O Impacto da paralelização com OpenMP no desempenho e na qualidade das soluções de um algoritmo genético

Henrique de Oliveira Gressler [1]
Márcia Cristina Cera [1]

Resumo: O Problema do Roteamento de Veículos (PRV) é um problema combinatório de difícil solução, aplicável tanto para logística de empresas de transporte quanto para melhor ocupação das vias públicas. Resolvê-lo testando todas as combinações possíveis (método de força bruta) torna-se inviável à medida que o problema escala, pois demanda um tempo de computação muito grande. Os Algoritmos Genéticos (AG) são meta-heurísticas capazes de encontrar soluções em um tempo computacional aceitável. Entretanto, mesmo os AG podem demandar um elevado tempo de processamento, dependendo das configurações utilizadas. Com a evolução das arquiteturas computacionais e a difusão das arquiteturas multicore, o uso da programação *multithread* torna-se uma alternativa para reduzir o tempo envolvido na solução de problemas combinatórios. Este artigo objetiva acelerar a resolução do PRV por meio da paralelização do AG com OpenMP, que é um padrão amplamente difundido para programação *multithread*. Nossos resultados atingiram um *speedup* acima de 2, utilizando 4 *threads* em um processador quadcore. Esse ganho está limitado à forma como o AG está implementado. Além do impacto no desempenho do AG também comprovou-se que o uso do OpenMP não afeta a qualidade das soluções. Adicionalmente, o uso do OpenMP permitiu que o AG encontrasse melhores soluções devido ao aumento do número de evoluções computadas num mesmo intervalo de tempo.

***Palavras-chave*:** Algoritmos genéticos. Desempenho. OpenMP. Qualidade das soluções.

Abstract: *Routing Vehicle Problem (RVP) is a combinatorial problem, hard to solve, used to improve the logistics of transport enterprises as well as to improve the traffic in the public ways. To solve it testing all combinations (brute force method) became unfisible as the problem scale, because it demands a large computing time. Genetic Algorithms (GA) are meta-heuristics able to find solutions in an acceptable computing time. However, even GA can demand a large computing time as they are set. Computional architectures evolution and the multicore difusion became the multithread programming an alternativ to reduce GA time. This article aims to speed up the RVP solution through the GA parallelization using OpenMP, which is a popular standard to multithreading programming. Our results show an speedup up to 2 for 4 threads in a quadcore processor. This gain is limited according to our GA is implemented. Beside the performance impact, we also show that the usage of OpenMP did not affect the solutions quality. Furthermore, OpenMP allow the GA to find better solutions because it make possible to increase the number of evolution in an time slice.*

***Keywords*:** *Genetic Algorithms. OpenMP. Performance. Quality of solutions.*

1 Introdução

A resolução de problemas combinatórios tem um papel importante em diversas áreas. Por exemplo, a concepção de circuitos integrados demanda o teste de diferentes combinações de parâmetros, tais como largura e comprimento de transistores [1]. Outro exemplo seriam os testes combinatórios aplicados ao aprimoramento de protocolos de roteamento de redes de computadores, como o *Open Shortest Path First* (OSPF) e o *Distributed Exponentially Weighted Flow Splitting* (DEFT) [2].

[1]Universidade Federal do Pampa - Campus Alegrete - Av. Tiarajú, 810 - CEP: 97546-550 - Alegrete/RS, Brasil
gresslerbwg@gmail.com, marciacera@unipampa.edu.br

http://dx.doi.org/10.5335/rbca.2014.3660

O problema combinatório investigado neste artigo é o Problema do Roteamento de Veículos (PRV) (*Vehicle Routing Problem* - VRP) [3]. A solução desse problema é importante para a sociedade, tendo em vista que possibilita uma logística de entrega e movimentação de produtos com maior eficácia. O PRV caracteriza-se por apresentar um conjunto de cidades - cada uma com sua demanda específica - que precisam ser visitadas por veículos de capacidade limitada. Cada cidade deve ser visitada uma única vez e sempre que se esgotar a capacidade do veículo, o mesmo deve retornar ao depósito para reabastecer-se. Como solução desse problema, espera-se obter um conjunto de rotas que satisfaça à demanda dos clientes e que o somatório dessas rotas tenha um custo total mínimo.

Meta-heurísticas são aplicadas para que seja possível propor soluções suficientemente genéricas para atender a uma vasta gama de problemas combinacionais, como os já citados. Suas vantagens são robustez, simplicidade e possibilidade de execução em um tempo razoável quando comparado ao teste exaustivo (todas as combinações possíveis) [4]. Dentre as meta-heurísticas mais utilizadas, estão os Algoritmos Genéticos (AG) (*Genetic Algorithms* - GA) [5, 6], os quais são um tipo de algoritmo evolutivo baseado em técnicas de otimização global que combinam características de soluções já conhecidas, a fim de encontrar soluções de melhor qualidade.

No geral, os Algoritmos Genéticos retornam a melhor solução encontrada, mas não há garantias de que essa seja a melhor solução possível. Por outro lado, o ajuste de seus parâmetros, com ações tais como aumentar o tamanho da população ou o número de gerações, gera melhoras na qualidade das soluções [5]. Neste artigo, trabalharemos com um AG cujos parâmetros foram configurados de maneira a fornecer soluções de boa qualidade ao PRV. Entretanto, esse ajuste de parâmetros elevou o tempo de computação do AG.

A alternativa escolhida para melhorar o desempenho do AG aplicado ao PRV foi a paralelização. Hoje, para se obter melhor desempenho de computação, não é necessário aguardar que um processador com maior poder computacional seja lançando pela indústria. Novas gerações de processadores, com várias unidades de processamento em um único chip, já dominam o mercado. Essa evolução na arquitetura dos processadores levou ao uso de paradigmas ligados à programação *multithread* e à programação concorrente [7]. Visando tirar proveito do poder de processamento de arquiteturas multicore, este trabalho prevê a utilização da *Application Program Interface* (API) OpenMP [8], que se configura como padrão portável para programação de sistemas de memória compartilhada. OpenMP provê diversas diretivas de compilação, biblioteca de rotinas e variáveis de ambiente, as quais são inseridas diretamente em códigos sequenciais [7]. É usado para estender linguagens de programação como C, C++ e Fortran, com instruções do tipo *single program multiple data* (SPMD) [9].

O objetivo deste artigo é analisar os ganhos obtidos por meio da paralelização do AG com OpenMP em dois aspectos: no desempenho de execução e na qualidade das soluções encontradas. Assim, pretende-se mostrar que além de reduzir o tempo de execução, a execução *multithread* do AG é capaz de encontrar soluções de boa qualidade. Para isso, este artigo apresenta uma análise do impacto da variação de parâmetros de paralelização do OpenMP no desempenho e na qualidade das soluções.

O restante do artigo está organizado da seguinte forma; a seção 2 contextualiza o problema-alvo e os AG. Após, a seção 2.1 detalha as características do AG implementado e da configuração de seus parâmetros. A paralelização com OpenMP do AG está descrita na seção 3 e a seção 4 apresenta os nossos resultados experimentais. Por fim, tem-se a conclusão e as referências deste artigo.

2 Contextualização

O Problema do Roteamento de Veículos (PRV) [3] é uma variante do Problema do Caixeiro Viajante (PCV) [6], que considera que cada cidade terá uma demanda específica. O veículo utilizado tem uma determinada capacidade de atendimento. Assim, sempre que se esgotar a capacidade do veículo, esse deve retornar ao estoque e iniciar uma nova rota de entrega, envolvendo cidades que ainda não foram visitadas. Logo, o conjunto de rotas resultante do atendimento das cidades em função da capacidade do veículo corresponde a uma solução do problema. O objetivo é encontrar o conjunto de rotas cujo somatório da distância percorrida seja o menor, ou seja, é minimizar a distância total percorrida na viagem.

Nosso trabalho usa parte do *Benchmark* de Christofides et al. [10] como fonte de instâncias do PRV, por esse ser largamente utilizado [11, 3, 12, 13, 14]. Para esse *Benchmark*, é conhecido o menor custo em km já obtido para as instâncias, logo, é possível determinar o quão longe as soluções encontradas estão da melhor solução conhecida.

Algorithm 1 Algoritmo Genético sequencial

```
1: Início
2: Gera população inicial
3: Avalia todos os indivíduos
4: while ( !Terminou ) do
5:     Copia indivíduos perpetuados para Nova Geração
6:     for (1/2 da taxa de cruzamento) do
7:         if (Iterador menor que 1/4 da taxa de cruzamento) then
8:             Seleciona 1 indivíduo entre os melhores
9:             Seleciona 1 indivíduo aleatoriamente
10:            Cruza os indivíduos selecionados
11:            if (Com certa probabilidade) then
12:                Aplica mutação em parte dos genes de 1 ou 2 descendentes
13:            end if
14:            Avalia indivíduos gerados
15:        else
16:            Seleciona 2 indivíduos aleatoriamente
17:            Cruza os indivíduos selecionados
18:            if (Com certa probabilidade) then
19:                Aplica mutação em parte dos genes de 1 ou 2 descendentes
20:            end if
21:            Avalia indivíduos gerados
22:        end if
23:    end for
24:    if (Número de evoluções foi atingido) then
25:        Terminou ← Verdade
26:    end if
27: end while
28: Retorna Melhor Solução Encontrada
29: Fim
```

Foram escolhidas as instâncias `c50`, `c100` e `c120`, respectivamente com 50, 100 e 120 cidades. A escolha levou em consideração o tempo de computação para o estabelecimento de rotas entre as cidades. Cada instância tem sua própria capacidade de transporte do veículo. Além disso, cada uma delas especifica a coordenada do depósito, assim como o número, a coordenada e a demanda de cada cidade da instância.

Os Algoritmos Genéticos [5, 6] são inspirados na teoria da evolução das espécies de Charles Darwin, a qual considera-se que as possíveis soluções do problema são indivíduos, os quais estão organizados num grupo chamado população. Na área computacional, um cromossomo é uma estrutura de dados que representa uma das possíveis soluções do espaço de busca do problema. Os indivíduos podem ser selecionados e cruzados, gerando novos indivíduos, preservando algumas características dos genitores. Ao longo das gerações, uma nova população substitui a antecessora. Os constantes cruzamentos, balizados por uma função de avaliação, aumentam a diversidade genética, o que, eventualmente, leva a soluções de melhor qualidade. Depois de um certo número de evoluções, o algoritmo retorna o melhor indivíduo encontrado.

2.1 Algoritmo Genético implementado

Para explicar as características do AG implementado, observaremos o Algoritmo 1. Primeiramente, é gerada uma população inicial (linha 2), a qual é composta por vários indivíduos, sendo que cada um deles uma possível solução para o PRV. Para gerar um indivíduo na população inicial, escolhe-se, aleatoriamente, qual a primeira cidade a ser visitada. A partir desse ponto, busca-se sempre a cidade mais próxima até que todas as cidades, sem repetição, tenham sido visitadas. Esse processo se repete até que o número de indivíduos da população tenha sido alcançado. Com a população inicial completa, os indivíduos são avaliados (linha 3). A avaliação é o processo que define qual é o custo de cada solução, em km, ou seja, qual a distância total percorrida. O custo é a métrica utilizada para determinar se um indivíduo é mais apto que outro. Quanto menor o custo, mais apto é o indivíduo. Então, começa o ciclo das novas gerações (laço entre as linhas 4 e 27).

Tabela 1: Configuração dos parâmetros do AG para as instâncias alvo

Parâmetros	c50	c100	c120
Tamanho da população	$N \times 10$	$N \times 10$	$N \times 10$
Número de evoluções	$N \times N \times 10$	$N \times N \times 10$	$N \times N \times 10$
Taxa de cruzamento	80%	80%	80%
Técnica de cruzamento	Híbrido 2	2 pontos	1 ponto
Probabilidade de mutação	20%	20%	20%
Taxa de mutação	4 a 10%	4 a 10%	4 a 10%
Técnica de mutação	Troca bloco	Troca bloco	Troca bloco

Uma nova geração começa a ser construída pela perpetuação de um percentual dos indivíduos da geração atual (linha 5). Desse percentual, metade é composta por aqueles mais aptos, e, a outra metade, por indivíduos aleatórios. O restante dos indivíduos da nova geração é preenchido com aqueles gerados a partir de cruzamentos entre indivíduos da população atual (laço entre linhas 6 e 23). O cruzamento ou *crossover* combina a carga genética de dois indivíduos chamados de pais e cada cruzamento retorna dois descendentes. Metade dos cruzamentos é realizada entre um dos melhores indivíduos e outro escolhido aleatoriamente dentro da população atual (linha 10). A outra metade dos cruzamentos é realizada entre dois indivíduos distintos selecionados randomicamente (linha 17). Essa escolha aleatória visa à manutenção da diversidade genética, de forma que a população não venha a convergir rapidamente, aumentando, assim, as chances de produzir indivíduos mais aptos.

Com certa probabilidade, cada indivíduo retornado sofre mutação em parte de seus genes (linhas 12 e 19). Essas operações consistem em alterar aleatoriamente os genes de um ou dos dois descendentes gerados no cruzamento, a fim de obter uma maior diversidade genética. Os descendentes, então, são avaliados e adicionados à nova geração (linhas 14 e 21). Quando estiver completa, a nova geração substitui a população anterior. Isso se repete até que o número de evoluções predeterminado seja atingido e, então, o algoritmo retorna o melhor indivíduo encontrado.

2.2 Parâmetros do AG para as instâncias-alvo

Os Algoritmos Genéticos permitem uma série de ajustes em seus parâmetros, os quais impactam diretamente na qualidade das soluções encontradas. A Tabela 1 resume a melhor configuração encontrada para as instâncias-alvo. Esses valores foram obtidos a partir do estudo do estado da arte e de execuções de experimentos com nossa implementação de AG e estão detalhados a seguir.

Uma população muito pequena tende a fornecer uma diversidade genética insuficiente para conduzir o AG a boas soluções, por outro lado, se a população for grande demais, podemos estar aproximando-nos de uma busca exaustiva [5]. Adicionalmente, realizar um número pequeno de evoluções faz com que aconteçam poucos cruzamentos o que reduz a diversidade genética da população [5]. Baseando-se nisso, nossa implementação foi testada com dois tamanhos de população: N e $N \times 10$ e com dois números de evolução: $N \times N$ e $N \times N \times 10$, sendo N o número de cidades da instância. Os melhores resultados foram encontrados com um tamanho da população de $N \times 10$ e o número de evoluções de $N \times N \times 10$, o que resultou em distâncias de aproximadamente 35% a 60% menores, dependendo da instância testada, quando comparada com os resultados das demais combinações desse parâmetro [15].

A taxa de cruzamento determina o número de indivíduos de uma nova população que serão gerados a partir de cruzamentos. Segundo Michalewicz [16] e Linden [5], a taxa de cruzamento deve variar entre 60% e 95%. Neste trabalho, foram testadas duas possibilidades de variação na taxa de cruzamento, visando um algoritmo elitista, ou seja, perpetuando os melhores indivíduos da população atual na nova população. A primeira considera uma taxa de cruzamento de 100% - 1 indivíduo, perpetuando somente o melhor indivíduo da população atual. A segunda possibilidade considera uma taxa de cruzamento de 80% (aproximadamente a média dos valores sugeridos por Michalewicz [16]). Essa versão preserva 20% dos indivíduos da população anterior, sendo que metade desses são aleatórios e a outra metade é composta pelos melhores indivíduos. A identificação desses melhores garante que cada um deles tem características distintas, ou seja, não há indivíduos repetidos. A taxa de cruzamento de 80% mostrou-se mais eficiente, com soluções aproximadamente 60% a 80% menores, dependendo da instância, quando

comparadas as soluções com a primeira opção testada [15].

As técnicas de cruzamento recombinam a carga genética de dois indivíduos, gerando descendentes que tenham características dos dois genitores. Em Linden [5], estão definidas as três técnicas básicas: **cruzamento de 1 ponto** – os cromossomos dos genitores são seccionados em um ponto aleatório e recombinados intercalando as partes dos genitores; **cruzamento de 2 pontos** – similar ao cruzamento de 1 ponto, entretanto, com a escolha aleatória de 2 pontos de secção; e **cruzamento uniforme** – sorteio gene a gene, de qual dos dois genitores o filho herdará a característica, formando, ao final do processo, dois novos cromossomos completos. Embora encontrem-se trabalhos com implementações de outras técnicas de cruzamento, tais como CX [17], PMX (*Partially Matched Crossover*), OC (*Ordered Crossover*) e CC (*Cyclic Crossover*) [18], entre outras, essas representam variações das técnicas básicas. Neste trabalho, como nosso foco está na paralelização, optamos por usar as técnicas básicas e implementar duas técnicas híbridas, que são combinações das técnicas básicas. Essas combinações foram implementadas porque enquanto o cruzamento uniforme sorteia gene a gene qual genitor passará a característica para o descendente, o cruzamento de 1 ponto preserva uma das extremidades do código genético do genitor no descendente e o cruzamento de 2 pontos preserva as duas extremidades do código genético do genitor no descendente. No decorrer das evoluções, essas características podem ser boas ou ruins. Por exemplo, se uma técnica levou o algoritmo a uma condição de estagnação, trocá-la por outra pode conduzir a bons custos, em outro ponto no espaço de busca. Por outro lado, se o algoritmo está evoluindo na direção certa dentro do espaço de busca, trocar a técnica pode afastá-lo dos bons custos. Assim, implementou-se a técnica **Híbrida 1**, que seleciona randomicamente qual das três técnicas será adotada. Desse modo, pode-se chegar a melhores soluções pela diversidade genética decorrente do uso aleatório das técnicas básicas de cruzamento. Já a técnica **Híbrida 2**, ocorre uma análise da aptidão dos indivíduos gerados. Se essa não estiver melhorando, procede-se a troca de técnica de cruzamento. Assim, pode-se chegar a melhores soluções, alternando a técnica de cruzamento sempre que houver estagnação na qualidade da solução encontrada.

De acordo com as características da instância-alvo, as técnicas de cruzamento atingem diferentes níveis de qualidade das soluções encontradas. Nossos testes apontaram que, para a instância `c50`, a técnica Híbrida 2 conseguiu a maior redução na distância percorrida, aproximadamente 60% quando comparada às demais técnicas. Para a instância `c100`, a melhor técnica foi a de 2 pontos, com reduções de aproximadamente 45%, e, para a `c120`, a melhor técnica foi a de 1 ponto, com cerca de 55% de redução na distância percorrida [15].

De acordo com Linden [5], as mutações devem acontecer com uma probabilidade baixa e indica-se que essa probabilidade seja dada por: 100% - taxa de cruzamento. Como nossa taxa de cruzamento foi fixada em 80% (aproximadamente a média dos valores sugeridos por Michalewicz [16]), logo utilizou-se uma probabilidade de mutação de 20%. Segundo Dalba e Dorronsoro [19], a mutação é o fator responsável por incluir na população um grau de diversidade considerável, sendo uma importante ferramenta para evitar a convergência prematura.

A taxa de mutação indica a quantidade de genes do cromossomo que sofrerão alguma alteração. De acordo com Rayward-Smith [20], Michalewicz [16] e Linden [5], por razões históricas, quando estamos trabalhando com cromossomos binários, a taxa de mutação geralmente fica entre 0,5% e 1%. Há consenso de que se a codificação do cromossomo for decimal, a taxa de mutação deve ser maior. Adicionalmente, uma taxa de mutação variável é mais eficiente do que uma taxa de mutação fixa, já que, com o avanço das evoluções, as novas populações tendem à estagnação. Nesse sentido, uma taxa de mutação maior nas últimas evoluções das populações compensaria essa tendência. Com base nisso, foram testadas três variações: mutação em 4%, 10% e variando de 4% a 10% dos cromossomos. Essa última começa com 4% e vai aumentando gradativamente até chegar a 10%, antes que se esgote o número total de evoluções. A taxa variável de 4% a 10% levou aos maiores percentuais de redução de distância percorrida de aproximadamente 7% a 63%, conforme a instância e quando comparada às demais variações [15].

Segundo Linden [5], as três técnicas de mutação mais conhecidas são: por troca - troca a posição de genes ou bloco de genes selecionados aleatoriamente; por inversão - inverte o posicionamento dos genes de um bloco selecionado aleatoriamente; e por inserção - remove um bloco de genes selecionado aleatoriamente e o insere noutra posição do cromossomo. Nossa implementação suporta tais técnicas, separando o procedimento por troca em: troca de genes e troca de bloco de genes. Adicionalmente, implementou-se uma quinta técnica que seleciona randomicamente qual das quatro técnicas será utilizada. Para todas as instâncias testadas, as maiores reduções na distância percorrida – de aproximadamente 30% a 65% quando comparada às demais técnicas – foram encontradas utilizando a técnica de troca de bloco de genes [15].

3 Paralelização do Algoritmo Genético

O primeiro passo para a paralelização do algoritmo proposto foi a realização de uma análise do tempo de processamento da versão sequencial. Essa análise foi apoiada pela ferramenta `GProf` vinculada ao *GNU Compiler Collection* (GCC), a qual gera um perfil da execução (*profiling*), e apontou que o consumo concentra-se nas funções de cruzamento (linhas 10 e 17 no Algoritmo 1) e de avaliação de indivíduos (linhas 14 e 21) invocadas durante as evoluções (geração de novas populações ou gerações). Entretanto, o consumo é resultante do grande número de chamadas das funções cruzamento e avaliação e não do tempo necessário para computar uma chamada a cada uma dessas funções. Nesse sentido, os esforços foram concentrados na paralelização das chamadas a essas funções (linhas 4 a 27).

Uma das diretivas de paralelização mais difundidas do OpenMP realiza a paralelização das iterações de laços `for`: diretiva `#pragma omp for`, a qual faz com que as iterações sejam divididas entre um conjunto de *threads*, sendo que há uma barreira de sincronização implícita ao encerrar o laço. Outra diretiva de paralelização do OpenMP realiza a paralelização de tarefas não iterativas: diretiva `#pragma omp sections`. Dentro desse construtor, diferentes blocos de código são chamados de `section` e são distribuídos a diferentes *threads*.

Na paralelização do AG, a diretiva `#pragma omp sections` foi inserida, envolvendo a linha 5 do Algoritmo 1. A perpetuação dos indivíduos de uma geração para a outra se dá em dois trechos distintos de código. Um busca os melhores indivíduos distintos e o outro indivíduos aleatórios. Como essas tarefas não apresentam dependências, podemos usar a diretiva de paralelização de tarefas não iterativas. Já a diretiva `#pragma omp for` foi inserida no laço mostrado no Algoritmo 1, linhas 6 a 23. Assim, a geração dos indivíduos da nova população acontece simultaneamente. É interessante notar que há dependência entre as iterações do laço `while` (linhas 4 até 27), já que a população gerada numa iteração torna-se a entrada da iteração seguinte. Assim, esse laço não é candidato à paralelização.

O OpenMP prevê para a diretiva `#pragma omp for` a cláusula `schedule(type,[chunk])` que permite alterar a política de escalonamento das iterações entre as *threads*. Adicionalmente, as cláusulas também permitem determinar o tamanho da fatia de iterações (`chunk`) destinada a cada *thread*. Essas cláusulas permitem ao programador ajustar a carga de trabalho das *threads*. As políticas de escalonamento utilizadas neste artigo foram: ***static*** – iterações divididas em fatias de tamanho fixo, associadas às *threads* em tempo de compilação, ideais para problemas em que os tempos de computação das iterações são semelhantes entre si; ***dynamic*** – iterações são divididas em fatias de tamanho fixo, as quais são associadas às *threads* em tempo de execução e tendem a oferecer um bom balanceamento da carga de trabalho quando o tempo de computação das iterações difere entre si; e ***guided*** – as fatias de iterações são associadas às *threads* em tempo de execução, entretanto o tamanho das fatias é decrescente a cada alocação, onde o *chunk* define o tamanho mínimo da fatia e aprimora o escalonamento oferecido pelo *dynamic*, deixando fatias menores para equilibrar a finalização da execução.

4 Resultados experimentais

Esta seção descreve os resultados obtidos a partir do Algoritmo Genético implementado. Inicialmente, apresentamos o ambiente de execução utilizado nos testes na seção 4.1. Após, na seção 4.2, mostramos o tempo de execução sequencial e a qualidade das soluções encontradas de acordo com os parâmetros especificados na seção 2.2. A análise do impacto da paralelização no desempenho de execução do AG é mostrada na seção 4.3, enquanto o impacto na qualidade da solução é apresentado na seção 4.4.

4.1 Ambiente de execução

Os testes foram realizados num Intel® Core 2 Quad Q8300 2.50GHz e Sistema Operacional Debian Squeeze com kernel 2.6.32-5-amd64. O compilador utilizado foi o *The GNU C Compiler*, versão 4.4.5-1. Os dados coletados foram armazenados em uma base de dados MySQL, com intuito de facilitar a análise posteriormente. Os tempos de execução representam a média de trinta execuções, com um desvio padrão inferior a 3% do tempo de execução.

4.2 Resultados da versão sequencial

O objetivo desta seção é analisar o efeito dos parâmetros definidos na seção 2.2 no tempo de execução sequencial e na qualidade das soluções encontradas pelo AG implementado. A Tabela 2 apresenta o tempo de execução sequencial em segundos, a melhor solução computada pelo AG implementado e a melhor solução conhecida da literatura para cada uma das instâncias testadas. As soluções representam o custo para percorrer todas as cidades da instância em km.

Tabela 2: Tempos de execução sequencial (em segundos) para as instâncias alvo, a melhor solução computada e a melhor solução encontrada na literatura, ambas em Km.

Instância	Tempo (s)	Melhor solução computada (km)	Melhor solução conhecida (km)
c50	7,54	543,19	524,61
c100	517,83	891,17	826,14
c120	1553,98	1069,63	1042,11

Em relação ao tempo de execução mostrado na Tabela 2, pode-se perceber que quanto mais cidades a percorrer, maior é o tempo de execução do AG. Isso é uma característica inerente aos problemas combinacionais. Para todas as instâncias, a melhor solução computada pelo AG sequencial obteve um custo maior quando comparada à melhor solução conhecida da literatura. Para a instância `c50` a solução encontrada foi 18,58 km maior do que a solução conhecida, enquanto que esta diferença foi de 65,03 km para a `c100` e de 27,52 km para a instância `c120`. Essas distâncias serão utilizadas como base na análise das seções seguintes.

4.3 Impacto da paralelização no tempo de execução

O objetivo desta seção é mostrar o impacto da paralelização no tempo de execução do AG implementado, ou seja, o ganho de desempenho atingido por intermédio do uso do OpenMP. Como os testes foram executados em um processador com quatro núcleos (*cores*), o algoritmo paralelo computou com 2, 3 e 4 *threads*. Explorando as potencialidades do `#pragma omp for`, variou-se a política de distribuição de iterações testando-se as políticas *static*, *dynamic* e *guided*. Para cada política, foram testados três tamanhos de *chunk*, que resultam em uma granularidade grossa (fatias representando o total de iterações divido pelo número de *threads*); média (fatias de 10% do total de iterações) ou fina (fatias de uma iteração). Essa configuração das execuções visa uma comparação entre as políticas e o tamanho das fatias de iterações.

As Tabelas 3, 4 e 5 apresentam os tempos de execução (em segundos) e os *speedups* para cada uma das instâncias-alvo, variando-se os tamanhos de fatia de iterações e o número de *threads* com as políticas de distribuição de iterações *static*, *dynamic* e *guided*, respectivamente. O melhor desempenho obtido para cada instância está destacado em negrito em cada uma das tabelas. É possível notar que todos os maiores *speedups* foram obtidos utilizando-se quatro *threads*. Entretanto, esses ficaram abaixo do ideal (que seria um *speedup* de 4), sendo o maior de 2,48 para a instância `c120` com a política *static*, e o menor de 1,66 para a instância `c50` com as políticas *static* e *guided*. Isso se deve ao fato de que a nossa implementação paralelizou a geração de indivíduos para uma nova população. Como de uma evolução para a outra, a nova população baseia na atual, fazem-se necessárias sincronizações, o que impacta no desempenho.

Tabela 3: Tempo de execução em segundos (T) e *speedup* (S) com a política *Static*, variando-se o tamanho da fatia de iterações (fina, média e grossa) e o número de *threads* (2, 3 e 4) para as instâncias-alvo

Instâncias		c50			c100			c120		
Num. Threads		2	3	4	2	3	4	2	3	4
Fina	T	6,27	4,98	4,53	328,63	252,62	228,71	921,34	740,22	663,45
	S	1,20	1,51	**1,66**	1,58	2,05	**2,26**	1,69	2,10	2,34
Média	T	6,21	4,87	4,61	328,72	260,29	235,19	919,79	723,71	652,77
	S	1,21	1,55	1,64	1,58	1,99	2,20	1,69	2,15	2,38
Grossa	T	6,36	5,06	4,64	326,07	253,70	234,69	920,16	739,81	626,09
	S	1,19	1,49	1,63	1,59	2,04	2,21	1,69	2,10	**2,48**

Tabela 4: Tempo de execução em segundos (T) e *speedup* (S) com a política *Dynamic*, variando-se o tamanho da fatia de iterações (fina, média e grossa) e o número de *threads* (2, 3 e 4) para as instâncias-alvo

Instâncias		c50			c100			c120		
Num. Threads		2	3	4	2	3	4	2	3	4
Fina	T	6,13	5,06	4,36	328,70	259,90	235,49	928,63	736,09	645,51
	S	1,23	1,49	**1,73**	1,58	1,99	2,20	1,67	2,11	2,41
Média	T	6,56	4,87	4,72	332,31	257,91	235,64	919,06	727,09	645,24
	S	1,15	1,55	1,60	1,56	2,01	2,20	1,69	2,14	**2,42**
Grossa	T	6,06	4,86	5,19	328,85	252,89	332,32	920,18	730,98	654,85
	S	1,24	1,55	1,45	1,57	2,05	**2,23**	1,69	2,13	2,37

Os menores ganhos de *speedup* obtidos foram para a instância `c50`, independentemente da política de distribuição utilizada (*static* e *guided* de 1,19 a 1,66 e *dynamic* de 1,15 a 1,73). Esse resultado já era esperado tendo em vista que essa instância tem o menor tempo de computação (Tabela 2), na qual os ganhos obtidos acabam sendo afetados pela sobrecarga (*overhead*) da paralelização. Adicionalmente, a diferença de tempo obtida com a variação do tamanho das fatias de iterações é baixa, ficando na casa dos décimos de segundo. Logo, para esta instância que executa em um tempo pequeno, pode-se concluir que é mais eficiente manter a configuração padrão do OpenMP, que é a granularidade fina. Com relação à política de distribuição das iterações, *Dynamic* mostrou-se 4% mais eficiente (*speedup* de 1,73 com quatro *threads*).

Os maiores *speedups* obtidos foram para a instância `c120`: 2,48 com a política *static* e granularidade grossa; 2,42 com *dynamic* e granularidade média; e 2,43 com *guided* e granularidade média. Para essa instância, as granularidades média e grossa mostraram-se mais eficientes, além de que o maior *speedup* foi atingido com a política *static*. Como o tempo de computação dessa instância é significativamente maior, uma atribuição de fatias de iterações em tempo de compilação associada a grandes fatias é capaz de resultar em desempenhos melhores. Essa tendência pode ser também identificada para a instância `c100`, que tem o maior *speedup* com a política *static* (2,26). Em relação ao tamanho das fatias de iterações para a instância `c100`, a granularidade fina resultou no melhor *speedup* para a política *static*, enquanto que, para as políticas *dynamic* e *guided* a granularidade grossa foi mais eficiente (*speedups* de 2,23 e 2,22, respectivamente).

Os resultados apresentados nas Tabelas 3, 4 e 5 permitem-nos concluir que o desempenho da paralelização do AG implementado está diretamente relacionado com as características da instância-alvo: número de cidades, capacidade de transporte do veículo e demandas de cada cidade - características essas que determinam o tempo de computação necessário para solucionar o problema. Assim, é difícil apontar previamente qual política de distribuição de iterações ou granularidade será mais eficiente sem realizar testes envolvendo as possíveis configurações, conforme apresentamos nesta seção. A faixa de ganho de desempenho obtida, principalmente para as instâncias maiores, que foi acima de dois, embora abaixo do *speedup* ideal, reduz satisfatoriamente o tempo de espera por soluções. Entretanto, isto somente será útil se a qualidade das soluções encontradas minimamente mantenha-se nos mesmos patamares das soluções da versão sequencial. A seguir, analisaremos mais detalhadamente a qualidade das soluções obtidas.

Tabela 5: Tempo de execução em segundos (T) e *speedup* (S) com a política *Guided*, variando-se o tamanho da fatia de iterações (fina, média e grossa) e o número de *threads* (2, 3 e 4) para as instâncias-alvo

Instâncias		c50			c100			c120		
N. Threads		2	3	4	2	3	4	2	3	4
Fina	T	6,33	5,13	4,92	331,65	258,77	235,98	916,09	729,04	650,20
	S	1,19	1,47	1,53	1,56	2,00	2,19	1,70	2,13	2,39
Média	T	6,12	5,23	4,81	329,93	256,39	235,6	919,53	740,57	638,52
	S	1,23	1,44	1,57	1,57	2,02	2,20	1,69	2,10	**2,43**
Grossa	T	6,29	5,04	4,55	331,69	253,16	233,12	930,04	748,19	661,69
	S	1,20	1,50	**1,66**	1,56	2,05	**2,22**	1,67	2,08	2,35

4.4 Impacto da paralelização na qualidade da solução

O objetivo desta seção é mostrar que além de reduzir o tempo de execução, o uso do OpenMP proporcionou a obtenção de soluções de melhor qualidade. O ponto de partida das investigações mostradas nesta seção era identificar se a execução concorrente da geração de indivíduos da nova população (nossa implementação paralela do AG) não afetaria a diversidade genética dos indivíduos, levando a resultados de pior qualidade. No AG, a diversidade genética advém da escolha dos genitores e das mutações aplicadas com certa probabilidade. Assim, compararemos quando toda a população é gerada sequencialmente (um indivíduo após o outro) com a geração paralela (*threads* gerando indivíduos ao mesmo tempo).

A Tabela 6 resume as melhores soluções encontradas pela versão paralela do AG implementado para as instâncias alvo. Comparando com as soluções encontradas pela versão sequencial (Tabela 2) para a instância `c50`, obteve-se exatamente a mesma solução. Como essa instância possui um tempo de computação pequeno, obter-se soluções na versão paralela de mesma qualidade que a versão sequencial é um resultado satisfatório. Expandindo esse mesmo tipo de análise para as instâncias `c100` e `c120`, pode-se notar que as versões paralelas atingiram soluções melhores que a versão sequencial. Para a instância `c100`, a diferença entre a melhor solução conhecida e a melhor solução computada baixou de 65,03 km para 49,28 km (redução de 15,75 km, ou seja, 24%). Já para a instância `c120` essa diferença baixou de 27,52 Km para 23,52 Km (redução de 3,95 Km, ou seja, 14%). Isso mostra que foi possível obter soluções de melhor qualidade computando paralelamente. Em outras palavras, a paralelização não prejudicou a diversidade genética dos descendentes, possibilitando a geração de indivíduos mais aptos. Isso porque, a cada geração de indivíduos acontece a combinação dos genes e a perpetuação dos mais aptos. A paralelização permitiu que ramos distintos de descendentes fossem evoluídos numa mesma população fazendo com que um escopo maior de alternativas fosse testado.

Buscando mostrar que a solução paralela encontra soluções de boa qualidade e que os resultados mencionados não representam apenas um caso em que o resultado foi melhor, apresentamos a Tabela 7, na qual se tem, para cada instância, o percentual de soluções que se encontram dentro de uma faixa de tolerância de 5% de distância da melhor solução conhecida. Para a instância `c50`, percebe-se que a paralelização reduziu o percentual de soluções dentro da tolerância. Cabe ressaltar que essa instância tem um tempo de computação muito baixo e que a paralelização não encontrou soluções melhores que a obtida sequencialmente. Para a instância `c100`, embora tenhamos obtido uma redução de 24% entre a melhor solução conhecida e a melhor solução computada, tanto a versão sequencial quanto a paralela encontraram soluções fora do limite de 5% de tolerância. Por fim, para a instância `c120`, apenas a paralelização com dois *threads* não encontrou 100% das soluções dentro da tolerância. Com isto, podemos concluir que os resultados não representam apenas um caso isolado. Adicionalmente, a solução do problema alvo visa o encontro da melhor solução, ou seja, da configuração e ordenamento de visitas a cidades e ao estoque que proporcione a menor distância total a ser percorrida.

É importante notar que não há uma relação direta entre a configuração que fornece os melhores *speedups* e a que fornece as soluções de melhor qualidade. As configurações de melhor desempenho atingiram as seguintes soluções: 543,19 km para a instância `c50`; 897,16 km para a `c100`; e 1068,67 km para a `c120`. No caso da instância `c100`, o custo da solução quando se obteve o melhor *speedup* foi pior que o encontrado pela versão sequencial (Tabela 2). Isto se justifica pelo fato de que a qualidade das soluções está diretamente relacionada ao número de evoluções e a diversidade genética dos indivíduos.

Dada essa situação, resolvemos explorar a paralelização do AG para possibilitar a execução de um número maior de evoluções em um mesmo período de tempo. Para isso, executamos as versões paralelas do AG alterando o critério de parada de número de evoluções para tempo de computação, deixando-os computar pelo tempo gasto pela versão sequencial. A Tabela 8 mostra as melhores soluções encontradas para as instâncias alvo e a configuração

Tabela 6: Melhor solução encontrada (em km) pela versão paralela do AG implementado e sua configuração para as instâncias-alvo

Instância	**Melhor solução encontrada (Km)**	**Configuração:** Política - Granularidade - Número Threads
`c50`	543,19	Todas - Todas - Todas
`c100`	875,42	*Guided* - grossa - 4 threads
`c120`	1065,68	*Dynamic* - média - 2 threads

Tabela 7: Percentual de soluções encontradas dentro de uma faixa de tolerância de 5% da melhor solução conhecida para cada instância-alvo

Instâncias	5% de tolerância	Sequencial	2 *threads*	3 *threads*	4 *threads*
c50	524,61 a 550,84	83%	73%	67%	67%
c100	826,14 a 867,45	0%	0%	0%	0%
c120	1042,11 a 1094,21	100%	93%	100%	100%

que as atingiu.

Seguindo a mesma tendência que nos testes anteriores, para a instância c50 não houve melhora na qualidade da solução. Entretanto, é importante notar que a mudança no parâmetro de parada do algoritmo levou a execução de um número de evoluções de 1,19 a 1,72 vezes maior para essa instância do que com o critério original. Isso porque a instância demanda um baixo tempo de computação que consequentemente resultou em um pequeno acréscimo no número de evoluções computadas com a mudança de critério de parada, o qual foi insuficiente para levar a melhorias na qualidade da solução.

Já para as instâncias c100 e c120, essa mudança de critério de parada levou a uma pequena melhora na qualidade das soluções: de 875,42 km passou para 872,25 km com uma quantidade de evoluções de 1,56 a 2,12 vezes maior para a c100 e de 1065,68 km para 1065,27 Km com uma quantidade de evoluções de 1,63 a 2,35 vezes maior para a c120. É importante notar que embora o número de evoluções e o tempo de execução tenham aumentado significativamente com essa alteração de critério de parada, o ganho na qualidade foi pequeno quando comparado com o ganho obtido com a utilização do OpenMP mostrado no início desta seção. Muito provavelmente as soluções estejam situadas próximas a ótimos locais sem que ocorressem fatores capazes de fazê-las melhorar. Sendo assim, a paralelização do AG com OpenMP mostrou-se mais eficiente do que mudar o parâmetro de parada do algoritmo. Futuramente, pretende-se testar outra arquitetura multicore a qual tenha um número maior de *cores*, possibilitando execuções com mais que quatro *threads*. Isso tende a ser mais eficiente do que executar pelo mesmo tempo que a versão sequencial, uma vez que cada *thread* explora ramos diferenciados de diversidade genética de indivíduos das populações.

5 Trabalhos relacionados

Com o objetivo de melhorar o desempenho dos Algoritmos Genéticos, a maioria dos autores recorre à hibridização desses [5, 21, 22]. Essa técnica combina um AG com outra técnica de Inteligência Artificial para obter resultados melhores do que os que seriam alcançados, aplicando-se as técnicas isoladamente [5]. A ideia básica consiste em usar o AG para fazer uma busca global e uma heurística ou outra meta-heurística para realizar uma busca local na vizinhança do cromossomo. Bons resultados também são alcançados gerando uma população inicial com auxílio de uma heurística e não de um método puramente estocástico, como num AG tradicional [21]. Outro mecanismo utilizado para melhorar a qualidade das soluções é o cruzamento otimizado. Enquanto a maioria dos AG usam métodos estocásticos para realizar os cruzamentos, essa técnica usa a função objetivo para auxiliar o processo [21]. O cruzamento otimizado gera dois descendentes, um otimizado (que tenta obter a melhor função objetivo dos descendentes recém criados) e outro exploratório (que visa manter a diversidade genética).

A hibridização exemplificada melhora a qualidade das soluções ao custo de um aumento no tempo de computação dos AG. Nossa proposta diferencia-se dessas pelo fato de que utilizamos a programação paralela para reduzir o tempo de computação do AG, mantendo, e na maioria casos, melhorando a qualidade das soluções

Tabela 8: Melhor solução encontrada (em km) pela versão paralela do AG implementado executando pelo tempo sequencial e sua configuração para as instâncias-alvo

Instância	**Melhor solução encontrada (Km)**	**Configuração:** Política - Granularidade - Número Threads
c50	543,19	Todas - Todas - 2, 3 e 4 Threads
c100	872,25	*Static* - fina - 3 threads
c120	1065,27	*Static* - média - 2, 3 e 4 Threads

encontradas. Assim, é possível tirar proveito de uma tendência que são as arquiteturas multicore implementando o AG sob o paradigma de programação com memória compartilhada. Nós partimos de uma versão tradicional do AG e paralelizamos com OpenMP, o qual exige um esforço menor de programação quando comparado com outras Interfaces de Programação Paralela (IPP), como por exemplo *Pthreads*. É importante salientar que o uso do OpenMP na paralelização do AG também pode ser aplicado a versões que façam uso da hibridização.

A paralelização de AG vem sendo alvo de pesquisas como, por exemplo, implementações baseadas no modelo de ilhas onde as populações são isoladas em ilhas e o melhor indivíduo de cada ilha é migrado para outra [23]. Esse modelo torna a granularidade do problema maior, possibilitando a execução em *clusters* de computadores via *Message-Passing Interface* (MPI) [24]. Entretanto, esse pode ser adaptado para ambiente híbridos conforme proposto por Parcut (2014), em que o MPI é combinado com CUDA para permitir a execução do algoritmo em *Graphic Processing Units* (GPUs) [25].

Considerando paralelizações voltadas a processadores multicore, destacamos a paralelização com OpenMP de uma variação de um Algoritmo Genético para a resolução do Problema de Atribuição de Localidades a Anéis em Redes SONET [26]. Nesse, o algoritmo apresentou *speedups* baixos, de 0,5 a 2, porém o objetivo era encontrar soluções ótimas e a paralelização proporcionou de 91% a 100% de acerto nas instâncias testadas. Isso demonstra que a relação ganho de desempenho *vs.* ganho de qualidade de solução não é direta na paralelização de heurísticas conforme mostramos em nossos resultados.

Silva (2012) [27] comparou uma versão OpenMP com uma versão *Pthreads* da paralelização de um AG. A comparação considerou duas arquiteturas de testes diferenciadas, uma com dois e a outra com quatro *cores*. Na arquitetura com dois *cores*, OpenMP obteve o menor tempo, enquanto os resultados encontrados por *Pthreads* foram os mais próximos do ideal, o que colabora com nossa argumentação de que para AG paralelizados o melhor desempenho não está relacionado as soluções de melhor qualidade. Na arquitetura com quatro *cores*, *Pthreads* obteve um melhor *speedup*, porém a diferença aumenta conforme aumentam-se o número de cidades da instância (diferença entre *speedups* de 0,02 para 52 cidades, chegando até 6,89 com 318 cidades). Novamente, reforça-se que o ganho de desempenho na paralelização está diretamente relacionado à característica da instância, em que as maiores instâncias trazem mais oportunidades de ganho. Considerando que o esforço de programação é menor utilizando OpenMP quando comparado com *Pthreads*, OpenMP mostra-se como uma alternativa para a paralelização de AG para processadores multicore, justificando nossa escolha por esta IPP.

6 Conclusão

Este artigo apresentou a paralelização de um Algoritmo Genético, visando solucionar o Problema de Roteamento de Veículos. Os parâmetros de AG foram configurados de maneira a obter soluções de melhor qualidade. Entretanto, isso causou um aumento significativo no tempo de execução. No intuito de reduzir o tempo de computação, buscou-se a paralelização do algoritmo usando a API OpenMP. Assim, o objetivo deste artigo foi mostrar que além de melhorar o desempenho do AG, sua paralelização não prejudicaria a qualidade das soluções encontradas.

A paralelização atingiu ganhos de *speedups* acima de dois quanto utilizou-se quatro *threads*. Esses valores abaixo do ideal justificam-se devido ao sincronismo presente na implementação do AG, ou seja, na limitação de paralelismo e nas características exploradas do OpenMP. Em relação às políticas de distribuição de iterações da primitiva `#pragma omp for`, a que se demonstrou mais eficiente para as instâncias alvo foi a política *static*. Isso porque para as instâncias que demandam um tempo maior de computação é mais eficiente associar as iterações em tempo de compilação (*static*) do que em tempo de execução (*Dynamic* e *Guided*). Como o tempo de execução do AG é diretamente relacionado com as características da instância-alvo – número de cidades, capacidade de transporte do veículo e demanda por cidade – identificou-se uma grande variação quando se alterna o tamanho das fatias de iterações. Logo, para cada instância faz-se necessário determinar via testes a configuração mais eficiente.

Além dos ganhos de desempenho obtidos com a paralelização, também foi possível encontrar soluções melhores que as encontradas pela versão sequencial. Para a instância `c100`, a melhor solução encontrada na literatura é 65,03 km menor que a melhor solução computada sequencialmente por nossa implementação. Com a paralelização, foi possível reduzir essa diferença em 15,75 km, o que representa uma redução de aproximadamente 24%. Fazendo a mesma analogia para a instância `c120`, a diferença entre a melhor solução conhecida e a computada sequencialmente foi de 27,52 km, sendo que a paralelização permitiu reduzi-la em 3,95 km, ou seja, aproxima-

damente 14%. Entretanto, não há uma relação direta entre as configurações que proporcionam o maior ganho de desempenho com a que computa as melhores soluções.

Buscou-se explorar o uso da paralelização de maneira que ocorressem mais evoluções e, por consequência, que mais indivíduos fossem avaliados, aumentando as chances de se encontrar melhores soluções. Para isso, alterou-se o critério de parada do AG implementado para deixar de contabilizar o número de evoluções e contar o tempo decorrido. Assim, executou-se a versão paralela pelo tempo da sequencial. Obteve-se uma melhora de aproximadamente 5% para a instância `c100` (3,17 km) com um aumento no número de evoluções de até 2,12 vezes e de aproximadamente 1,5% para a instância `c120` (0,41 Km) com um aumento no número de evoluções de até 2,35 vezes. Logo, percebe-se que o aumento no número de evoluções obtido por meio da alteração do critério de parada resultou em uma melhora na qualidade da solução inferior a melhora obtida com o uso do OpenMP na paralelização do AG. Desse modo, podemos concluir que o ganho obtido tanto na redução do tempo de execução quanto na melhora da qualidade das soluções justifica o uso das características do OpenMP na paralelização do AG estudado.

Tendo em vista que o desempenho da paralelização do AG depende das características da instância-alvo, pretendemos realizar futuramente o teste com as demais instâncias do *Benchmark* de Christofides, as quais têm 75, 150 e 199 cidades. Também, como trabalhos futuros, pretende-se usar um processador multicore com um número maior de *cores*, suportando, assim, um número maior de *threads*. Pretende-se explorar outras APIs de paralelização e usar alguma técnica mais elaborada para semear a população inicial, como o uso de coordenadas polares. Adicionalmente, pretende-se melhorar o mecanismo de seleção, procurando evitar vários sorteios consecutivos para o mesmo cruzamento.

Referências

[1] SEVERO, L. C. et al. Simulated annealing to improve analog integrated circuit design: Trade-offs and implementation issues. In: TSUZUKI, M. (Ed.). *Simulated Annealing / Book 1*. Rijeka - Croácia: InTech, 2012. p. 261–284.

[2] REIS, R. et al. A biased random-key genetic algorithm for OSPF and DEFT routing to minimize network congestion. *International Transactions in Operational Research*, v. 18, p. 401–423, 2011.

[3] JOZEFOWIEZ, N.; SEMET, F.; TALBI, E.-G. An evolutionary algorithm for the vehicle routing problem with route balancing. *European Journal of Operational Research*, Elsevier, The Netherlands, v. 195, n. 3, p. 761–769, Jun 2009.

[4] TSAI, C.-W. et al. A time-efficient method for metaheuristics: Using tabu search and tabu ga as a case. In: *Proceedings of the Ninth International Conference on Hybrid Intelligent Systems - Volume 02*. Washington, DC, USA: IEEE Computer Society, 2009. p. 24–29.

[5] LINDEN, R. *Algoritmos Genéticos*. CIDADE, ESTADO: CIÊNCIA MODERNA, 3ª Edição, 2012.

[6] MICHALEWICZ, Z. *Genetic algorithms + data structures = evolution programs*. London, UK: Springer-Verlag, 3rd ed, 1996.

[7] DOROODIAN, S.; GHAEMIAN, N.; SHARIFI, M. Estimating overheads of openmp directives. In: *19th Iranian Conference on Electrical Engineering, 2011*. [S.l.: s.n.], 2011. p. 1 –5.

[8] CHAPMAN, B.; JOST, G.; PAS, R. van der. *Using OpenMP: Portable Shared Memory Parallel Programming*. Cambridge, MA: MIT Press, 2008. (Scientific and Engineering Computation Series).

[9] RAUBER, T.; RüNGER, G. *Parallel Programming: for Multicore and Cluster Systems*. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 364204817X, 9783642048173.

[10] CHRISTOFIDES, N. et al. *Combinatorial optimization*. Chichester, UK: John Wiley, 1979.

[11] KHEIRKHAHZADEH, M.; BARFOROUSH, A. A. A hybrid algorithm for the vehicle routing problem. In: *Proceedings of the Eleventh conference on Congress on Evolutionary Computation*. Piscataway, NJ, USA: IEEE Press, 2009. p. 1791–1798.

[12] LIN, S.-W. et al. Applying hybrid meta-heuristics for capacitated vehicle routing problem. *Expert Systems with Applications*, Tarrytown, NY, USA, v. 36, n. 2, Part 1, p. 1505–1512, Mar 2009.

[13] PRINS, C. A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research*, Coventry, UK, v. 31, n. 12, p. 1985–2002, Oct 2004.

[14] LAPORTE, G. et al. Classical and modern heuristics for the vehicle routing problem. *International Transactions in Operational Research*, Malden, MA, USA, v. 7, n. 4-5, p. 285–300, Sep 2000.

[15] GRESSLER, H. *Aumentando a Eficiência de um Algoritmo Genético através da Paralelização com OpenMP*. 2013. Trabalho de Conclusão de Curso. Graduação em Ciência da Computação - Universidade Federal do Pampa.

[16] MICHALEWICZ, Z. *How to Solve It: Modern Heuristics*. Berlin, Heidelberg: Springer-Verlag, 2nd ed, 2010.

[17] LIU, X.; JIANG, W.; XIE, J. Vehicle routing problem with time windows: A hybrid particle swarn optimization approach. In: *Proceedings of the Fifth International Conference on Natural Computation*. [S.l.]: IEEE Press, 2009. p. 502–506.

[18] KUMAR, N.; KARAMBIR; KUMAR, R. A comparative analysis of pmx, cx and ox crossover operators for solving travelling salesman problem. *International Journal of Latest Research in Science and Technology*, MNK Publication, v. 1, n. 2, p. 98–101, Jul 2012.

[19] ALBA, E.; DORRONSORO, B. A hybrid cellular genetic algorithm for the capacitated vehicle routing problem. In: ABRAHAM, A.; GROSAN, C.; PEDRYCZ, W. (Ed.). *Engineering Evolutionary Intelligent Systems*. Berlin, Germany: Springer Berlin Heidelberg, 2008. p. 379–422.

[20] RAYWARD-SMITH, V. *Modern heuristic search methods*. New York, USA: Wiley, 1st Edition, 1996. 314 p.

[21] NAZIF, H.; LEE, L. S. Optimised crossover genetic algorithm for capacitated vehicle routing problem. *Applied Mathematical Modelling*, Swansea University, Swansea, UK, v. 36, n. 5, p. 2110–2117, May 2012.

[22] WANG, C.-H.; LU, J.-Z. An effective evolutionary algorithm for the practical capacitated vehicle routing problems. *Journal of Intelligent Manufacturing*, Springer Netherlands, Houten, Netherlands, v. 21, p. 363–375, Aug 2010.

[23] DRUMMOND, L. M. A.; OCHI, L. S.; VIANNA, D. S. An asynchronous parallel metaheuristic for the period vehicle routing problem. *Future Generation Computer Systems*, Elsevier, v. 17, p. 379–386, 2001.

[24] YUSSOF, S. et al. A coarse-grained parallel genetic algorithm with migration for shortest path routing problem. In: *Proceedings of the 11th International Conference on High Performance Computing and Communications - HPCC 2009*. [S.l.]: IEEE Press, 2009. p. 615–621.

[25] KARPINSKI, M.; PARCUT, M. Multi-gpu parallel memetic algorithm for capacitated vehicle routing problem. *CoRR*, abs/1401.5216, 2014.

[26] OLIVEIRA, W. de. *Algoritmo Evolutivo Paralelo para o Problema de Atribuição de Localidades a Aneis em redes SONET/SDH*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, 2010.

[27] SILVA, H. H.; PAIVA, C. A. P. S. Avaliação de implementações do algoritmo genético paralelo para solução do problema do caixeiro viajante usando openmp e pthreads. In: *Workshop de Iniciação Científica evento integrante do XIII Simpósio em Sistemas Computacionais WSCAD-SSC 2012*. Petrópolis - RJ/BR: SBC, 2012.