

Analysis of the Performance of Genetic Algorithm Parallelized with OpenMP Through Execution Traces

Análise do Desempenho de um Algoritmo Genético Paralelizado com OpenMP Através de Rastros de Execução*

Gabriella Lopes Andrade^{1*}, Márcia Cristina Cera^{1**}

Abstract: Run tracing allows you to identify issues affecting the performance of parallel applications. This work consists in evaluating the parallelization of a Genetic Algorithm applied to the Vehicle Routing Problem with OpenMP, where the performance obtained was not ideally expected. Being that it was possible to obtain a performance increase of 1.4 times in the architecture used, however, but still below ideal. Therefore, the general objective of this work is to investigate the causes of the low performance obtained by the Genetic Algorithm, performing an analysis from the execution traces. Our results showed that the parallelization of the Genetic Algorithm is according to the model in which it was implemented and to the set of instances of the target Vehicle Routing Problem used.

Keywords: Genetic Algorithms — OpenMP — Parallelization — Performance — Score-P — Tracking — Vampir — Vehicle Routing Problem

Resumo: O rastreamento de execução permite a identificação de problemas afetando o desempenho de aplicações paralelas. Este trabalho consiste em avaliar a paralelização de um Algoritmo Genético aplicado ao Problema de Roteamento de Veículos com OpenMP, onde o desempenho obtido não foi o idealmente esperado. Sendo que foi possível obter um aumento do desempenho de até 1,4 vezes na arquitetura utilizada, porém ainda abaixo do ideal. Logo, o objetivo geral deste trabalho é investigar as causas do baixo desempenho obtido pelo Algoritmo Genético, realizando uma análise a partir dos rastros de execução. Nossos resultados mostraram que a paralelização do Algoritmo Genético está de acordo para o modelo em que foi implementado e para o conjunto de instâncias do Problema de Roteamento de Veículos utilizadas.

Palavras-Chave: Algoritmos Genéticos — OpenMP — Paralelização — Desempenho — Score-P — Rastreamento — Vampir — Problema de Roteamento de Veículos

¹ Universidade Federal do Pampa (UNIPAMPA), Brazil - RS - Alegrete

*Corresponding author: gabie.lop.s@gmail.com

**In memoriam.

DOI: <http://dx.doi.org/10.22456/2175-2745.85091> • Received: 23/07/2018 • Accepted: 21/03/2019

CC BY-NC-ND 4.0 - This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

1. Introdução

Uma das principais dificuldades encontradas no desenvolvimento de aplicações paralelas é explorar todo paralelismo existente nas arquiteturas com múltiplas unidades de processamento. Após realizar a paralelização, faz-se necessário realizar uma profunda análise da aplicação, com o objetivo de identificar as regiões do programa que realizam uma baixa exploração dos recursos computacionais disponíveis [1]. A partir dessa análise é possível realizar melhorias na aplicação e assim obter alto desempenho.

Este trabalho propõe o uso de rastros de execução [1] para realizar a análise de uma paralelização de um Algoritmo Genético (AG) aplicado ao Problema de Roteamento de

Veículos (PRV) implementado na linguagem de programação C por [2]. O PRV é um problema de otimização combinatória muito utilizado na distribuição e logística, o qual consiste em rotear veículos com uma capacidade limitada para atender as requisições de um grupo de cidades. A solução do PRV compreende um conjunto de rotas capazes de satisfazer as demandas de todas as cidades e cujo custo seja mínimo [3]. Entre os métodos propostos para solucionar o PRV, o AG foi o escolhido por [2] por conseguir bons resultados quando aplicado a problemas que não possuem uma técnica especializada para resolvê-los.

Apesar do AG retornar a melhor solução encontrada, ele não garante que essa seja a melhor solução possível. Sendo assim, [2] realizaram alguns ajustes, como aumentar o tamanho

da população e o número de evoluções, os quais melhoraram a qualidade das soluções encontradas. Porém, esse ajuste dos parâmetros elevou o tempo de computação do AG. Buscando reduzir o tempo de computação do AG e aumentar seu desempenho, [2] realizaram a paralelização do mesmo utilizando a interface de programação de aplicações (*Application Program Interface - API Open Multi Processing* (OpenMP) [4]. Os resultados obtidos por [2] mostraram que o desempenho da paralelização, embora tenham reduzido o tempo de execução, não foram os idealmente esperados. Os resultados de [2] foram confirmados na análise do AG através da avaliação de seu *Speedup* em trabalhos anteriores [5]. Neste trabalho realiza-se um complemento à análise do desempenho, analisando a determinação das regiões paralelas e a política utilizada para distribuir as iterações por entre as *threads*. Além disso, realiza-se uma análise mais profunda do AG, a partir de seus rastros de execução, os quais permitem identificar a existência de possíveis gargalos afetando o desempenho da aplicação [1]. Em trabalhos anteriores [6] [7] iniciou-se a análise do AG a partir dos rastros de execução, a qual será concluída neste trabalho.

O restante deste trabalho está organizado da seguinte forma: inicialmente é apresentada uma revisão dos principais conceitos que foram aplicados por [2] no desenvolvimento e paralelização do AG (Seção 2); os mecanismos existentes para realizar a análise de aplicações paralelas e os que foram utilizados para analisar o AG paralelo (Seção 3); ambiente de execução, juntamente com os resultados obtidos (Seção 4) e a conclusão deste trabalho (Seção 5).

2. Contextualização

2.1 Problema de Roteamento de Veículos

O PRV é um problema de otimização combinatória pertencente à classe dos problemas NP-Difíceis, os quais são difíceis de solucionar utilizando métodos exatos por se tratarem de problemas com um alto custo computacional [3].

No PRV tem-se um depósito de abastecimento, existe uma demanda de um determinado produto para cada cidade a ser visitada e o veículo possui uma capacidade limitada. O veículo deve sair do depósito, visitar cada uma das cidades e retornar novamente ao depósito. Sendo que cada cidade deve ser visitada apenas uma vez e por apenas um veículo, e que a demanda total atribuída a cada rota não exceda a capacidade do veículo. A solução para o PRV é encontrar um conjunto de rotas capaz de satisfazer a demanda de todas as cidades, e cujo o custo total seja mínimo [3].

Segundo [3] o PRV é geralmente representado por um grafo não direcionado $G = (V, E)$, onde $V = \{v_0, v_1, \dots, v_n\}$ é o conjunto de vértices, sendo que o vértice v_0 representa o depósito e os demais n vértices representam as cidades ($V - v_0$); $E = \{(v_i, v_j) \in V, i < j\}$ é o conjunto de arestas que ligam uma cidade v_i a outra v_j . Cada cidade i é visitada por exatamente um veículo e m veículos visitam o depósito,

conforme a equação 1:

$$\sum_{k=1}^m y_i^k = \begin{cases} m, & \text{se } i = 0, \\ 1, & \text{se } i = 1, \dots, n \end{cases}; \quad (1)$$

onde y_i^k especifica se a rota k contém o vértice v_i e m é o número total de veículos. A capacidade Q_k do veículo não deve ser ultrapassada, conforme a inequação 2:

$$\sum_{i=1}^n q_i y_i^k \leq Q_k, \quad k = 1, \dots, m; \quad (2)$$

onde n é o número total de cidades e q_i representa a demanda da cidade i ($i \in \{1, \dots, n\}$). O veículo que entrou em uma cidade é o mesmo que partiu dela, conforme a equação 3:

$$\sum_{i=0}^n x_{ij}^k = \sum_{i=0}^n x_{ji}^k = y_i^k, \quad j = 0, \dots, n, \quad k = 1, \dots, m; \quad (3)$$

onde x_{ij}^k indica se (v_i, v_j) é ou não uma aresta da rota k na solução ótima. O custo total da viagem C pode ser obtido a partir soma do custo de cada uma das rotas, conforme a equação 4:

$$C(S) = \sum_{i \leq k \leq m} \sum_{(i,j) \in R_k} c_{ij}; \quad (4)$$

onde $R_k = (v_0, v_{k_1}, v_{k_2}, \dots, v_0)$ é o conjunto ordenado, o qual representa os vértices consecutivos na rota do veículo k , $S = \{R_1, R_2, \dots, R_m\}$ é o conjunto solução com m rotas, sendo uma para cada veículo. Onde c_{ij} é o custo da viagem do vértice v_i ao vértice v_j , sendo que $c_{ij} = c_{ji}$. Como o objetivo é encontrar um conjunto de rotas que satisfaça a demanda de todas as cidades com um custo total mínimo [3], o PRV é considerado como um problema de minimização, conforme a equação 5:

$$\min \sum_{i=0}^n \sum_{j=0}^n c_{ij} \sum_{k=1}^m x_{ij}^k. \quad (5)$$

2.2 Algoritmo Genético

O AG é um algoritmo inspirado na teoria da seleção natural de Charles Darwin [8]. No AG cada possível solução do problema é codificada em uma estrutura de dados chamada indivíduo, que compõem a população inicial. Essa população é aplicada ao processo de evolução, que consiste na avaliação dos indivíduos e seleção de quais indivíduos serão aplicados aos operadores genéticos de cruzamento (cria novos indivíduos cruzando características dos genitores) e mutação (altera uma porcentagem de indivíduos). A cada ciclo de evolução uma nova população é gerada, a qual substitui a antiga, que é descartada. Após satisfazer o critério de parada (por número de evoluções ou qualidade das soluções) a população final irá conter os indivíduos mais aptos [8].

A Figura 1 apresenta o fluxograma do AG aplicado ao PRV desenvolvido por [2]. O AG **Inicializa a População**

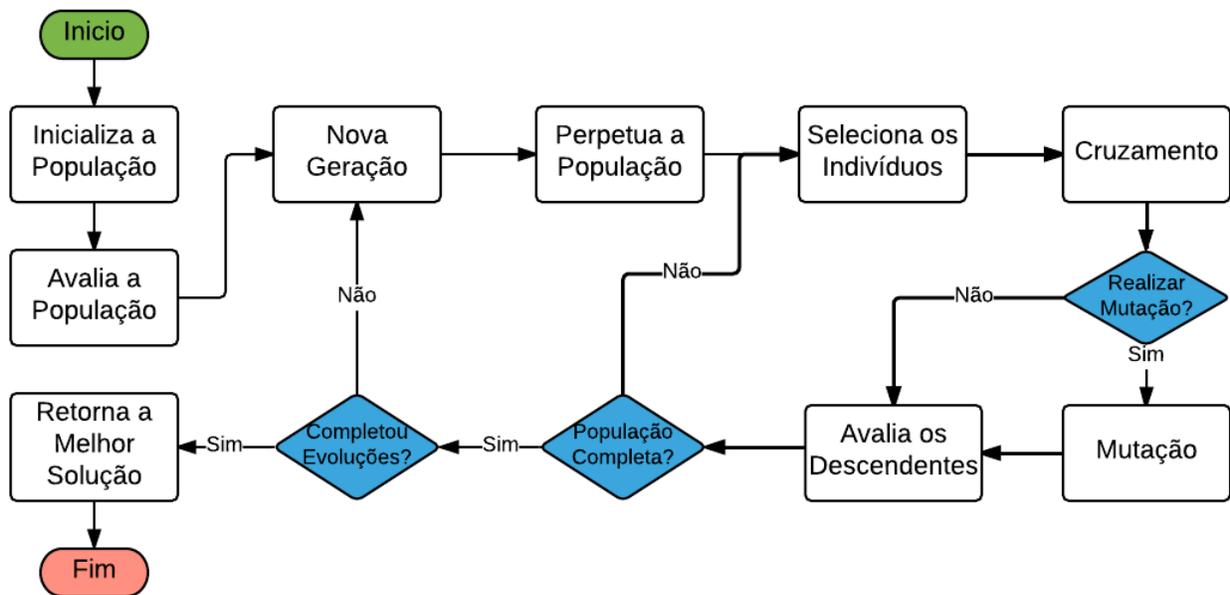


Figura 1. Fluxograma do Algoritmo Genético Implementado por [2].

com várias possíveis soluções para o PRV, onde cada possível solução é gerada sorteando a primeira cidade a ser visitada, e a partir dessa cidade, busca sempre a cidade mais próxima até que todas as cidades tenham sido visitadas. Logo em seguida, o algoritmo **Avalia a População** gerada calculando o custo total (km) de cada conjunto de rotas, sendo que as menores distâncias indicam as melhores soluções. Logo após inicia-se o ciclo de evolução.

A cada ciclo de evolução é recebida uma população como entrada e uma **Nova Geração** de indivíduos é construída. Para construir uma **Nova Geração**, inicialmente ocorre a perpetuação da população (**Perpetua a População**), onde copia-se uma certa porcentagem de indivíduos para a nova população. Dessa porcentagem de indivíduos, metade é composta dos mais aptos e metade de indivíduos aleatórios. O restante da **Nova Geração** de indivíduos é obtida através dos operadores genéticos de **Cruzamento** e **Mutaçao**.

Inicialmente, ocorre a seleção dos indivíduos (**Seleciona os Indivíduos**) que serão aplicados ao **Cruzamento**, o qual combina a carga genética de dois indivíduos selecionados retornando dois descendentes. Metade dos cruzamentos é realizada entre o indivíduo mais apto e um indivíduo escolhido aleatoriamente, e a outra metade entre dois indivíduos distintos escolhidos aleatoriamente. Visando um algoritmo elitista [2] testou duas taxas de cruzamento diferentes, de 100% e de 80%. As técnicas de **Cruzamento** utilizadas por [2] foram: cruzamento uniforme - realiza um sorteio gene a gene de qual dos dois genitores o filho herdará as características; cruzamento de 1 ponto - os cromossomos genitores são seccionados em um ponto aleatório e recombinados intercalando as partes dos genitores; cruzamento de 2 pontos - similar ao cruzamento de 1 ponto, escolhendo dois pontos de secção; híbrida 1 - seleciona aleatoriamente qual das técnicas anteri-

ores utilizar; e híbrida 2 - se após um número de iterações a aptidão dos indivíduos gerados não melhorar troca a técnica de cruzamento.

Após o **Cruzamento** ocorre a etapa de **Mutaçao**, a qual ocorre com uma certa probabilidade em parte de seus genes dos indivíduos. A **Mutaçao** consiste em modificar aleatoriamente os genes de um ou dos dois descendentes gerados pelo cruzamento, com o objetivo de obter uma maior diversidade genética [2]. Foram utilizadas por [2] cinco técnicas de mutação: por troca - troca a posição de genes ou blocos de genes; por inversão - inverte a posição de um bloco de genes; por inserção - remove um bloco de genes e o insere em outra posição do cromossomo; e randômica - seleciona aleatoriamente qual das outras técnicas será utilizada.

A **Nova Geração** de indivíduos é então avaliada (**Avalia os Descendentes**) e após completar o número de evoluções necessárias o AG **Retorna a Melhor Solução** encontrada, ou seja, um conjunto de rotas cuja a soma das distâncias seja a mínima possível.

2.2.1 Algoritmo Genético Paralelo

Segundo [2] ajustes nos parâmetros do AG, como aumentar o tamanho da população ou o número de evoluções, fizeram-se necessários para melhorar a qualidade das soluções. Porém, esse ajuste elevou o tempo de computação do AG, logo, a alternativa escolhida por [2] foi a de paralelizá-lo. A paralelização foi realizada com a API OpenMP, a qual fornece um modelo escalável e portátil para o desenvolvimento de programas com múltiplas *threads* em memória compartilhada [4]. Segundo [2], a OpenMP foi a escolhida por fornecer ganhos de desempenho similares, mas exigir um menor esforço de programação quando comparada a outras APIs de programação paralela.

Para definir o que podia ser paralelizado no AG, [2] geraram o perfil da execução (*profiling*) da aplicação utilizando a ferramenta `gprof`, que é uma ferramenta livre e incorporada ao *GNU Compiler Collection* (GCC). A partir da análise do *profiling* verificou-se que as funções que mais impactam no tempo de execução são a **Cruzamento** e a **Avaliação Descendentes**, que são invocadas durante a construção de uma **Nova Geração** de indivíduos. Porém, o que leva ao impacto no tempo de execução é o grande número de chamadas a essas funções, sendo que uma chamada representa uma porcentagem muito pequena do tempo total de execução. Logo, optou-se por paralelizar as chamadas a essas funções.

Foram identificadas por [2] duas regiões de maior processamento na função **Nova Geração**. A primeira região possui dois blocos de código responsáveis por copiar uma porcentagem de indivíduos para a nova geração, representados na Figura 1 por “**Perpetua População**”. Para paralelizar essa região foi utilizada a diretiva `#pragma omp sections`. A segunda região possui um laço que controla a geração de uma nova população, representado na Figura 1 por “**População Completa?**”. Para paralelizar essa região foi utilizada a diretiva `#pragma omp for`, dessa forma as iterações do laço serão distribuídas entre as *threads*.

3. Mecanismos para Análise de Desempenho

A análise de desempenho é uma etapa muito importante no desenvolvimento de aplicações paralelas por permitir identificar as regiões do programa que realizam uma baixa exploração dos recursos computacionais disponíveis. Nessa etapa o desenvolvedor realiza várias execuções experimentais na aplicação para coletar informações a respeito de sua execução e a partir da análise dessas informações determina o quão eficiente é a aplicação e quais ajustes são necessários para melhorar seu desempenho [1].

Existem várias técnicas para coleta e análise de informações a respeito do comportamento de uma aplicação paralela. Sendo que, a técnica de coleta de dados a ser utilizada depende exclusivamente do tipo de análise que se deseja realizar [1]. Segundo [1] as principais técnicas de coleta de dados utilizadas são:

- **Amostragem:** Consiste em coletar o comportamento da aplicação em intervalos regulares de tempo, os quais são definidos pelo analista;
- **Contagem:** Consiste em medir a quantidade de vezes que um evento ocorre utilizando locais de armazenamento, chamados contadores, que são incrementados a cada vez que um evento específico ocorre;
- **Cronometragem:** É uma técnica que consiste em medir o tempo de execução de uma região do código da aplicação inserindo instruções adicionais, as quais são responsáveis por cronometrar o tempo de execução;

- **Rastreamento:** É uma técnica de coleta interna, onde instrumenta-se a aplicação com uma ferramenta para coletar os rastros de execução.

Nesse trabalho foi utilizada a técnica de rastreamento de execuções pois através dela é possível medir tempos de comunicação, registrar eventos de emissão e recepção, exibir gargalos, entre outras informações importantes sobre o comportamento de aplicação paralelas [9]. Para analisar esses rastros foi utilizada a técnica de análise interativa por visualização de rastros, a qual consiste em transformar os rastros coletados em representações visuais e assim permitir a análise [1].

3.1 Análise de Desempenho Através de Rastros de Execução

Para realizar o rastreamento da execução de uma aplicação paralela é necessário instrumentar a aplicação utilizando uma ferramenta para a coleta de dados [1]. Após a execução da aplicação instrumentada gera-se o rastro, o qual é visualizado através de uma ferramenta de análise interativa. As ferramentas de visualização consistem em transformar o rastro em representações visuais como matriz de comunicações, gráficos de espaço-tempo, etc. Essas representações evidenciam padrões que permitem a identificação de problemas existentes na aplicação [1].

Existem muitas ferramentas tanto para a coleta quanto para a análise dos rastros. Nesse trabalho foi utilizada a Score-P na versão 2.0.2 [10], cuja principal vantagem é de o formato do rastro gerado ser compatível com várias ferramentas de visualização como *Tuning and Analysis Utilities* (TAU) [11], Vampir [12], entre outras. Para utilizá-la acrescenta-se a opção `scorep` à linha de compilação e habilita-se o rastreamento a partir do `prompt` através do comando `export SCOREP_ENABLE_TRACING=true`. Após a execução da aplicação é criado um diretório contendo os eventos rastreados e um arquivo com os rastros de execução. Como o AG foi paralelizado com a API OpenMP, serão coletados eventos referentes aos construtores paralelos.

Inicialmente foi realizada uma tentativa de visualização do rastro com a ferramenta ViTE [13], a qual visualiza um formato de rastro diferente do gerado pela Score-P. Foi realizada a conversão do formato do rastro para um formato compatível. Porém, a visualização com a ViTE não era compreensível, dificultando a análise. Outra tentativa de visualização foi realizada com o pacote `ggplot2` [14], que é um sistema de plotagem de gráficos estatísticos para a linguagem R [15]. A alternativa escolhida foi a de utilizar a ferramenta Vampir na versão de demonstração 9, que embora não seja gratuita apresenta uma interface de fácil uso.

4. Análise dos Resultados

Essa seção apresenta a análise dos resultados obtidos. Inicialmente foi realizada uma análise da determinação das regiões paralelas (Subseção 4.2) e do desempenho do AG (Subseção

4.3) executado em uma arquitetura diferente da utilizada por [2], para verificar se o comportamento da aplicação se mantém apesar da arquitetura utilizada. Logo após foi realizada a análise das regiões paralelas a partir dos rastros de execução (Subseções 4.4 e 4.5) para verificar os possíveis problemas existentes na aplicação.

4.1 Ambiente de Execução

Neste trabalho foram utilizadas instâncias do PRV pertencentes ao *Benchmark* de [16], com restrição quanto a capacidade de transporte do veículo. Foram utilizadas as instâncias c50, c100, c120 e c150, que consistem em percursos com respectivamente 50, 100, 120 e 150 cidades.

O ambiente de execução utilizado é a *Scherm Workstation* do Laboratório de Estudos Avançados em Computação (LEA) da Universidade Federal do Pampa (UNIPAMPA) - Campus Alegrete, a qual possui um processador Intel® Xeon® E5-2603 v3, com frequência de 1.9 GHz e 6 núcleos físicos. O sistema operacional utilizado é o Ubuntu 16.04 LTS com compilador GCC em sua versão 5.31.

Considerando a arquitetura utilizada, o AG foi executado com 2, 3, 4, 5 e 6 *threads*, além da versão sequencial. Para o OpenMP a política de distribuição de iterações foi definida como *Static*, pois essa levou ao melhor desempenho devido ao AG possuir carga regular, onde todas as iterações do laço demandam aproximadamente o mesmo tempo de processamento. Dessa forma, é mais eficiente distribuir as iterações do laço uma a uma de forma igualitária entre todas as *threads* e em tempo de compilação, não acarretando em custo adicional para distribuição da carga. Logo, o uso de políticas de dinâmicas, que distribuem a carga tempo de execução, como *Dynamic* e *Guided* não fornecem ganhos [17].

Baseando-se nos resultados obtidos por [2] foram utilizados os parâmetros do AG que levaram a uma melhora na qualidade das soluções, sendo N o número total de cidades:

- **Tamanho da População:** $N \times 10$;
- **Número de Evoluções:** $N \times N \times 10$;
- **Técnica de Cruzamento:** 1 ponto, onde escolhe-se um ponto em dois cromossomos e intercala-se as seções de gene resultantes gerando dois novos indivíduos;
- **Probabilidade de Mutação:** ocorre mutação em 20% das vezes;
- **Taxa de Mutação:** variável entre 4 a 10% dos genes dos cromossomos;
- **Técnica de Mutação:** Randômica, a qual sorteia quais das técnicas de mutação utilizar entre as técnicas de troca (troca um gene de lugar), troca bloco (troca um bloco de genes), inversão (seleciona e inverte uma seção de genes) e inserção (remove uma seção de genes e a insere em outro lugar do mesmo cromossomo).

Tabela 1. Impacto das Funções no Tempo Total de Execução para a Instância c100 no Processador Intel® Xeon®, Obtido pelo *gprof*.

Função	Número de Chamadas	Soma das Chamadas (s)	% do Tempo Total
Cruzamento	40.000.000	163,07 s	40,37%
Avalia	80.000.000	124,57 s	30,84%
Busca Distância	4.349.585.012	66,01 s	16,34%
Compara Solução	-	20,20 s	5,00%
Nova Geração	100.000	19,92 s	4,93%
Valor Randômico	654.845.231	5,47 s	1,35%
Mutação	160.002.774	3,49 s	0,86%
Calcula Distâncias	1	0,63 s	0,16%
Calcula Tempo	1	0,39 s	0,10%
Vizinho Próximo	1000	0,38 s	0,09%
População Inicial	1	0,04 s	0,01%
Distância	5050	0,00 s	0,00%

4.2 Análise da Determinação das Regiões Paralelas

Para verificar quais regiões do AG poderiam ser paralelizadas, [2] realizou a execução do AG sequencial com a instância c100 do PRV utilizando a ferramenta *gprof*. Essa execução foi novamente realizada para verificar se o comportamento da aplicação se mantém na arquitetura utilizada. A Tabela 1 apresenta a execução realizada no processador Intel® Xeon®. Através dessa execução verificou-se que foram obtidos resultados similares aos de [2], onde as funções que mais impactam no tempo total da execução são: **Cruzamento** representando 40,35% do tempo total da execução com 40.000.000 chamadas, **Avalia** representando 30,82% do tempo total de execução com 80.000.000 chamadas e **Busca Distância** representando 16,33% do total com 4.349.585.012 chamadas. Logo, o comportamento da aplicação se mantém apesar da arquitetura utilizada. Onde as funções que mais impactam no tempo total permanecem a **Cruzamento** e **Avalia**, sendo que o grande número de chamadas a essas funções ocasiona esse grande impacto. Embora essas funções apresentem o maior impacto no tempo de computação, elas não podem ser paralelizadas por apresentarem dependência de dados. Logo, é mais eficiente que elas recebam chamadas paralelas.

Foi identificado que a função **Nova Geração** recebeu 100.000 chamadas, as quais representaram apenas 4,93% do tempo total de execução. Essa função é chamada apenas uma vez a cada geração, porém, ela realiza chamadas diretas às funções **Avalia** e **Cruzamento** e indiretas à **Busca Distância**, as quais mais impactam no tempo de execução. Logo, justifica-se a escolha de [2] em paralelizar as instruções da função **Nova Geração** e não sua chamada.

4.3 Análise de Desempenho

Para validar o AG implementado, [2] utilizaram as instâncias c50, c100 e c120 do PRV, as quais representam percursos com 50, 100 e 120 cidades. Os testes foram realizados em um microcomputador com um processador Intel® Core™ 2 Quad Q8300, com 2.50GHz de frequência e 4 núcleos físicos.

Tabela 2. Configuração das Versões Paralelas do AG que Levaram ao Melhor *Speedup*, a partir dos Resultados de [2], [18] e [5].

Fonte	Instância	Threads	Política	Tempo (s)	Speedup	Eficiência
[2]	c50	4 threads	static	4,36	1,73	0,43
	c100	4 threads	static	228,71	2,26	0,57
	c120	4 threads	static	626,09	2,48	0,62
[18]	c50	8 threads	static	2,60	2,04	0,26
	c100	16 threads	static	121,00	2,64	0,17
	c120	16 threads	static	350,00	2,76	0,17
	c150	32 threads	static	1242,00	3,10	0,10
[5]	c50	6 threads	static	2,87	2,37	0,39
	c100	6 threads	static	127,19	3,26	0,54
	c120	6 threads	static	352,17	3,41	0,57
	c150	6 threads	static	1325,53	3,66	0,61

O AG executado com 2, 3 e 4 *threads*, além da versão sequencial. Variou-se as políticas de distribuição de iterações do OpenMP entre *static*, *dynamic* e *guided*. Três tamanhos de blocos de iterações (*chunk*) foram utilizados por [2], resultando em uma granularidade fina (1 iteração), média (10% do total de iterações) ou grossa (o total de iterações dividido pelo número de *threads*). Foram realizadas trinta execuções para cada conjunto de configurações.

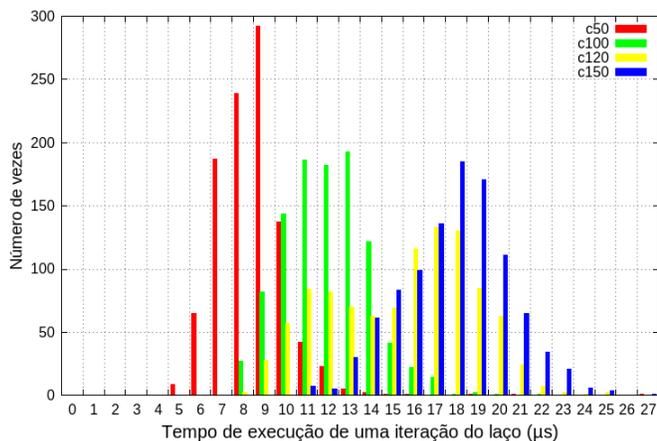
A Tabela 2 apresenta as configurações do AG que levaram ao melhor desempenho, sendo que para todas as instâncias utilizadas por [2] os menores tempos de execução foram obtidos com 4 *threads*. Os tempos de execução representam a média das execuções realizadas, com um desvio padrão inferior a 3% do tempo de execução [2]. Esses tempos representam *speedups* na faixa de 1,73 à 2,48. Logo, o uso de uma *thread* por núcleo fornece o melhor desempenho. Pode-se observar que valor do *speedup* aumenta conforme executa-se instâncias com maiores cargas computacionais, porém estão sempre abaixo do ideal.

A política de distribuição de iterações que levou ao melhor desempenho foi a *static*, devido ao AG possuir uma carga regular, onde todas as iterações demandam aproximadamente o mesmo tempo de processamento. Dessa forma é mais eficiente distribuir as iterações estaticamente e em

tempo de compilação, não acarretando em custo adicional na distribuição da carga. A Figura 2 apresenta um gráfico com o tempo de execução em μs das iterações por instância do PRV pela quantidade de iterações que demoram o mesmo tempo de execução. A média do tempo de execução das iterações para a instância c50 é de 9 μs e o desvio padrão é de 14,21% em relação à média, para a instância c100 a média é de 12 μs e o desvio padrão é de 13,11% em relação à média, para a c120 a média é de 15 μs e o desvio padrão é de 18,30% em relação à média e para a instância c150 a média é de 18 μs e o desvio padrão é de 10,93% em relação à média. Observando esse gráfico verificou-se que a maioria das iterações de cada instância está próxima à média, sendo que a instância c120 é que a possui maior variação no tempo de execução (de alguns μs), devido a probabilidade de ocorrer ou não a mutação (20%). A eficiência obtida por [2] é baixa (usou um pouco mais de 60% do potencial da arquitetura), a qual está na faixa de 0,43 à 0,62. Demonstrando que a aplicação possui limitações de desempenho e que faz-se necessário uma avaliação mais detalhada.

O trabalho realizado por [2] foi expandido por [18], buscando verificar a escalabilidade do AG, executando em uma arquitetura com um maior número de núcleos. Além das instâncias do PRV utilizadas por [2], [18] também utilizou a instância c150 (150 cidades) para testar o comportamento do AG para instâncias com diferentes cargas computacionais, levando em consideração o tempo de computação que o AG consome para processá-las. A arquitetura utilizada por [18] foi a *Gama Workstation* da Unipampa - Campus Alegrete, que possui 2 processadores Intel® Xeon® E5-2650, com 2.80 GHz de frequência, cada um com 8 núcleos físicos e suporte a tecnologia *Hyper-Threading*. O AG foi executado com 2, 4, 8, 16, 32 e 64 *threads*, além da versão sequencial. A política de distribuição de iterações utilizada foi a *static* do OpenMP, por esta ter se mostrado mais eficiente em trabalhos anteriores do mesmo [17]. Foram realizadas trinta execuções para cada configuração.

A Tabela 2 também apresenta as configurações do AG que levaram ao melhor desempenho para [18], sendo que os tempos de execução representam a média das execuções realizadas para cada configuração, com desvio padrão inferior

**Figura 2.** Tempo de Execução do Laço por Instância do PRV.

a 1%. Para as instâncias c50, c100, c120 e c150, os menores tempos de execução foram obtidos com 8, 16, 16 e 32 *threads*, respectivamente. Foi identificado por [18] que o aumento do número de *threads*, associado ao uso de uma máquina com um número maior de núcleos, conseguiu melhorar o desempenho do AG, visto que obteve um *speedup* na faixa de 1,73 à 3,10. Porém, o *speedup* permanece abaixo do ideal. A eficiência obtida é mais baixa, está na faixa de 0,10 à 0,26, a qual diminui conforme aumenta-se o número de *threads* utilizadas.

Na Tabela 2 também são apresentados os resultados das execuções realizadas no processador Intel® Xeon® E5-2603 v3 (Subseção 4.1). Para essa arquitetura foram realizadas trinta execuções para cada uma das configurações. O tempo de execução (em segundos) representa a média das execuções realizadas, com desvio padrão inferior à 0,5%. Esses resultados [5] mostram que a arquitetura utilizada obteve o maior desempenho quando comparado com os resultados obtidos por [2] e [18]. Atribui-se este ganho às características da arquitetura utilizada, uma vez que os demais parâmetros mantiveram-se os mesmos. Os melhores *speedups* por obtidos para as instâncias c50, c100, c120 e c150, foram utilizando 6 *threads*, ou seja, quando utilizam-se todos os núcleos presentes no processador. Esses *speedups* estão na faixa de 2,37 à 3,66. A eficiência obtida está na faixa de 0,39 à 0,61. O aumento do número de *threads* diminui a eficiência do AG paralelo, confirmando que existe um possível gargalo na implementação do AG paralelo afetando o seu desempenho. Sendo assim, foi observado que o problema do baixo desempenho do AG paralelo persiste apesar da arquitetura utilizada. Logo, faz-se necessário uma análise mais detalhada da aplicação, justificando esse trabalho.

4.4 Análise do uso de Sections

A região paralelizada com *sections* possui dois blocos de código, os quais são responsáveis pela perpetuação dos indivíduos da população (“Perpetua a População”). Esses blocos de código não possuem tarefas iterativas, logo podem ser executados em paralelo. Cada um dos blocos fica dentro de uma *section* e cada *section* é executada por uma *thread* diferente. Neste caso, como são apenas dois blocos de código, as tarefas de cada bloco sempre serão executadas por apenas duas *threads*. Logo, quando utilizam-se mais de 2 *threads*, apenas 2 *threads* estarão executando, enquanto que as demais não estão realizando nenhum trabalho útil, estarão apenas aguardando o término da execução para realizar a sincronização (*Join*).

No OpenMP, ocorre uma sincronização implícita quando a execução das regiões paralelas termina, a partir da qual o programa continua sua execução apenas com a *thread* mestre. O tempo de espera em sincronização representa o tempo que uma *thread* que já finalizou sua computação espera até que todas as demais também terminem. Esse tempo aumenta conforme se aumenta o número de *threads* utilizadas [6]. Onde o pior caso é com 6 *threads*, que embora forneça o melhor

desempenho [5], apresenta um tempo de sincronização até 3 vezes maior quando comparado com a execução com apenas 2 *threads* [5]. Isso é ilustrado na Figura 3, onde as *sections* estão representadas em verde e a sincronização em azul. Nessa figura ao executar com 6 *threads*, 4 delas estarão ociosas. Além disso, as *sections* podem ser executadas por *threads* diferentes e uma *section* pode finalizar a sua execução antes da outra, como é o caso da Figura 3d. Embora o uso de 6 *threads* forneça o melhor desempenho [5], apresenta um tempo de sincronização até 3 vezes maior quando comparado com a execução com apenas 2 *threads* [6].

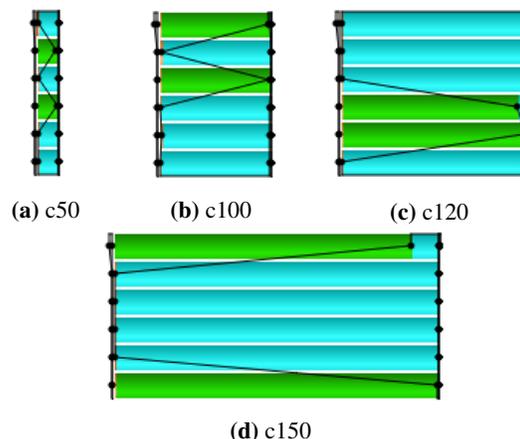


Figura 3. Visualização do Rastro da Execução da Região Paralelizada Utilizando *sections* com 6 *threads*, Gerada pelo Vampir.

Os resultados de trabalhos anteriores [7] mostram que embora o uso de *sections* aumente o tempo espera por sincronização, seu uso aumenta o desempenho do AG paralelo em até 7% quando comparado ao não uso das *sections*. Logo, é preferível esse aumento no tempo de espera em sincronização do que a execução sequencial das regiões paralelizadas com *sections*. Uma alternativa seria a paralelização dessa região utilizando *tasks* ao invés de *sections*. Porém como são apenas dois blocos de código, o problema do tempo de espera por sincronização permanecerá.

4.5 Análise do Parallel For

A Figura 4 ilustra o comportamento do AG paralelo quando é executado um *parallel for* com 6 *threads* para as instâncias c50, c100, c120 e c150. Observando essa figura percebe-se que logo após a execução das *sections* ocorre um *Fork*, criando um conjunto de *threads* para a execução do *for* e após essa execução ocorre a sincronização das *threads* (*Join*). Não há problemas na paralelização com o *parallel for*, sendo que todas as 6 *threads* o executam. Porém, foi identificada a existência de uma região que não foi paralelizada entre o *Join* do *parallel for* e o *Fork* da *section*, a qual é representada pela parte em branco apresentada na Figura 4.

A partir da análise do código foi identificado que entre um *Join* e um *Fork* ocorre a ordenação da população (**Ordena**

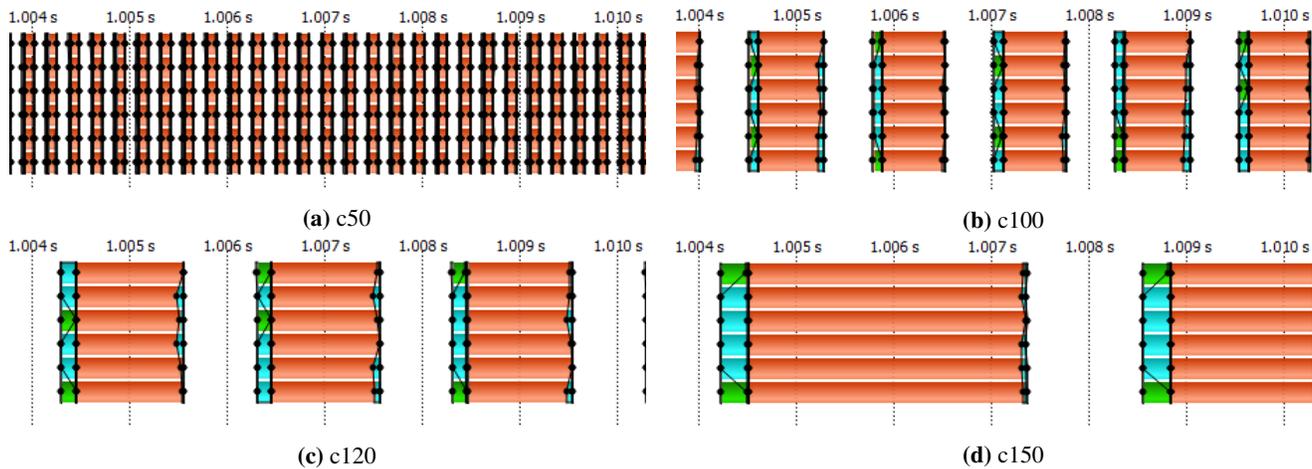


Figura 4. Visualização do Rastro da Execução do AG Paralelo Gerada pelo Vampir.

Tabela 3. Perfil de Execução do AG Sequencial para Cada uma das Instâncias do PRV Gerado a partir do gprof.

Instância	Função	Nº de Chamadas	Soma das Chamadas (s)	% do Tempo Total
c50	Ordena População	12500	-	-
	Compara Solução	-	0,48 s	7,28%
	Inicializa Nova População	12500	0,01 s	0,15%
	Libera Memória	12500	0,03 s	0,61%
c100	Ordena População	100000	-	-
	Compara Solução	-	19,65 s	4,76%
	Inicializa Nova População	100000	0,49 s	0,12%
	Libera Memória	100000	1,58 s	0,38%
c120	Ordena População	172800	-	-
	Compara Solução	-	51,13 s	4,12%
	Inicializa Nova População	172800	1,02 s	0,08%
	Libera Memória	172800	3,55 s	0,29%
c150	Ordena População	337500	-	-
	Compara Solução	-	161,45 s	3,23%
	Inicializa Nova População	337500	3,10 s	0,06%
	Libera Memória	337500	9,89 s	0,20%

População), a inicialização da estrutura para armazenar a nova população (**Inicializa Nova População**) e a liberação da memória da estrutura que armazena a população (**Libera Memória**). O AG sequencial foi novamente executado utilizando o `gprof` para identificar o impacto dessas operações no tempo total de execução. A Tabela 3 apresenta o *profiling* gerado para cada uma das instâncias alvo.

Para ordenar a população [2] escolheu o algoritmo de ordenação QuickSort [19], o qual foi executado utilizando a função `qsort()`, incorporada na biblioteca `stdlib.h` da linguagem C. Essa função recebe como parâmetro a função **Compara Solução**, a qual é responsável por comparar dois elementos da estrutura a ser ordenada com o objetivo de verificar qual é maior e menor. Alguns detalhes do comportamento dessas funções não aparecem no perfil de execução, pelo fato da função `qsort` ser uma função pronta dentro da biblioteca e a função **Compara Solução** ser parâmetro da `qsort`. A função `qsort` é uma função de ordenação rápida, com desempenho $O(n \log(n))$, onde n é o número de elementos a serem ordenados. No caso do PRV, n é o tamanho da população, sendo igual à 500 para a instância c50, 1000 para a c100, 1200

para a c120 e 1500 para a c150. Existe uma paralelização dessa função conhecida como `qsort paralelo`, porém para ordenar poucos elementos a paralelização pode não fornecer ganhos de desempenho, levando em conta o *overhead* que a paralelização pode ocasionar para conjuntos pequenos de elementos.

A função **Inicializa Nova População** e a função **Libera Memória** possuem um impacto muito pequeno no tempo total de execução do AG sequencial, representando 0,76% do tempo total para a instância c50, 0,50% para a c100, 0,37% para a c120 e 0,26% para a c150. Logo, a paralelização dessas duas funções não fornece ganhos de desempenho devido ao custo adicional gerado na criação e gerenciamento das *threads* (*overhead*). A execução da função **Libera Memória** é muito importante para a execução correta da aplicação. Uma vez que a alocação realiza na memória não seja liberada, mesmo que o programa termine sua execução essa memória continuará alocada, ocasionando no chamado vazamento de memória. Em alguns casos o vazamento de memória faz com que o programa consuma toda a memória disponível e interrompa sua execução ocasionando uma falha [20].

Logo, conclui-se que para esse modelo de implementação do AG a paralelização realizada está de acordo. Sendo que a paralelização das funções na região entre o `Join` do `parallel for` e o `Fork` das `sections` não fornece ganhos de desempenho devido ao custo adicional gerado na criação e gerenciamento das `threads`. Uma possível solução para o aumento do desempenho da aplicação é aumentar o grau da paralelização, implementando um modelo mais propenso a ser eficiente após a paralelização, como o modelo de AG baseado em ilhas [21]. Nesse modelo de paralelização, cada ilha consiste em uma `thread` ou processo e executa um AG evoluindo sua própria subpopulação. As ilhas podem trabalhar em conjunto, trocando periodicamente uma porção de suas populações em processo chamado de migração [21].

5. Conclusão

Neste trabalho foi apresentado um AG aplicado ao PRV implementado e paralelizado por [2], cujo o desempenho da paralelização está abaixo do ideal. Logo, o objetivo deste trabalho é investigar os problemas de desempenho existentes e suas causas, a partir da análise dos rastros de execução.

Inicialmente foi verificado se o comportamento dessa aplicação se mantém apesar da arquitetura utilizada. Através dos resultados obtidos com a execução do AG utilizando a ferramenta `gprof`, verificou-se que as funções que mais impactam no tempo de execução do AG se mantêm as mesmas, justificando as escolhas de paralelização de [2]. Através da análise de desempenho verificou-se que houve um aumento do desempenho de até 1,4 vezes na arquitetura utilizada, porém ainda abaixo do ideal.

Foi realizado o rastreamento das execuções do AG e a partir da análise dos rastros verificou-se que o tempo de espera em sincronização é maior quando se usa `sections` do quando não se usa. Porém, seu uso aumenta o desempenho da aplicação em até 7%. Entre o `Join` do `parallel for` e o `Fork` das `sections` verificou-se a existência de uma região que não foi paralelizada, a qual contém 3 funções, onde cada uma representa no máximo 7% do tempo total da execução da aplicação. Logo, a paralelização dessas funções não fornece ganhos de desempenho devido ao *overhead* causado na paralelização.

Conclui-se que para o modelo implementado do AG e para o conjunto de instâncias do PRV usadas, a paralelização da aplicação está de acordo. Logo, não foram identificadas possíveis melhorias. Entretanto, uma possível solução para o aumento do desempenho da aplicação é uma implementação de um modelo de AG baseado em ilhas, o qual é mais propenso a ser eficiente quando paralelizado.

Contribuição dos Autores

Este estudo fez parte do trabalho de conclusão de curso da graduação em Ciência da Computação da primeira autora, Gabriella Lopes Andrade. A autora realizou execuções, análise dos dados, escrita do texto e submissão do artigo. A segunda

autora, Márcia Cristina Cera, foi a orientadora. Ambas trabalharam com a revisão do texto e com a análise dos resultados. Márcia faleceu antes da publicação do trabalho, e sua contribuição foi de suma importância para a conclusão deste.

Referências

- [1] SCHNORR, L. M. Análise de desempenho de programas paralelos. In: *Anais da ERAD/RS 2014*. Alegrete, RS: SBC, 2014. p. 57–81.
- [2] GRESSLER, H. d. O.; CERA, M. C. O impacto da paralelização com `openmp` no desempenho e na qualidade das soluções de um algoritmo genético. *Revista Brasileira de Computação Aplicada*, SBC, Porto Alegre, RS, p. 35–47, 2014.
- [3] REGO, C.; ALIDAE, B. *Metaheuristic Optimization via Memory and Evolution: Tabu Search and Scatter Search*. USA: Springer, 2006. (Operations Research/Computer Science Interfaces Series).
- [4] CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP: portable shared memory parallel programming*. Cambridge, MA, USA: MIT Press, 2008.
- [5] ANDRADE, G. L.; CERA, M. C. Avaliando a paralelização de um algoritmo genético com `openMP`. In: *Anais do WSCAD-WIC 2016*. Aracaju, SE: SBC, 2016. p. 68–73.
- [6] ANDRADE, G. L.; CERA, M. C. Avaliando o uso de `sections openmp` na paralelização de um algoritmo genético. In: *Anais do Salão Internacional de Ensino, Pesquisa e Extensão*. Uruguaiana, RS: SBC, 2016.
- [7] ANDRADE, G. L.; CERA, M. C. Avaliação da eficiência da paralelização com `sections OpenMP` em um algoritmo genético através de rastros de execução. In: *Anais do ERAD/RS 2017*. Ijuí, RS: SBC, 2017. p. 68–73.
- [8] LINDEN, R. *Algoritmos genéticos: uma importante ferramenta da inteligência computacional*. 2a. ed. Rio de Janeiro, RJ: Brasport, 2008.
- [9] PIOLA, T. d. F. *Tracing de aplicações paralelas com informações de alto nível de abstração*. Tese (Doutorado em Física Aplicada) — Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 2007. Disponível em: <http://www.teses.usp.br/teses/disponiveis/76/76132/tde-25102007-114809/>. Acesso em: 18 de maio de 2016.
- [10] VI-HPS. *Score-P User Manual 2.0.2*. [S.l.], 2016. Disponível em: <https://silc.zih.tu-dresden.de/scorep-current/pdf/scorep.pdf>. Acesso em: 27 jul. 2018.
- [11] SHENDE, S. S.; MALONY, A. D. The tau parallel performance system. In: *The International Journal of High Performance Computing Applications*. Thousand Oaks, CA, USA: Sage, 2006. v. 20, n. 2, p. 287–311.
- [12] GWT-TUD GmbH. *Vampir 9.1*. [S.l.], 2016. Disponível em: <https://www.vampir.eu/>. Acesso em: 15 ago. 2016.

- [13] COULOMB, K. et al. *ViTE - Visual Trace Explorer*. [S.l.], 2010. Disponível em: <http://vite.gforge.inria.fr/index.php>. Acesso em: 23 jul. 2018.
- [14] WICKHAM, H. *ggplot2*. 2013. <http://ggplot2.org/>. Acessado em 15/08/2017.
- [15] MATLOFF, N. *The Art of R Programming: A Tour of Statistical Software Design*. San Francisco: No Starch Press, 2011. (No Starch Press Series).
- [16] CHRISTOFIDES, N. et al. *Combinatorial optimization*. John Wiley & Sons, Chichester, UK, 1979.
- [17] DA ROSA, M. F. G.; CERA, M. C. Estudo preliminar sobre a escalabilidade de um algoritmo genético paralelizado com openmp. In: *Anais da ERAD/RS 2015*. Gramado, RS: SBC, 2015. p. 217–220.
- [18] DA ROSA, M. F. G.; CERA, M. C. Análise da escalabilidade de um algoritmo genético paralelizado usando openmp. In: *Anais do WSCAD-WIC 2015*. Florianópolis, SC: SBC, 2015. p. 51–56.
- [19] CORMEN, T. H. et al. *Introduction to Algorithms*. 3. ed. Cambridge, Massachusetts, EUA: MIT Press, 2009.
- [20] GRIFFITHS, D.; GRIFFITHS, D. *Use a Cabeça! C*. Rio de Janeiro, RJ: Alta Books Editora, 2013. (Use a Cabeça!).
- [21] WHITLEY, D.; RANA, S.; HECKENDORN, R. B. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, University Computing Centre Zagreb, v. 7, p. 33–48, 1999.