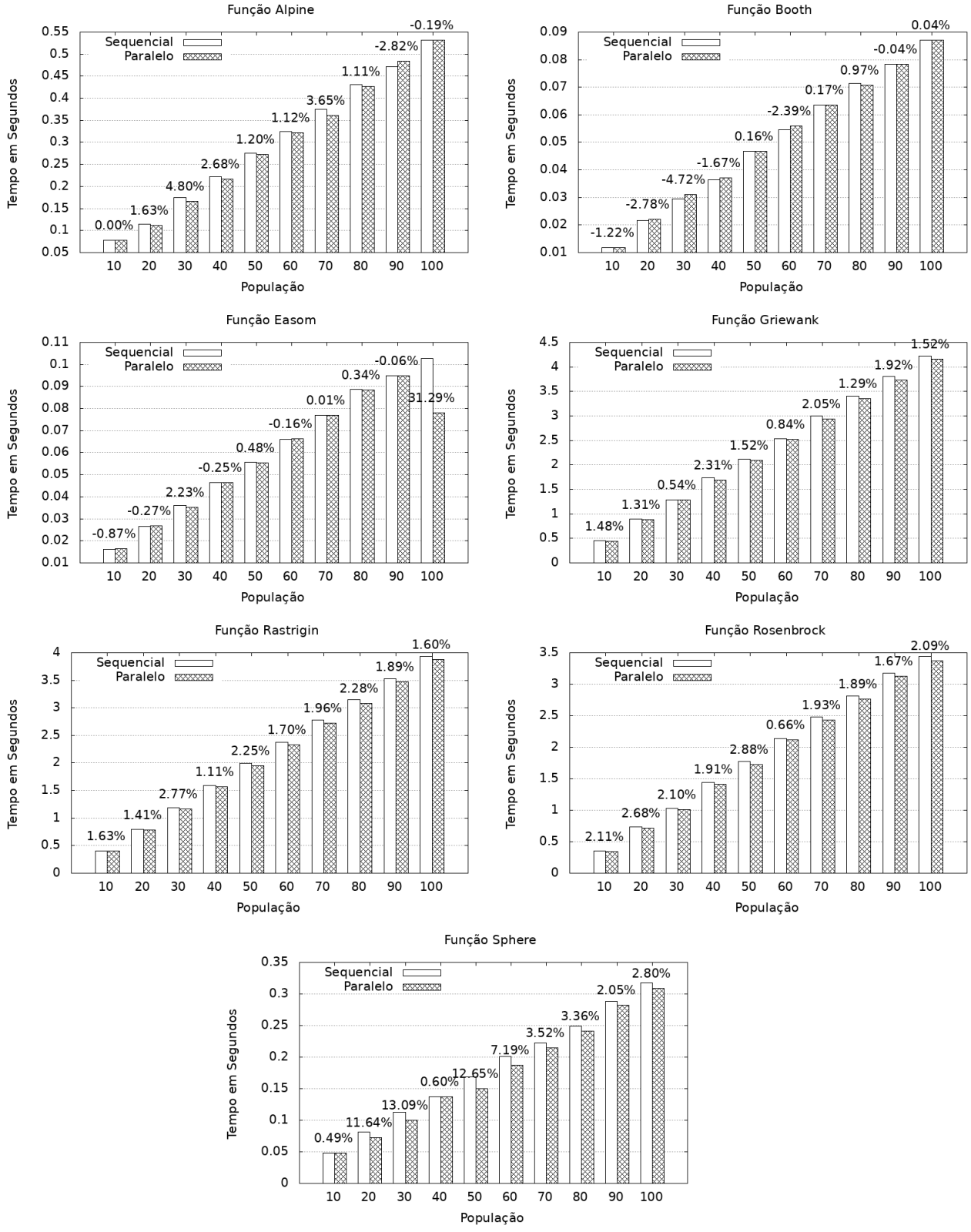
Para representar de forma gráfica os resultados obtidos nos testes realizados, na Figura 13, Figura 14 e Figura 15 são apresentados os tempos de execução obtidos para cada um dos sete casos de teste, variando o tamanho da população de 10 a 100, de 10 em 10 unidades. Como descrito na seção 3.2, o valor de cada par de colunas representa o ganho de desempenho obtido, sendo que os valores negativos representam que a versão paralela teve um desempenho pior que a versão sequencial e valores positivos expressam o ganho obtido da versão paralela sobre a versão sequencial.

4.4 Validação Numérica das Soluções

Os testes realizados a fim de comparar as versões sequencial e paralela[1] retornaram a mesma solução ótima, pois ambas as versões eram submetidas ao mesmo caso de teste.

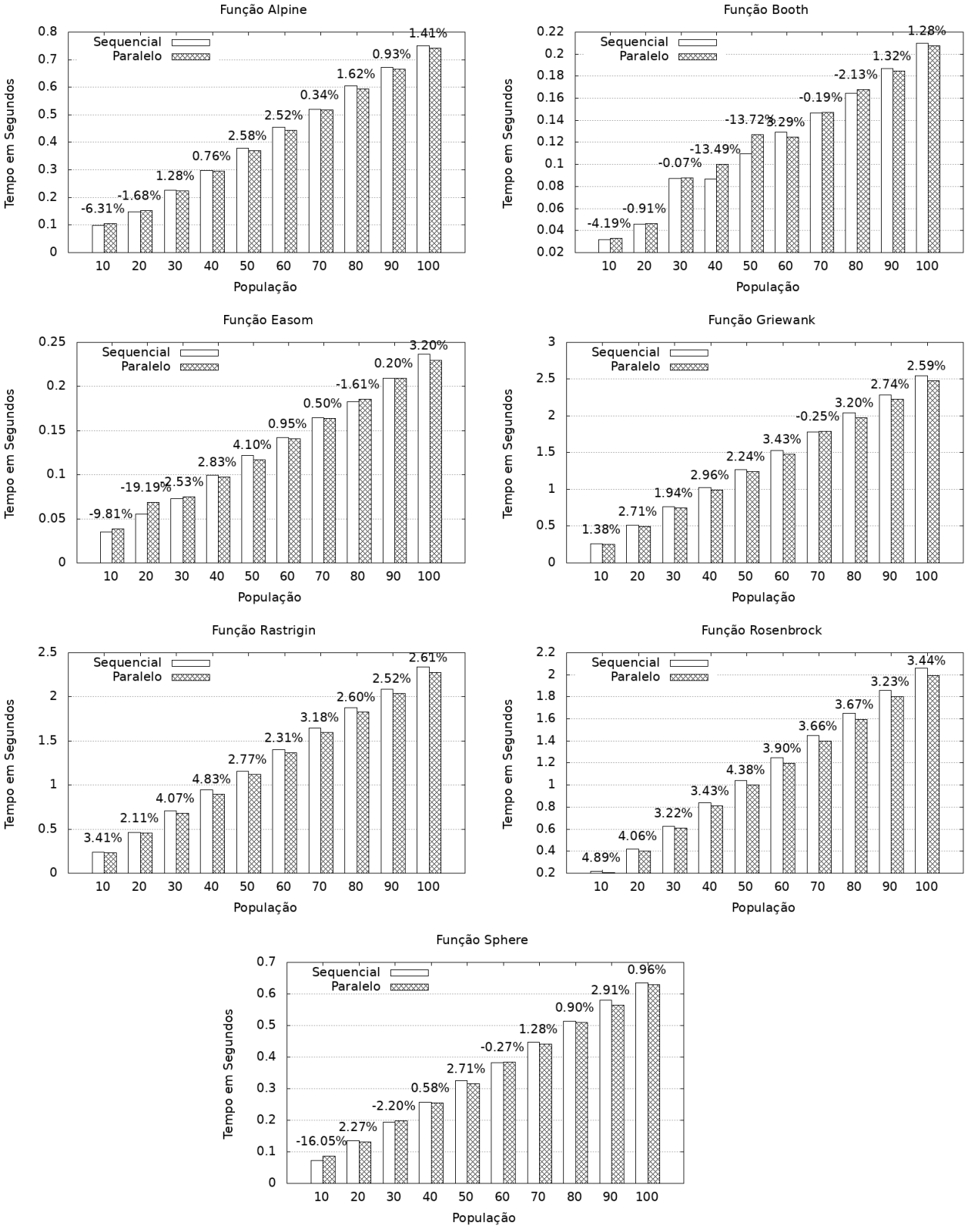
[1] Disponível em: https://drive.google.com/drive/folders/1sFq1T2sKbkmPp1DGrz1ISooR6uqy2Vng?usp=sharing

Figura 13 – PSO - Tempo de Execução das Funções



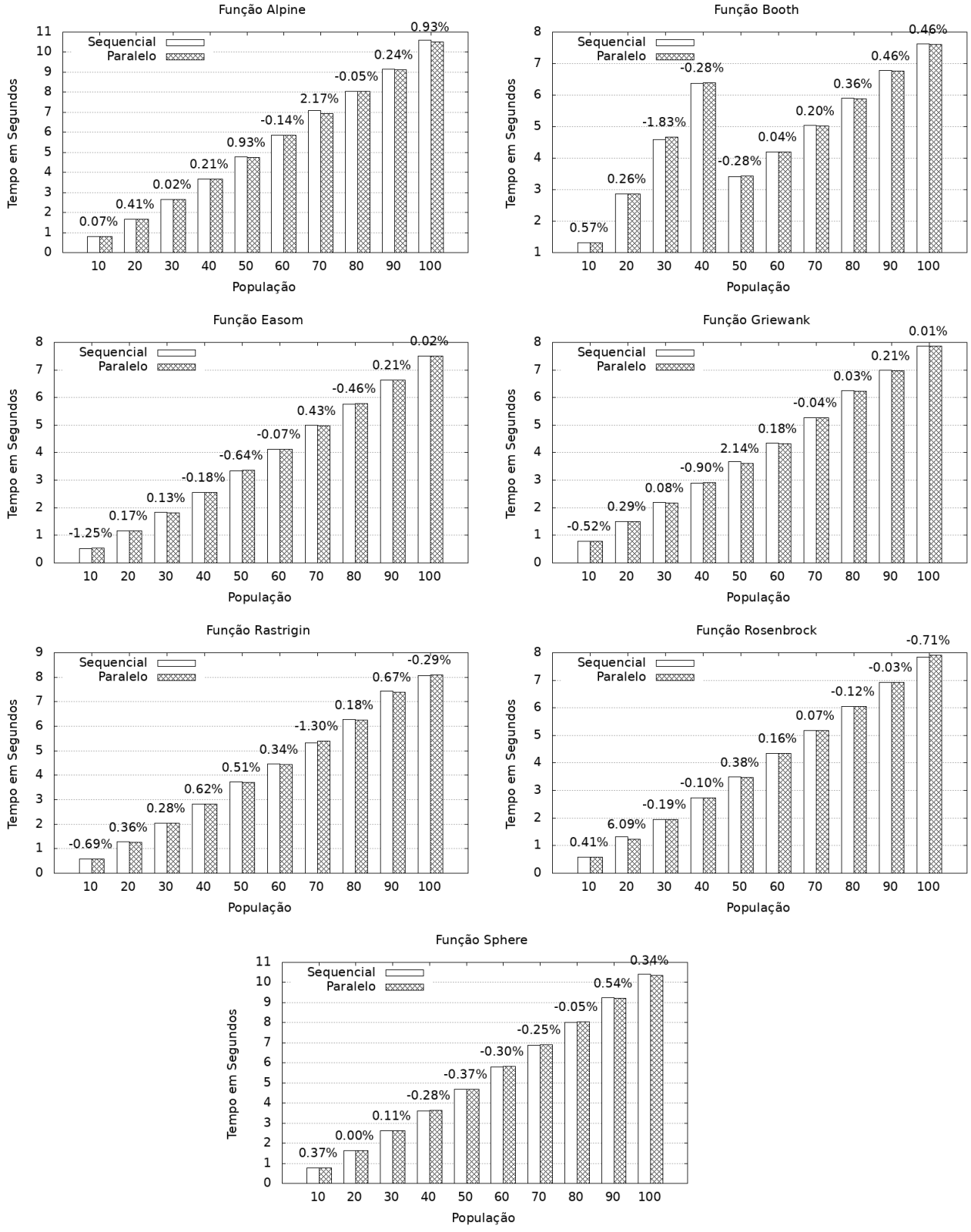
Fonte: Autora

Figura 14 – ABC - Tempo de Execução das Funções



Fonte: Autora

Figura 15 – ACO - Tempo de Execução das Funções



Fonte: Autora

A partir da Tabela 13 e da Figura 14 apresentados na seção anterior, podemos observar que o algoritmo ABC obteve para a função *Alpine*, *Booth*, *Griewank*, *Rastrigin* e *Sphere* soluções próximas à ótima, sendo que o ótimo global dessas funções é $0,00E+00$, a média das soluções obtidas nos testes é de $0,0E-18$, $0,0E-17$, $0,0E-13$, $0,0E-11$ e $0,0E-18$, respectivamente. Enquanto a função *Easom* obteve o ótimo global em 7 dos 10 valores utilizados para a quantidade de agentes da população, sendo que o ótimo global dessa função é $-1,0E+00$, os demais testes da função *Easom* resultaram em soluções próximas à ótima. A melhor solução média encontrada na função *Rosenbrock* foi $10,3E+00$, sendo o ótimo global de $0,0E+00$. Esse foi o pior resultado encontrado pelo algoritmo ABC.

A Tabela 14 e a Figura 15 destacam que o algoritmo ACO obteve a solução ótima para a função *Griewank* com uma população de 30 agentes. As funções *Alpine*, *Booth*, *Easom* e *Sphere*, obtiveram $0,0E-25$, $0,0E-01$, $-0,9E+00$ e $0,0E-55$, respectivamente. Essas funções encontraram soluções próximas da ótima, sendo que o ótimo da função *Easom* é de $-1,0E+00$ e das demais $0,0E+00$. As funções *Rastrigin* e *Rosenbrock* obtiveram a solução $4,39E+01$ e $6,91E+02$. Ambas foram as funções que obtiveram os piores resultados do algoritmo ACO.

A Tabela 12 e a Figura 13 apontam que algoritmo PSO obteve a solução ótima para a função *Booth* no caso de teste com população igual a 20 agentes. As funções *Easom*, *Griewank*, *Rosenbrock* e *Sphere* obtiveram as soluções $-0,9E+00$, $0,9E+00$, $0,6E+00$ e $0,3E-05$, respectivamente, sendo estas soluções próximas às ótimas. As funções *Alpine* e *Rastrigin* obtiveram as soluções $2,5E+00$ e $3,0E+00$, respectivamente, sendo estas as piores soluções encontradas pelo algoritmo.

Comparando os algoritmos, o ABC obteve melhores soluções para as funções *Easom*, *Rastrigin* e *Rosenbrock*. O ACO obteve as melhores soluções para as funções *Alpine*, *Griewank* e *Sphere*. Enquanto o PSO foi melhor para a função *Booth*.

4.5 Avaliação de Desempenho

Os testes apresentados na Tabela 12, na Tabela 13 e na Tabela 14 expõem os tempos de execução para a versão sequencial e paralela com os respectivos desvios padrões entre parênteses. As tabelas também contém o ganho de desempenho, obtido a partir da fórmula descrita na seção 3.6. Esse valor representa o ganho de desempenho da versão paralela sobre a versão sequencial.

O algoritmo PSO, apresenta poucos resultados com perda de desempenho, ou seja, os resultados dos tempos da execução da versão paralela são menores que o tempo de execução sequencial. Para a função *Alpine*, 7 dos 10 testes foram positivos. As funções bidimensionais *Booth* e *Easom* apresentam 4 e 5 testes positivos, respectivamente. Para as demais funções n-dimensionais todos os testes resultaram em ganho de desempenho. Dessa forma é notável que as funções com os piores resultados são bidimensionais com

11 de 20 testes negativos, enquanto as outras cinco funções n-dimensionais apresentam 47 testes positivos contra 3 testes negativos. No total dos 70 testes (7 funções, cada uma com 10 populações), apenas 14 foram negativos.

O algoritmo ABC obteve maior variação sobre ganho de desempenho se comparado com o PSO. As funções *Alpine* e *Sphere* resultaram em 7 testes positivos, a função *Booth*, *Easom* e *Griewank* obtiveram 3, 6 e 9 testes com ganho de desempenho, respectivamente. Somente nas funções *Rastrigin* e *Rosenbrock* todos os testes resultaram em ganhos positivos. Novamente as funções *Booth* e *Easom* obtiveram os piores resultados com 11 de 20 testes negativos. No total dos 70 testes, 52 foram positivos e 18 obtiveram perdas de desempenho.

O ganho de desempenho que o algoritmo ACO obteve para as funções *Alpine*, *Booth*, *Easom*, *Griewank*, *Rastrigin*, *Rosenbrock* e *Sphere* foi de 8, 7, 5, 7, 7, 5 e 5 resultados positivos, respectivamente. Todas as funções obtiveram testes com perdas de desempenho. Esse algoritmo obteve a maior variação entre o ganho de desempenho dos três algoritmos estudados neste trabalho, sendo no total 44 com ganho e 26 com perda de desempenho.

Os maiores ganhos de desempenho obtidos pelo algoritmo ABC são para as funções *Alpine* (4,8%), *Easom* (31,29%) e *Sphere* (13,09%). O algoritmo ACO foi melhor para a função *Rosenbrock* (6,09%). Enquanto as funções *Booth* (3,29%), *Griewank* (3,43%) e *Rastrigin* (4,83%) obtiveram melhores ganhos de desempenho quando submetidas ao algoritmo PSO.

O algoritmo PSO obteve a menor perda de desempenho, considerando os algoritmos testados neste trabalho. Entretanto, obteve a melhor solução apenas para uma função.

É observável que as melhores médias variam o valor da população que a atingiu, tanto de soluções quanto de ganho de desempenho. Cada problema tem um algoritmo e uma população que é melhor aplicado ao problema.

5 CONSIDERAÇÕES FINAIS

Os algoritmos bio-inspirados correspondem a uma gama de estratégias eficientes para a solução de problemas de diversas áreas, desde uma simples análise de dados até problemas complexos como de geração de energia distribuída ou simulações de engenharia. O objetivo principal do trabalho foi alcançado com a implementação da biblioteca bio-inspirada composta pelos algoritmos clássicos ABC, ACO e PSO. As versões sequencial e paralela foram implementadas e submetidas as funções clássicas *Alpine*, *Booth*, *Easom*, *Griewank*, *Rastrigin*, *Rosenbrock* e *Sphere*. A biblioteca implementada permite execuções detalhadas que armazenam as soluções ótimas obtidas no decorrer da execução ou execuções simples que retornam apenas a solução ótima, permitindo que profissionais de diversas áreas utilizem essa biblioteca.

Quanto às soluções obtidas, a maioria dos algoritmos obteve soluções ótimas para as funções de teste. No entanto, as funções *Rastrigin* e *Rosenbrock* obtiveram, em média, soluções menos precisas, por serem funções complexas, uma vez que possuem maior dimensionalidade, mesmo utilizando um número maior de iterações.

Com o auxílio da programação paralela os algoritmos bio-inspirados puderam obter ainda mais desempenho, devido às estratégias de otimização e ao bom uso dos recursos disponíveis. Em nossa implementação, utilizamos instruções do tipo SIMD, usando a API OpenMP.

Quanto ao tempo de execução, todos os algoritmos obtiveram, para todas as funções, um ganho de desempenho médio paralelo positivo. A grande maioria dos testes obteve uma pequena redução de tempo na execução paralela. Já para as funções *Easom* e *Sphere* foi possível obter um ganho de desempenho médio mais expressivo, com 31,29% e 13,09%, respectivamente para o algoritmo PSO.

Observa-se que as funções n-dimensionais obtém ganho de desempenho na grande maioria dos testes realizados, enquanto as funções bidimensionais variam de acordo com o algoritmo. As funções de maior dimensionalidade possuem o maior potencial de paralelismo, uma vez que demandam mais computação que as demais.

A biblioteca bio-inspirada implementada neste trabalho contém os algoritmos propostos testados e documentados. As versões dos algoritmos sequenciais e paralelas são desenvolvidas, descritas, testadas e obtém ganhos de desempenho. A documentação da biblioteca é disponibilizada a fim de facilitar o uso, pois dispõem de exemplos de construções, descrição das variáveis e funções de teste.

Como perspectivas futuras buscamos, com o auxílio da comunidade de desenvolvedores, continuar complementando a biblioteca paralela bio-inspirada com novos algoritmos.

REFERÊNCIAS

ABRAHAM, S.; SANYAL, S.; SANGLIKAR, M. Particle swarm optimization based diophantine equation solver. **arXiv preprint arXiv:1003.2724**, 2010. Citado na página 51.

ALBA, E. et al. MALLBA: A Library of Skeletons for Combinatorial Optimisation. In: MONIEN, B.; FELDMANN, R. (Ed.). **Euro-Par 2002 Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 927–932. Citado na página 44.

ALVES, F. A. et al. Ensinando arquiteturas vetoriais utilizando um simulador de instruções mips. **International Journal of Computer Architecture Education (IJCAE)**, v. 4, n. 1, p. 9–12, 2015. Citado 2 vezes nas páginas 38 e 40.

ARDEH, M. A. **BenchmarkFcns Toolbox: A collection of benchmark functions for optimization**. 2016. [Online; acesso 10-Julho-2019]. Disponível em: <http://benchmarkfcns.xyz/>. Citado 2 vezes nas páginas 48 e 49.

ASTORGA, D. del R. et al. Paving the way towards high-level parallel pattern interfaces for data stream processing. **Future Generation Computer Systems**, v. 87, p. 228 – 241, 2018. ISSN 0167-739X. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167739X17324524>. Citado na página 44.

BANZHAF, W. et al. **Genetic programming: an introduction**. California: Morgan Kaufmann San Francisco, 1998. v. 1. Citado na página 29.

BARNEY, B. **Introduction to Parallel Computing**. 2018. [Online; acesso 10 Maio 2019]. Disponível em: <https://computing.llnl.gov/tutorials/parallel_comp/>. Citado 2 vezes nas páginas 37 e 38.

BISCANI, F.; STROE, D. **PaGMO (Parallel Global Multiobjective Optimizer)**. 2018. [Online; acesso 06 Abril 2019]. Disponível em: <http://esa.github.io/pagmo/>. Citado na página 45.

CARVALHO, G. C. de et al. Otimização com algoritmo bio-inspirado de controle de tráfego em sistemas de grupos de elevadores. **Revista Interdisciplinar de Pesquisa em Engenharia**, v. 2, n. 9, p. 56–76, 2016. Citado na página 23.

CASTRO, L. N. de. Engenharia imunológica: desenvolvimento e aplicação de ferramentas computacionais inspiradas em sistemas imunológicos artificiais. **Universidade Estadual de Campinas, Campinas-SP**, 2001. Citado na página 29.

CASTRO, L. N. de. Fundamentals of natural computing: an overview. **Physics of Life Reviews**, Elsevier, v. 4, n. 1, p. 1–36, 2007. Citado na página 23.

CASTRO, L. N. de et al. Computação natural: Uma breve visão geral. In: **Workshop em nanotecnologia e Computação Inspirada na Biologia**. Santos, São Paulo: Universidade Católica de Santos (UniSantos), 2004. Citado na página 23.

CHAPMAN, B.; MEHROTRA, P.; ZIMA, H. Enhancing openmp with features for locality control. In: CITESEER. **Proc. ECWMF Workshop" Towards Teracomputing-The Use of Parallel Processors in Meteorology**. Austrian: PSU, 1998. Citado na página 40.

CHIACHIA, G.; PENTEADO, B.; MARANA, A. Fusão de métodos de reconhecimento facial através da otimização por enxame de partículas. **UNESP-Faculdade de Ciências, São Paulo**, 2003. Citado na página 35.

COUTO, D. C.; SILVA, C. A.; BARSANTE, L. S. Otimização de funções multimodais via técnica de inteligência computacional baseada em colônia de vaga-lumes. In: **Proceedings of the XXXVI Iberian Latin American Congress on Computational Methods in Engineering**. Rio de Janeiro, RJ: CILAMCE, 2015. Citado na página 48.

DARIANE, A.; MORADI, A. Reservoir operating by ant colony optimization for continuous domains (acor) case study: Dez reservoir. **International Journal of Mathematical, Physical and Engineering Sciences**, v. 3, n. 2, p. 125–129, 2009. Citado na página 34.

DICIOLLA, M. et al. Patient classification and outcome prediction in iga nephropathy. **Computers in biology and medicine**, Elsevier, v. 66, p. 278–286, 2015. Citado na página 23.

DORIGO, M.; BIRATTARI, M. **Ant colony optimization**. Boston, MA, EUA: Springer, 2010. Citado 2 vezes nas páginas 31 e 53.

DORIGO, M. et al. 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. Citado na página 53.

DUNCAN, R. A survey of parallel computer architectures. **Computer**, IEEE, v. 23, n. 2, p. 5–16, 1990. Citado na página 37.

ENGELBRECHT, A. P. **Computational intelligence: an introduction**. South Africa: John Wiley & Sons, 2007. Citado 3 vezes nas páginas 28, 33 e 35.

FLYNN, M. J.; RUDD, K. W. Parallel architectures. **ACM Computing Surveys (CSUR)**, Citeseer, v. 28, n. 1, p. 67–70, 1996. Citado 2 vezes nas páginas 24 e 38.

GUIDORIZZI, H. L. Um curso de cálculo-vol. 1, 5a. edição. **Editora LTC**, 2001. Citado na página 48.

IGNÁCIO, A. A. V.; FERREIRA, V. J. M. F. MPI: uma ferramenta para implementação paralela. **Pesquisa Operacional**, scielo, v. 22, p. 105 – 116, 06 2002. ISSN 0101-7438. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-74382002000100007&nrm=iso>. Citado na página 23.

INTEL. **Intel® SSE4 Programming Reference**. 2007. [Online; acesso 05-Dezembro-2019]. Disponível em: <http://www.info.univ-angers.fr/pub/richer/ens/l3info/ao/intel_sse4.pdf>. Citado na página 40.

INTEL. **Tecnologia de extensões do conjunto de instruções Intel**. 2019. [Online; acesso 02-Dezembro-2019]. Disponível em: <https://www.intel.com.br/content/www/br/pt/support/articles/000005779/processors.html>. Citado na página 40.

IZZO, D.; BISCANI, F. **Welcome to PyGMO**. 2015. [Online; acesso 10 Abril 2019]. Disponível em: <http://esa.github.io/pygmo/index.html>. Citado na página 45.

KARABOGA, D.; BASTURK, B. Artificial bee colony (abc) optimization algorithm for solving constrained optimization problems. In: MELIN, P. et al. (Ed.). **Foundations of Fuzzy Logic and Soft Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 789–798. ISBN 978-3-540-72950-1. Citado na página 30.

KENNEDY, J.; EBERHART, R. C. **Swarm Intelligence**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN 1-55860-595-9. Citado 3 vezes nas páginas 23, 27 e 29.

KRONFELD, M.; PLANATSCHER, H.; ZELL, A. The eva2 optimization framework. In: BLUM, C.; BATTITI, R. (Ed.). **Learning and Intelligent Optimization**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 247–250. Citado na página 44.

LIN, J.; LUCAS, T. A. A particle swarm optimization model of emergency airplane evacuations with emotion. **NHM**, v. 10, n. 3, p. 631–646, 2015. Citado na página 23.

LUKE, S. The ecj owner's manual. **San Francisco, California, A user manual for the ECJ Evolutionary Computation Library**, Citeseer, p. 1–206, 2010. Citado na página 44.

MOLGA, M.; SMUTNICKI, C. Test functions for optimization needs. **Test functions for optimization needs**, v. 101, 2005. Citado na página 48.

MORAES, A. d. **Desenvolvimento e implementação de versões paralelas do algoritmo do enxame de partículas em clusters utilizando MPI**. Tese (Doutorado) — Dissertação de Mestrado, Universidade Federal do Rio de Janeiro, 2011. Citado na página 54.

NIEVERGELT, J. et al. **All the needles in a haystack: Can exhaustive search overcome combinatorial chaos?** Berlin, Heidelberg: Springer Berlin Heidelberg, 1995. 254–274 p. ISBN 978-3-540-49435-5. Disponível em: <https://doi.org/10.1007/BFb0015248>. Citado na página 23.

NORVIG, P.; RUSSELL, S. **Inteligência Artificial: Tradução da 3a Edição**. Rio de Janeiro, RJ: Elsevier Brasil, 2014. v. 1. ISBN 9788535251418. Disponível em: <https://books.google.com.br/books?id=BsNeAwAAQBAJ>. Citado na página 27.

OPENMP. **OpenMP:**. 2008. [Online; acesso 04 Junho de 2019]. Disponível em: <http://www.mathcs.emory.edu/~cheung/Courses/561/Syllabus/91-pthreads/openMP.html>. Citado na página 41.

OPENMP. **The OpenMP API specification for parallel programming**. 2018. [Online; acesso 15 Maio 2019]. Disponível em: <https://www.openmp.org/>. Citado 3 vezes nas páginas 40, 41 e 42.

PAREJO, J. A. et al. Fom: A framework for metaheuristic optimization. In: SLOOT, P. M. A. et al. (Ed.). **Computational Science — ICCS 2003**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 886–895. Citado na página 44.

PAREJO, J. A. et al. Metaheuristic optimization frameworks: a survey and benchmarking. **Soft Computing**, Springer, v. 16, n. 3, p. 527–561, 2012. Citado na página 43.

RAUBER, T. W. Redes neurais artificiais. **Universidade Federal do Espírito Santo**, 2005. Citado na página 29.

RIEDER, R.; BRANCHER, J. D. Development of a micro world for the education of the fundamental mathematics, using opengl and delphi. In: **X Congreso Iberoamericano de Educación Superior em Computación en el marco de CLEI**. Erechim, RS: Universidade Regional Integrada do Alto Uruguai e das Missões, 2002. Citado na página 25.

SERAPIAO, A. Fundamentos de Otimização por Inteligência de enxames: Uma Visão Geral. **Controle y Automacao**, v. 20, p. 271–304, 07 2009. Citado 10 vezes nas páginas 23, 24, 29, 31, 34, 35, 36, 48, 51 e 52.

SILVA, M. A. L. et al. Hybrid metaheuristics and multi-agent systems for solving optimization problems: A review of frameworks and a comparative analysis. **Applied Soft Computing**, Elsevier, 2018. Citado 4 vezes nas páginas 43, 44, 45 e 46.

SOCHA, K.; DORIGO, M. Ant colony optimization for continuous domains. **European journal of operational research**, Elsevier, v. 185, n. 3, p. 1155–1173, 2008. Citado na página 34.

TALUKDER, S. **Mathematicle modelling and applications of particle swarm optimization**. 2011. Citado na página 51.

TAVARES, Y.; NEDJAH, N.; MOURELLE, L. de M. Utilização de otimização por enxame de partículas e algoritmos genéticos em rastreamento de padrões. In: **12 Congresso Brasileiro de Inteligência Computacional**. Rio de Janeiro, Brasil: Diretoria de Sistemas de Armas da Marinha, 2015. Citado na página 35.

TORELLI, J. C.; BRUNO, O. M. Programação paralela em smps com openmp e posix threads: um estudo comparativo. In: **Anais do IV Congresso Brasileiro de Computaçao (CBComp)**. São Carlos, SP: Instituto de Ciências Matemáticas e de Computação Universidade de São Paulo, 2004. v. 1, p. 486–491. Citado na página 40.

UFRJ. **Teoria dos Grafos Aula 8**. 2011. [Online; acesso 10 Maio de 2019]. Disponível em: <http://www.land.ufrj.br/~classes/grafos/slides/aula_8.pdf>. Citado na página 37.

WHITE, D. R. Software review: the ecj toolkit. **Genetic Programming and Evolvable Machines**, Springer, v. 13, n. 1, p. 65–67, 2012. Citado na página 44.

YANG, X.-S. Test problems in optimization. **arXiv preprint arXiv:1008.0549**, 2010. Citado na página 47.

Anexos

ANEXO A – ARQUIVO README.MD DO PSO

Compilar

```
g++ -std=c++11 -o pso main/main.cpp main/benchmarking.cpp
main/benchmarkingP.cpp include/benchmarking.hpp src/particle.cpp
src/pso.cpp include/particle.hpp include/pso.hpp -fopenmp
```

Observação

Para a execução paralela é obrigatório o uso de `-fopenmp`, para as demais execuções é opcional.

Opções de Funções de Teste

- alpine
- booth
- easom
- griewank
- rastringin
- rosenbrock
- sphere

Variáveis: Construtor PSO()

- population: (int) valor inteiro que corresponde ao número de agentes da população.
- boundMin: (double) valor decimal que corresponde ao limite inferior do espaço de busca.
- boundMax: (double) valor decimal que corresponde ao limite superior do espaço de busca.
- c1: (double) valor decimal que corresponde ao coeficiente social do bando de pássaros.
- c2: (double) valor decimal que corresponde ao coeficiente cognitivo do bando de pássaros.
- steps: (int) valor inteiro que corresponde ao npumero máximo de iterações do algoritmo.
- inertia: (double) valor decimal que corresponde a inércia.

- dim: (int) valor inteiro que corresponde a dimensão do enxame.
- typeF: (string) campo de texto [min|max] que corresponde a função de minimização (min) ou maximização (max).

Objeto PSO

`PSO pso (population, boundMin, boundMax, c1, c2, steps, inertia, dim, typeF);`

Exemplo:

`PSO pso(20, -100, 100, 2, 2, 2000, 0.7, 50, min);`

Execução Detalhada

`runDetails(function)` – método que inicializa a execução detalhada da função especificada.

Exemplo:

runDetails(function);

runDetails(griewank);

Execução Sequencial

`run(function)` – método que inicializa a execução seuqencial da função especificada.

Exemplo:

run(function);

run(sphere);

Execução Paralela

`runParallel(function)` – método que inicializa a execução paralelaa da função especificada.

Exemplo:

runParallel(function);

runParallel(rastringin);

ANEXO B – ARQUIVO README.MD DO ABC

Compilar

```
g++ -std=c++11 -o abc main/main.cpp main/benchmarking.cpp
main/benchmarking.hpp src/bee.cpp src/abc.cpp
include/bee.hpp include/abc.hpp -fopenmp
```

Observação

Para a execução paralela é obrigatório o uso de -fopenmp", para as demais execuções é opcional.

Opções de Funções de Teste

- alpine
- booth
- easom
- griewank
- rastringin
- rosenbrock
- sphere

Variáveis: Construtor ABC()

- dim: (int) valor inteiro que corresponde a dimensão do enxame.
- colony: (int) valor inteiro que corresponde ao número de agentes da população.
- lim: (int) valor inteiro que corresponde ao limite de tentativas de melhor uma solução.
- sn: (int) valor inteiro que corresponde ao número de fontes de alimento.
- stepsMax: (int) valor inteiro que corresponde ao número máximo de iterações do algoritmo.
- dMin: (double) valor decimal que corresponde ao limite inferior do espaço de busca.
- dMax: (double) valor decimal que corresponde ao limite superior do espaço de busca.
- typeF: (string) campo de texto [min|max] que corresponde a função de minimização (min) ou maximização (max).

Objeto ABC

ABC abc(dim, colony, lim, sn, stepsMax, dMin, dMax, typeF);
Exemplo:
ABC abc(50, 20, 100, 10, 3000, -100, 100, min);

Execução Detalhada

`runDetails(function)` – método que inicializa a execução detalhada da função especificada.
Exemplo:
abc.runDetails(function);
abc.runDetails(griewank);

Execução Sequencial

`run(function)` – método que inicializa a execução sequencial da função especificada.
Exemplo:
abc.run(function);
abc.run(sphere);

Execução Paralela

`runParallel(function)` – método que inicializa a execução paralela da função especificada.
Exemplo:
abc.runParallel(function);
abc.runParallel(rastringin);

ANEXO C – ARQUIVO README.MD DO ACO

Compilar

```
g++ -std=c++11 -o aco main/main.cpp main/benchmarking.cpp
include/benchmarking.hpp src/ant.cpp src/aco.cpp
include/ant.hpp include/aco.hpp -fopenmp
```

Observação

Para a execução paralela é obrigatório o uso de -fopenmp", para as demais execuções é opcional.

Opções de Funções de Teste

- alpine
- booth
- easom
- griewank
- rastringin
- rosenbrock
- sphere

Variáveis: Construtor ACO()

- k: (int) valor inteiro que corresponde ao tamanho do arquivo de soluções (k >= colony).
- q: (double) coeficiente da equação ω.
- evap: (double) valor decimal que corresponde a taxa de evaporação do feromônio.
- dim: (int) valor inteiro que corresponde a dimensão do enxame.
- colony: (int) valor inteiro que corresponde ao número de agentes da população.
- stepsMax: (int) valor inteiro que corresponde ao número máximo de iterações do algoritmo.
- dMin: (double) valor decimal que corresponde ao limite inferior do espaço de busca.
- dMax: (double) valor decimal que corresponde ao limite superior do espaço de busca.
- typeF: (string) campo de texto [min|max] que corresponde a função de minimização (min) ou maximização (max).

Objeto ACO

`ACO aco(k, q, evap, dim, colony, stepsMax, min, max, typeF);`
Exemplo:
ABC abc(dim, colony, lim, sn, stepsMax, dMin, dMax, typeF);
ACO aco(50, 0.001, 0.85, 50, 20, 100, -100, 100, min);

Execução Detalhada

`runDetails(function)` – método que inicializa a execução detalhada da função especificada.
Exemplo:
aco.runDetails(function);
aco.runDetails(griewank);

Execução Sequencial

`run(function)` – método que inicializa a execução sequencial da função especificada.
Exemplo:
aco.run(function);
aco.run(sphere);

Execução Paralela

`runParallel(function)` – método que inicializa a execução paralela da função especificada.
Exemplo:
aco.runParallel(function);
aco.runParallel(rastringin);

ÍNDICE