

Universidade Federal do Pampa

Henrique de Oliveira Gressler

Aumentando a Eficiência de um Algoritmo Genético através da Paralelização com OpenMP

Alegrete – RS

2013

Henrique de Oliveira Gressler

Aumentando a Eficiência de um Algoritmo Genético através da Paralelização com OpenMP

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

Orientador: Márcia Cristina Cera

Alegrete – RS

2013


Henrique de Oliveira Gressler


Aumentando a Eficiência de um Algoritmo Genético através da Paralelização com OpenMP

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

Trabalho de Conclusão de Curso defendido e aprovado em 04 de março de 2013.


Márcia Cristina Cera
Orientadora


Fábio Natanael Kepler
UNIPAMPA


Claudio Schepke
UNIPAMPA

Dedico este trabalho aos meus pais e a meu irmão, meus grandes incentivadores.

Agradecimentos

A meus pais, Zair e Wayne, que de tantas formas foram minhas referências, por terem me proporcionado uma infância tranquila, por me incentivarem a crescer, por terem me instigado a ter sede de conhecimento e por me darem ânimo, mesmo que involuntariamente, sempre que foi necessário.

A meu irmão, Fabrício, que mesmo estando distante, sempre se fez presente com palavras de incentivo.

A meus colegas, Arthur Lorenzon, Brandon Santos, Jaline Mombach, Sander Pivetta e Thiarles Medeiros, por terem sido bem mais que colegas nesses 4 anos de faculdade. Por terem sido amigos, companheiros tanto nas horas de estudo, quanto nas horas de lazer em nossos churrascos, viagens e pescarias.

A minha orientadora, Professora Doutora Márcia Cristina Cera, pela dedicação à profissão e ânimo contagiante. Agradeço por ter sido uma orientadora de fato, por ter indicado o caminho e não ter me deixado esmorecer nesta batalha. Com sua Carreira Acadêmica brilhante, és um grande exemplo a ser seguido por todos os alunos de graduação que almejam a docência.

A meus colegas de trabalho, Tarcísio Kercher, Marcelo Schenckel, Willian Fagundes e Iorram Padilha, pela amizade, compreensão, incentivo e por acreditarem na minha capacidade. Agradeço por serem os melhores colegas de trabalho que eu poderia ter.

A meus amigos, que de tantas formas me incentivaram ao longo dessa jornada e fizeram meu caminho menos árduo.

“O sucesso é ir de fracasso em fracasso sem perder o entusiasmo.”
(Winston Churchill)

Resumo

Este trabalho propõe a implementação e paralelização de um Algoritmo Genético (AG) para resolver o Problema do Roteamento de Veículos (PRV). O PRV é um problema combinatório pertencente à classe dos NP-Completo. Em sua forma canônica, consiste em rotear veículos que possuem uma capacidade de transporte para determinado produto. Os veículos devem atender as cidades distribuídas em um mapa, que possuem certa demanda para o produto transportado. A solução do problema passa pela determinação de rotas que minimizem a distância total percorrida e, que atenda todas as demandas das cidades do mapa, sendo que cada cidade deve ser visitada apenas uma vez. Resolver problemas combinatórios deste tipo traz avanços em várias áreas do conhecimento, como a concepção de circuitos integrados e a realização de testes combinatórios aplicados aos protocolos de redes de computadores. O objetivo geral deste trabalho é implementar um AG capaz de encontrar bons resultados para o PRV em um tempo de computação razoável e, através da paralelização com OpenMP, obter resultados compatíveis com os encontrados pela versão sequencial e, melhores se executado pelo tempo da versão sequencial. Nesse sentido, foram testados os diversos parâmetros de um AG, com objetivo de determinar qual combinação dos mesmos conduz a melhores resultados. Nesta busca ficou evidenciado que a variação do tamanho da população e do número de evoluções influencia significativamente no tempo necessário para encontrar uma solução e também na qualidade desta solução. Baseado na implementação sequencial, foi desenvolvida uma versão paralela usando a *Application Program Interface* (API) OpenMP. A paralelização consistiu em encontrar regiões que pudessem ser executadas em paralelo e eleger as diretivas de paralelização baseado nas características de cada região. Também foram testadas várias combinações de políticas de escalonamento e granularidade, para determinar quais as mais eficientes. Os melhores *speedups* foram obtidos pela versão com 4 *threads*, mantendo a qualidade das soluções compatível com o encontrado pela versão sequencial. Executar a versão paralela com 4 *threads* pelo tempo da versão sequencial, permitiu avaliar um número de indivíduos aproximadamente 100% maior, possibilitando encontrar melhores soluções.

Palavras-chave: Algoritmos Genéticos, PRV, Problema do Roteamento de Veículos, Paralelização, OpenMP, meta-heurísticas.

Abstract

This work proposes the implementation and parallelization of a Genetic Algorithm (GA) to solve the VRP. The VRP is a combinatorial problem which belongs to the class of NP-Complete. The VRP in its canonical form, consists to route vehicles that have a carrying capacity for a particular product. Vehicles must get to the cities distributed on a map, which have certain demand for the carried product. The solution to the problem is to identify routes that minimize the total distance traveled, and that meets all the demands of the cities on the map, and each city must be visited only once. Solving this kind of combinatorial problems brings advances in various areas of knowledge such as the design of integrated circuits and combinatorial testing protocols applied to computer networks. This work showed that the variation in population size and number of evolutions significantly influences the time needed to find a solution and also in the quality of this solution. Based on the sequential implementation, we developed a parallel version using the OpenMP API. The parallelization consisted on finding areas that could be executed in parallel and electing parallelization policies based on the characteristics of each region. We also tested several combinations of scheduling policies and granularity to determine the most efficient for the application. The best speedups were obtained by the version with 4 threads, which when executed by sequential version time, allowed to evaluate a number of individuals approximately 100% greater, enabling the finding of better solutions.

Key-words: Genetic Algorithms, VRP, Parallelization, OpenMP, meta-heuristics.

Lista de ilustrações

Figura 1	Exemplo de rotas.	26
Figura 2	Um exemplo de rotas no PRV	28
Figura 3	Ilustração de um cruzamento de 1 ponto.	33
Figura 4	Ilustração de um cruzamento de 2 pontos.	33
Figura 5	Ilustração de um cruzamento uniforme.	34
Figura 6	Mutação utilizando a técnica de Troca ou <i>swap</i>	35
Figura 7	Mutação utilizando a técnica de Inversão e Inserção.	36
Figura 8	Representação do modelo <i>fork-join</i>	37
Figura 9	Instância c50 submetida a versão Sequencial 1 e a versão Sequencial 2 .	50
Figura 10	Instância c100 submetida a versão Sequencial 1 e a versão Sequencial 2	50
Figura 11	Instância c50 submetida a versões com diferentes taxas de mutação . .	52
Figura 12	Instância c100 submetida a versões com diferentes taxas de mutação .	52
Figura 13	Instância c50 submetida a versões com diferentes técnicas de mutação .	53
Figura 14	Instância c100 submetida a versões com diferentes técnicas de mutação	54
Figura 15	Instância c50 submetida a versões com diferentes técnicas de cruzamento	55
Figura 16	Instância c100 submetida a versões com diferentes técnicas de cruzamento	56
Figura 17	Instância c120 submetida a versões com diferentes técnicas de cruzamento	57
Figura 18	Fluxograma do algoritmo genético sequencial implementado.	67
Figura 19	Instâncias c50, c100 e c120 submetidas às versões paralelas até 8 <i>threads</i> : <i>Speedup</i>	70
Figura 20	Instâncias c50, c100 e c120 submetidas às versões paralelas até 8 <i>threads</i> : Eficiência.	71
Figura 21	Instância c50 submetida à versão paralela com <i>schedule</i> estático	72
Figura 22	Instância c50 submetida à versão paralela com <i>schedule</i> dinâmico	72
Figura 23	Instância c50 submetida à versão paralela com <i>schedule</i> guiado	73
Figura 24	Instância c100 submetida à versão paralela com <i>schedule</i> estático	74
Figura 25	Instância c100 submetida à versão paralela com <i>schedule</i> dinâmico	75
Figura 26	Instância c100 submetida à versão paralela com <i>schedule</i> guiado	75
Figura 27	Instância c120 submetida à versão paralela com <i>schedule</i> estático	76
Figura 28	Instância c120 submetida à versão paralela com <i>schedule</i> dinâmico	76
Figura 29	Instância c120 submetida à versão paralela com <i>schedule</i> guiado	77
Figura 30	Instância c50 submetida à versão sequencial 2	79

Figura 31	Instância c50 submetida à versão com 2 <i>threads</i> , <i>schedule</i> dinâmico e <i>chunk</i> 25%	80
Figura 32	Instância c50 submetida à versão com 3 <i>threads</i> , <i>schedule</i> dinâmico e <i>chunk</i> 33%	80
Figura 33	Instância c50 submetida à versão com 4 <i>threads</i> , <i>schedule</i> dinâmico e <i>chunk</i> 1	81
Figura 34	Instância c100 submetida à versão sequencial 2	82
Figura 35	Instância c100 submetida à versão com 2 <i>threads</i> , <i>schedule</i> estático e <i>chunk</i> 25%	83
Figura 36	Instância c100 submetida à versão com 3 <i>threads</i> , <i>schedule</i> estático e <i>chunk</i> 1	83
Figura 37	Instância c100 submetida à versão com 4 <i>threads</i> , <i>schedule</i> estático e <i>chunk</i> 1	84
Figura 38	Instância c120 submetida à versão sequencial 2	85
Figura 39	Instância c120 submetida à versão com 2 <i>threads</i> , <i>schedule</i> guiado e <i>chunk</i> 1	86
Figura 40	Instância c120 submetida à versão com 3 <i>threads</i> , <i>schedule</i> estático e <i>chunk</i> 10%	87
Figura 41	Instância c120 submetida à versão com 4 <i>threads</i> , <i>schedule</i> estático e <i>chunk</i> 25%	88
Figura 42	Instância c50 submetida à versão com 2 <i>threads</i> executadas pelo mesmo tempo da versão sequencial, <i>schedule</i> dinâmico e <i>chunk</i> 25%	89
Figura 43	Instância c50 submetida à versão com 3 <i>threads</i> executadas pelo mesmo tempo da versão sequencial, <i>schedule</i> dinâmico e <i>chunk</i> 33%	90
Figura 44	Instância c50 submetida à versão com 4 <i>threads</i> executadas pelo mesmo tempo da versão sequencial, <i>schedule</i> dinâmico e <i>chunk</i> 1	90
Figura 45	Instância c100 submetida à versão com 2 <i>threads</i> executadas pelo tempo sequencial, <i>schedule</i> estático e <i>chunk</i> 25%	92
Figura 46	Instância c100 submetida à versão com 3 <i>threads</i> executadas pelo tempo sequencial, <i>schedule</i> estático e <i>chunk</i> 1	92
Figura 47	Instância c100 submetida à versão com 4 <i>threads</i> executadas pelo tempo sequencial, <i>schedule</i> estático e <i>chunk</i> 1	93
Figura 48	Instância c120 submetida à versão com 2 <i>threads</i> executadas pelo tempo sequencial, <i>schedule</i> guiado e <i>chunk</i> 1	95
Figura 49	Instância c120 submetida à versão com 3 <i>threads</i> executadas pelo tempo sequencial, <i>schedule</i> estático e <i>chunk</i> 10%	95
Figura 50	Instância c120 submetida à versão com 4 <i>threads</i> executadas pelo tempo sequencial, <i>schedule</i> estático e <i>chunk</i> 25%	96

Lista de tabelas

Tabela 1	Tempo aproximado necessário para calcular o custo de todas as rotas para n cidades.	27
Tabela 2	Menores custos (em Km) conhecidos para as instâncias usadas neste trabalho.	46
Tabela 3	Impacto do aumento da população e número de evoluções na qualidade da solução.	48
Tabela 4	Técnica de mutação que conduziu as instâncias c50 e c100 a menores custos	53
Tabela 5	Técnica de cruzamento que conduziu as instâncias c50 e c100 a menores custos médios	55
Tabela 6	Custo médio e a técnica de cruzamento empregada para a instância c120	58
Tabela 7	Tempo médio de computação para as instâncias c50, c100 e c120. . . .	58
Tabela 8	Impacto das funções no tempo total de execução do programa, obtido pelo <i>gprof</i>	64
Tabela 9	Quantidade de memória necessária para alocar a matriz distância. . . .	65
Tabela 10	Impacto das funções no tempo total de execução da nova versão do programa, obtido pelo <i>gprof</i>	65
Tabela 11	Comparação entre os tempos médios de computação	66
Tabela 12	Configuração das versões paralelas que obtiveram os melhores <i>speedups</i>	78
Tabela 13	Comparativo entre as versões com melhor <i>speedup</i> para instância c50 .	81
Tabela 14	Comparativo entre as versões com melhor <i>speedup</i> para instância c100 .	84
Tabela 15	Comparativo entre as versões com melhor <i>speedup</i> para instância c120 .	86
Tabela 16	Comparativo entre as versões com melhor <i>speedup</i> para instância c50, executadas pelo tempo sequencial	91
Tabela 17	Comparativo entre as versões com melhor <i>speedup</i> para instância c100, executadas pelo tempo sequencial	94
Tabela 18	Comparativo entre as versões com melhor <i>speedup</i> para instância c120, executadas pelo tempo sequencial	97

Lista de siglas

AE Algoritmo Evolutivo

AG Algoritmo Genético

AGH Algoritmo Genético Híbrido

AGP Algoritmo Genético Paralelo

API *Application Program Interface*

ARB *Architecture Review Board*

AS Anelamento Simulado

BT Busca Tabu

CVRP *Capacitated Vehicle Routing Problem*

GA *Genetic Algorithm*

GCC *GNU Compiler Collection*

IA Inteligência Artificial

OX *Order Crossover*

PCV Problema do Caixeiro Viajante

PRV Problema do Roteamento de Veículos

SA *Simulated Annealing*

SPMD *Single Program Multiple Data*

TS *Tabu Search*

TSP *Traveling Salesman Problem*

VRP *Vehicle Routing Problem*

Sumário

1	Introdução	23
1.1	Objetivo	24
1.2	Organização	24
2	Revisão da Literatura	25
2.1	Introdução	25
2.2	O Problema do Roteamento de Veículos	25
2.2.1	O Problema do Caixeiro Viajante	25
2.2.2	Busca Exaustiva aplicada ao PCV	25
2.2.3	Problema do Roteamento de Veículos	27
2.3	Meta-heurísticas	29
2.3.1	<i>Simulated Annealing</i>	29
2.3.2	Busca Tabu	29
2.3.3	Algoritmos Genéticos	30
2.4	Detalhando Algoritmos Genéticos	30
2.4.1	Pseudocódigo de um Algoritmo Genético	30
2.4.2	Função Objetivo	31
2.4.3	Operadores Genéticos	32
2.4.3.1	Seleção	32
2.4.3.2	Cruzamento	32
2.4.3.3	Mutação	33
2.5	Computação paralela	35
2.5.1	OpenMP	36
2.5.2	Diretivas OpenMP utilizadas neste trabalho	37
2.5.2.1	Regiões Paralelas	38
2.5.2.2	<i>Sections</i>	38
2.5.2.3	Laços Paralelos	39
2.5.2.4	Cláusulas	39
2.6	Trabalhos Relacionados	40
2.7	Considerações sobre a Revisão Bibliográfica	42
3	Implementação do Algoritmo Genético	45
3.1	Introdução	45
3.2	Definição das Características do PRV	45
3.3	Algoritmo Genético Implementado	46
3.3.1	Visão Geral	46

3.3.2	Tamanho da População e Número de Evoluções	47
3.3.3	Taxa de Cruzamento e Probabilidade de Mutação	48
3.3.4	Taxa de Mutação	50
3.3.5	Técnica de Mutação	51
3.3.6	Técnica de Cruzamento	53
3.3.7	Parâmetros para Instância c120	56
3.4	Tempo de Computação	57
3.5	Considerações sobre o Algoritmo Genético Implementado	59
4	Paralelização do Algoritmo Genético	61
4.1	Introdução	61
4.2	Análise da Versão Sequencial do Algoritmo Genético	61
4.2.1	Funções Implementadas	61
4.2.2	Perfil da Execução	63
4.2.3	Melhorando o Desempenho Sequencial	64
4.3	O que pode ser paralelizado?	66
4.4	Implementação da versão Paralela	68
4.4.1	Definindo as Regiões Paralelas	68
4.4.2	Métricas de Desempenho	69
4.4.3	Variando o Número de Threads	69
4.4.4	Variando a Forma de Escalonamento	71
4.5	Qualidade da Solução	78
4.5.1	Instância c50	79
4.5.2	Instância c100	82
4.5.3	Instância c120	84
4.6	Executando a versão Paralela pelo tempo da Sequencial	88
4.6.1	Instância c50	88
4.6.2	Instância c100	91
4.6.3	Instância c120	94
4.7	Considerações sobre a Paralelização do Algoritmo Genético	97
5	Conclusão	99
	Referências	101

1 Introdução

O Problema do Roteamento de Veículos (**PRV**) é um problema combinatório intratável muito estudado e normalmente utilizado para validar heurísticas justamente porque não há um método eficiente para resolvê-lo em tempo polinomial. A resolução deste problema é de grande interesse, principalmente do setor de transportes, pois a minimização das rotas implica em menores custos e maior lucratividade para o setor. Invariavelmente sua resolução passa por algum método heurístico ou por alguma meta-heurística. Enquanto heurísticas são algoritmos que usam métodos aproximados para resolver problemas, meta-heurísticas são algoritmos genéricos que combinam métodos heurísticos de diversas fontes e, normalmente são aplicadas a busca de soluções para problemas de otimização e combinatórios (**FREITAS et al., 2009; BLUM; ROLI, 2003**). Meta-heurísticas são empregadas com êxito sempre que não exista um algoritmo específico para resolver o problema que está sendo tratado (**BLUM; ROLI, 2003**). Normalmente encontram um resultado satisfatório (sem a garantia de que seja um resultado ótimo) em um tempo aceitável. Segundo **Rayward-Smith et al. (1996)**, muitos métodos heurísticos foram propostos nos últimos 30 anos, mas muitos deles adaptados a um problema em particular. Em contraste, três métodos genéricos, que podem ser aplicados a diversos problemas (meta-heurísticas), tornaram-se muito populares: Anelamento Simulado (**AS**), Busca Tabu (**BT**) e Algoritmos Genéticos (**AGs**). Devido a limitação de tempo, este trabalho abordará apenas **AGs**.

Algoritmos Genéticos (**AGs**), de acordo com **Linden (2012)**, **Stützle (1999)** e **Michalewicz (1996)** são um tipo de Algoritmo Evolutivo (**AE**), baseado em técnicas heurísticas de otimização global. **AGs** são inspirados na Teoria da Evolução das Espécies de Charles Darwin, que prega a ideia da sobrevivência do indivíduo mais apto. As possíveis soluções são chamadas de indivíduos e são organizadas num grupo chamado população. Os indivíduos dessa população podem ser selecionados e cruzados, gerando novos indivíduos com características dos genitores. Ao longo das gerações, uma população vai substituindo a outra. Com os constantes cruzamentos o algoritmo encontra, através de uma função de avaliação, soluções que cada vez mais se aproximam do aceitável. Depois de um número determinado de evoluções, o algoritmo termina a sua execução e apresenta o melhor indivíduo encontrado.

Buscando solucionar o **PRV**, este trabalho propõe a implementação de um **AG**. Na concepção deste algoritmo foram testados vários parâmetros, como tamanho da população, técnica de cruzamento e taxa de mutação, na busca de combinações que conduzem a melhores resultados. Entretanto, constatou-se que os melhores resultados só eram atin-

gidos após um longo período de processamento. A paralelização do algoritmo tornou-se a alternativa mais viável para melhorar o tempo de computação e ainda assim atingir as melhores soluções.

Para paralelizar o Algoritmo Genético implementado, este trabalho propõe o uso de técnicas de programação para memória compartilhada com a *Application Program Interface (API)* OpenMP, que de acordo com Rauber e Rüniger (2010) e Wilkinson e Allen (2005), é um padrão portátil para programação de sistemas de memória compartilhada. O OpenMP provê diversas diretivas de compilação, biblioteca de rotinas e variáveis de ambiente. Ele pode ser usado para estender as linguagens de programação C, C++ e Fortran com construções do tipo *Single Program Multiple Data (SPMD)*.

1.1 Objetivo

O objetivo geral deste trabalho é implementar um AG capaz de encontrar bons resultados para o PRV em um tempo de computação razoável e, através da paralelização, conceber uma versão capaz de obter resultados compatíveis com a versão sequencial, em um menor período de tempo e resultados melhores se executado pelo tempo da versão sequencial.

Para alcançar o objetivo geral, será necessário antes atingir os seguintes objetivos específicos:

- Investigar os diversos parâmetros de um AG, para determinar qual combinação conduz a melhores soluções;
- Paralelizar o AG implementado, visando diminuir o tempo de computação necessário para resolver o PRV;
- Analisar a qualidade das soluções encontradas pelas versões implementadas do AG.

1.2 Organização

Este trabalho está organizado em 5 capítulos. O Capítulo 1 apresenta uma introdução ao tema, o objetivo do trabalho e a organização do texto. O Capítulo 2 traz uma breve revisão da literatura sobre os temas abordados ao longo do trabalho. O Capítulo 3 traz a implementação do AG para resolver o PRV. O Capítulo 4 traz uma análise e os esforços de paralelização da meta-heurística implementada, bem como os resultados obtidos. O Capítulo 5 apresenta uma conclusão do trabalho.

2 Revisão da Literatura

2.1 Introdução

Este capítulo tem por objetivo apresentar uma revisão de todos os conceitos e técnicas que serão empregados nesse trabalho. Ele está estruturado da seguinte maneira: a Seção 2 descreve o Problema Alvo, a Seção 3 trata das Meta-heurísticas, na Seção 4 há uma descrição dos Algoritmos Genéticos, na Seção 5 há uma revisão sobre Computação Paralela, na Seção 6 alguns trabalhos relacionados são apresentados e na Seção 7 há as considerações sobre o capítulo.

2.2 O Problema do Roteamento de Veículos

Para entendermos o problema alvo, faremos antes um breve estudo sobre o Problema do Caixeiro Viajante (PCV) que é o problema do qual o PRV deriva.

2.2.1 O Problema do Caixeiro Viajante

O PCV ou em inglês *Traveling Salesman Problem* (TSP) é um dos problemas mais estudados e importantes da análise combinatória. De acordo com [Rayward-Smith et al. \(1996\)](#) esse problema exerce grande fascínio entre os pesquisadores, por ser de fácil entendimento, mas de difícil resolução.

O caixeiro viajante deve visitar todas as cidades do território, passando apenas uma vez em cada uma delas e logo após, retornar para o ponto de partida. Sabendo do custo para viajar entre cada par de cidades, a tarefa é encontrar a rota (*tour*) que ofereça menor custo ([MICHALEWICZ; FOGEL, 2004](#); [STEEMAN, 2012](#)).

2.2.2 Busca Exaustiva aplicada ao PCV

Uma busca exaustiva consiste em enumerar todas as possíveis soluções para um problema dado e então testá-las uma a uma. O objetivo é encontrar, entre todas as soluções possíveis, a melhor delas, que representará o ótimo global para o problema.

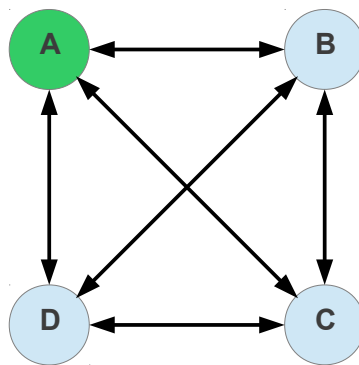
Para acharmos o número total de rotas $R(n)$, [Silveira \(2000\)](#) explica que devemos seguir o seguinte raciocínio: tomando como exemplo 4 cidades ($n = 4$), a cidade de partida é a mesma de chegada e não afeta o cálculo. Se partirmos de uma cidade qualquer,

devemos adicionar qualquer uma das outras três restantes, logo depois qualquer uma das outras duas restantes, em seguida a última cidade restante, e por fim, retornar à cidade inicial.

Considerando que as cidades são A, B, C e D, como representado na Figura 1 e, que partimos da cidade A, teríamos os seguintes conjuntos de rotas possíveis (SILVEIRA, 2000):

- 1- ABCDA
- 2- ABDCA
- 3- ACBDA
- 4- ACDBA
- 5- ADBCA
- 6- ADCBA

Figura 1 – Exemplo de rotas.



O número de rotas $R(n)$ é dado por $(n - 1)!$ que no exemplo acima é $(4 - 1)! = 3 \times 2 \times 1 = 6$. Se aumentarmos o número de cidades, o cálculo de todas as rotas possíveis torna-se inviável rapidamente. Para o caso de 20 cidades, precisamos de 20 operações de soma para calcular a distância percorrida em cada rota. Considerando um computador que faça 50 bilhões de adições por segundo, poderíamos calcular o custo de 2,5 bilhões $(5 \times 10^{10} \div 20 = 2,5 \times 10^9)$ de rotas por segundo. O número total de rotas é dado por $19!$ que equivale a aproximadamente $1,2164 \times 10^{17}$. O tempo necessário para avaliar todas as rotas é de $1,2164 \times 10^{17} \div 2,5 \times 10^9$ que equivale a $4,8656 \times 10^7$ segundos ou 1 ano e meio, aproximadamente. A Tabela 1 mostra o tempo necessário para calcular todas as rotas possíveis para algumas entradas n .

Conforme observamos na Tabela 1, Cunha (2002) relata que o aumento de n provoca um pequeno decréscimo no número de rotas que podem ser calculadas em 1 segundo, mas provoca um aumento muito grande no número de rotas possíveis e no tempo total de cálculo. Desta forma, torna-se inviável uma busca exaustiva e os métodos de resolução passam pelas meta-heurísticas que embora não garantam uma solução ótima,

Tabela 1 – Tempo aproximado necessário para calcular o custo de todas as rotas para n cidades.

n	Rotas por segundo	Rotas possíveis	Tempo total
5	10 bilhões	24	insignificante
10	5 bilhões	362880	insignificante
15	3,333 bilhões	$8,71 \times 10^{10}$	26 segundos
20	2,5 bilhões	$1,21 \times 10^{17}$	1,5 anos
25	2 bilhões	$6,2 \times 10^{23}$	9,83 milhões de anos
30	1,666 bilhões	$8,84 \times 10^{30}$	172 trilhões de anos

garantem um tempo de computação hábil.

2.2.3 Problema do Roteamento de Veículos

O PRV ou em inglês *Vehicle Routing Problem (VRP)*, conforme Steeman (2012), Baldacci, Toth e Vigo (2007) e Ochi et al. (1998) é uma generalização do PCV. Uma das diferenças é que o PRV admite mais de um veículo, o que de acordo com Wang e Lu (2010), não é uma restrição. Segundo Jozefowicz, Semet e Talbi (2009), a versão elementar do VRP também é conhecida como *Capacitated Vehicle Routing Problem (CVRP)*.

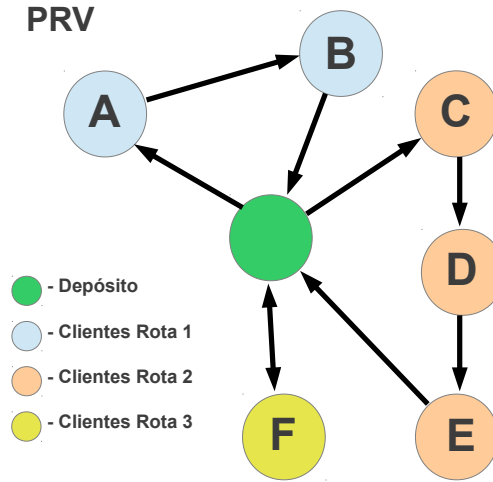
No PRV, cada cliente a ser visitado possui uma demanda para determinado produto e os veículos possuem uma capacidade de transporte limitada. O veículo deve partir de um ponto para atender os clientes e retornar a esse mesmo ponto, o depósito. O objetivo é obter um conjunto de rotas que satisfaça a demanda dos clientes, e que o somatório dessas rotas tenha um custo total mínimo. Cada cliente deve ser visitado apenas uma vez, e por apenas um veículo (STEEMAN, 2012; JOZEFOWIEZ; SEMET; TALBI, 2009; NAZIF; LEE, 2012; PRINS, 2004).

Na Figura 2, o depósito é representado por um círculo verde e, é de onde o veículo sai para atender as demandas dos clientes e para onde ele retorna após atender todos os clientes da rota. Percebe-se que para atender todos os clientes do grafo, foram necessárias 3 rotas: uma para atender os clientes representados em Azul (A e B), outra para atender os clientes representados em laranja (C, D e E) e uma última para atender o cliente representado em amarelo (F).

De acordo com Wink, Back e Emmerich (2012) o PRV pode ser representado pelo grafo completo e simétrico $G(V, E)$ onde:

- n é o número de clientes;
- $V = \{v_0, v_1, \dots, v_n\}$ representa o depósito (v_0) e o conjunto de todos os clientes ($V - v_0$);
- $E = \{(v_i, v_j) \in V : i \neq j\}$ é o conjunto de todos os arcos que ligam um cliente a outro;

Figura 2 – Um exemplo de rotas no PRV



- $d(v_i) : i \in \{1, \dots, n\}$ representa a demanda do cliente i ;
- $c_{ij} = \delta(v_i, v_j) = \sqrt{|v_{i_x} - v_{j_x}|^2 + |v_{i_y} - v_{j_y}|^2}$ é o custo de viagem entre os vértices v_i e v_j . Assume-se que $c_{ij} = c_{ji}$ e que $c_{ii} = 0$;
- $R_i : i \in \{1, \dots, m\} = \{v_0, v_{i1}, \dots, v_{i_{k(i)}}, v_0\}$ é uma rota para um veículo. Cada rota inicia e termina no depósito;
- $S = \{R_1, \dots, R_m\}$ é o conjunto de rotas que formam a solução;
- O custo da viagem para R_i é dado por: $Cost(R_i) = \sum_{j=1}^{k(i)-1} (c_{i_j i_{j+1}}) + c_{0i_1} + c_{i_{k(i)}0}$;
- Cada cliente é visitado uma única vez por um veículo: $\forall i (v_i \in V - v_0 \Rightarrow \exists j : v_i \in R_j \wedge \nexists j \neq k : v_i \in R_j \wedge v_i \in R_k)$;
- A demanda total da rota não excede a capacidade K do veículo: $\forall i \in \{1, \dots, m\} : \sum_{j=i}^{k(i)} d(v_{i_j}) \leq K$;

O objetivo é minimizar a distância total de viagem: $Cost(S) = \sum_{i=1}^m Cost(R_i)$.

O PRV pode ser facilmente reduzido ao PCV. Para isso, bastaria considerarmos um veículo com capacidade de transporte infinita (STEEMAN, 2012; BALDACCI; TOTH; VIGO, 2007). Assim, o veículo partiria do depósito, visitaria todos os clientes e retornaria ao depósito, buscando acumular o menor custo possível para a rota. Esse é exatamente o objetivo do PCV.

Passaremos agora a uma breve revisão sobre as meta-heurísticas mais utilizadas na resolução de problemas de otimização combinatória.

2.3 Meta-heurísticas

Heurística deriva do verbo *heuriskein* que significa “encontrar”, enquanto o prefixo meta significa “além de, em um nível superior”.

Um algoritmo de aproximação ou heurístico é um tipo de algoritmo que tenta encontrar uma solução boa (aceitável) para um problema de otimização. O termo meta-heurística refere-se a um algoritmo heurístico ou aproximado de otimização que não foi desenvolvido para um problema específico. Meta-heurísticas são genéricas e podem ser aplicadas para resolver diversos tipos de problemas (BLUM et al., 2011; BLUM; ROLI, 2003).

Yagiura e Ibaraki (1996) dizem que meta-heurísticas apresentam desempenho muito melhor que as heurísticas simples, mesmo que profundas propriedades matemáticas do domínio do problema não estejam disponíveis. Para Yagiura e Ibaraki (1996), um dos atrativos das meta-heurísticas é que elas são simples e robustas. Tsai et al. (2009) dizem que se executarmos uma meta-heurística por tempo razoável, o resultado mesmo não sendo ideal, é aceitável.

De acordo com Rayward-Smith et al. (1996), as seguintes meta-heurísticas tornaram-se muito populares: *Simulated Annealing* (SA), *Tabu Search* (TS) e *Genetic Algorithms* (GAs). Veremos a seguir uma breve descrição sobre SA e TS, e nos aprofundaremos em GA por ser a meta-heurística utilizada neste trabalho.

2.3.1 *Simulated Annealing*

Essa técnica, inspirada na termodinâmica, é uma analogia entre o anelamento dos sólidos e os problemas de otimização combinatorial. Quando um sólido é aquecido até seu ponto de fusão e então passa a ser resfriado, a qualidade estrutural do sólido está ligada à velocidade com que o mesmo é resfriado (STÜTZLE, 1999; RAYWARD-SMITH et al., 1996).

SA é uma técnica que faz buscas no conjunto total de soluções do problema, minimizando a chance de ficar preso num máximo local. Um máximo local é uma solução que parece ser a melhor quando comparada a uma vizinhança próxima, mas é pior que outras mais distantes no espaço de busca. SA consegue sair de máximos locais através de um controle randômico de busca em soluções de pior qualidade (RAYWARD-SMITH et al., 1996; BLUM; ROLI, 2003).

2.3.2 Busca Tabu

Baseia-se na busca de vizinhança, conforme relatam Rayward-Smith et al. (1996), Stützle (1999) e Gómez Arthur T. e Galafassi (2011), mas de uma forma determinista, guardando as últimas decisões tomadas num passado recente da pesquisa, que passam a ser proibidas para um certo número de iterações futuras. TS normalmente faz uma busca

local agressiva, procurando o melhor movimento possível, mesmo que isso piore o valor da função objetivo.

O objetivo do **TS**, é encorajar a exploração do espaço de soluções visitando partes que ainda não tenham sido visitadas anteriormente. Essa exploração é garantida através da proibição do movimento reverso que passa a ser um “tabu” por um determinado número de iterações (**RAYWARD-SMITH et al., 1996**).

2.3.3 Algoritmos Genéticos

AG ou **GA** recebem esse nome porque são inspirados na teoria evolucionista de Charles Darwin. Esses algoritmos são muito utilizados para resolver problemas de busca e otimização. Segundo a teoria de Darwin, as populações das espécies evoluíram na natureza, seguindo os princípios da seleção natural e da sobrevivência dos indivíduos melhor adaptados (**LINDEN, 2012; STÜTZLE, 1999**).

Imitando o comportamento das espécies na natureza, cada possível solução para um problema é considerada um indivíduo. **Pacheco (1999)** diz que na área computacional, um cromossomo é uma estrutura de dados que representa uma das possíveis soluções do espaço de busca do problema. Cromossomos podem ser avaliados, selecionados, recombinados (*crossover*) e sofrer mutação. Assim, após algumas gerações, a população terá evoluído para indivíduos mais aptos.

De acordo com **Fernandes (2005)**, Algoritmos Genéticos são menos eficientes que técnicas especializadas para resolver determinado problema. Por outro lado, podem obter bons resultados quando aplicados a problemas que não possuem uma técnica especializada para resolvê-lo. Por este motivo, este trabalho utilizará Algoritmos Genéticos para a resolução do **PRV**.

2.4 Detalhando Algoritmos Genéticos

O ponto crucial para o uso de Algoritmos Genéticos é a codificação do problema em cromossomos. Essa codificação juntamente com a função de avaliação é o que liga o **AG** ao problema (**LINDEN, 2012**).

2.4.1 Pseudocódigo de um Algoritmo Genético

No Algoritmo 1, é mostrado o pseudocódigo de um Algoritmo Genético Simples (**LINDEN, 2012; FERNANDES, 2005**):

Inicialmente o algoritmo cria uma população inicial (linha 2). Logo após, na linha 3, cada indivíduo da população inicial é avaliado pela função objetivo para determinar o grau de aptidão de cada um. O controle da evolução do **AG** está compreendido entre

Algoritmo 1 Algoritmo Genético Simples

```
1: Início (AG Simples)
2: Gera população inicial;
3: Avalia de todos os indivíduos da população inicial;
4: while ( !Terminou ) do
5:   Início (Nova geração)
6:   for (Tamanho da população)/2 do
7:     Seleciona com certa probabilidade 2 indivíduos da população anterior;
8:     Realiza cruzamento obtendo descendentes;
9:     com certa probabilidade realiza mutação nos descendentes;
10:    Avalia descendentes;
11:    Inclui os descendentes na nova geração;
12:  end for
13:  Fim (Nova geração);
14:  if (População convergiu) then
15:    Terminou ← Verdade
16:  end if
17: end while
18: Fim (AG Simples)
```

as linhas 4 e 17. O laço da linha 6 a 12 controla o número de cruzamentos que serão necessários para criar uma nova população. A cada iteração o AG elege dois indivíduos como pais (linha 7) e, a carga genética deles é combinada através do cruzamento (linha 8), formando novos indivíduos, que herdam características de cada um dos genitores. Com certa probabilidade, os novos indivíduos gerados sofrem mutação (linha 9). Os novos indivíduos são então avaliados pela função objetivo (linha 10) para determinar sua aptidão e logo após (linha 11) são inseridos na nova população. Na linha 14 há um teste que determina se o algoritmo continua a evolução ou encerra a sua execução através da validação da condição de parada do laço que controla as evoluções. Uma vez encerradas as evoluções, o indivíduo mais apto é apresentado como melhor solução encontrada para o problema que está sendo tratado.

2.4.2 Função Objetivo

Para Fernandes (2005) a determinação de uma função objetivo adequada é um aspecto fundamental para o comportamento do AG. Esta função é usada para avaliar os indivíduos da população e também pode ser chamada de função de avaliação. No Algoritmo 2, ela é usada nas linhas 3 e 10. De acordo com Linden (2012) a função de avaliação calcula um número que representa quão boa é a solução representada pelo cromossomo avaliado. Se a solução apresentada viola alguma restrição do problema, ela deve ser penalizada de forma que esta solução tenha uma avaliação pior em relação as soluções que não violam nenhuma restrição. Dada a generalidade dos AG, a função de avaliação muitas vezes é a única ligação entre o AG ao problema real.

2.4.3 Operadores Genéticos

De acordo com Michalewicz (1996) os operadores genéticos são os responsáveis por organizar a composição genética dos filhos durante a reprodução. Esses operadores são a seleção, o cruzamento (*crossover*) e a mutação. Esses operadores estão representados no Algoritmo 2, nas linhas 7, 8 e 9 respectivamente. Enquanto a seleção determina quem será escolhido como genitor, o cruzamento recombina a composição genética dos genitores, gerando novos indivíduos, e a mutação age apenas em uma parcela pequena dos novos indivíduos gerados, modificando aleatoriamente alguns genes. Vejamos eles em detalhes:

2.4.3.1 Seleção

A seleção tenta de alguma forma imitar a natureza, onde os indivíduos mais aptos tem mais chances de serem selecionados e por consequência passar seu material genético para as futuras gerações. Para Fernandes (2005) o método mais utilizado é o da seleção proporcional onde cada indivíduo tem uma probabilidade de ser selecionado baseado no valor da sua função objetivo.

Linden (2012) diz que devemos de alguma forma privilegiar os indivíduos com função objetivo alta, sem desprezar os indivíduos com função objetivo baixa, já que necessitamos de diversidade genética. Se somente permitirmos o cruzamento dos melhores indivíduos, a população ficará cada vez mais semelhante e, faltará diversidade genética para encontrar boas soluções. Este efeito recebe o nome de convergência genética.

Devemos levar em consideração que um indivíduo com função objetivo baixa, pode possuir características que combinadas com outro indivíduo qualquer, formem indivíduos que tenham função objetivo altas.

2.4.3.2 Cruzamento

Cruzamento é a técnica utilizada para recombinar a carga genética de dois indivíduos, gerando descendentes que tenham características dos dois genitores. A seguir são apresentados 3 tipos básicos de cruzamento:

- **Cruzamento de 1 ponto:** consiste em escolher um ponto aleatório para seccionar os cromossomos dos genitores e, logo após, recombiná-los intercalando as partes de um genitor com as de outro, formando novos cromossomos completos, como mostrado na Figura 3.
- **Cruzamento de 2 pontos:** consiste em escolher dois pontos aleatórios para seccionar os cromossomos dos genitores, e logo após, recombiná-los intercalando as partes de um genitor com as de outro, formando novos cromossomos completos, como mostrado na Figura 4.

Figura 3 – Ilustração de um cruzamento de 1 ponto.



Figura 4 – Ilustração de um cruzamento de 2 pontos.



- **Cruzamento uniforme:** consiste em sortear, gene a gene, de qual dos dois genitores o filho herdará a característica, formando ao final do processo dois novos cromossomos completos, como mostrado na Figura 5.

Os cruzamentos garantem a evolução da população através da recombinação dos genes. Para garantir a diversidade existe a mutação, que é aplicada de forma menos frequente, a qual modifica aleatoriamente alguns genes do indivíduo, conforme veremos a seguir.

2.4.3.3 Mutação

Depois de realizado com sucesso o cruzamento, gerando novos indivíduos, entra em cena o operador de mutação. Com uma probabilidade baixa, parte dos genes do indivíduo são alteradas aleatoriamente. De acordo com Linden (2012), o conceito fundamental é

Figura 5 – Ilustração de um cruzamento uniforme.



que o valor da probabilidade deve ser baixo. Do contrário, o AG se portará mais como uma técnica chamada *random walk*, onde a solução é determinada de forma aleatória.

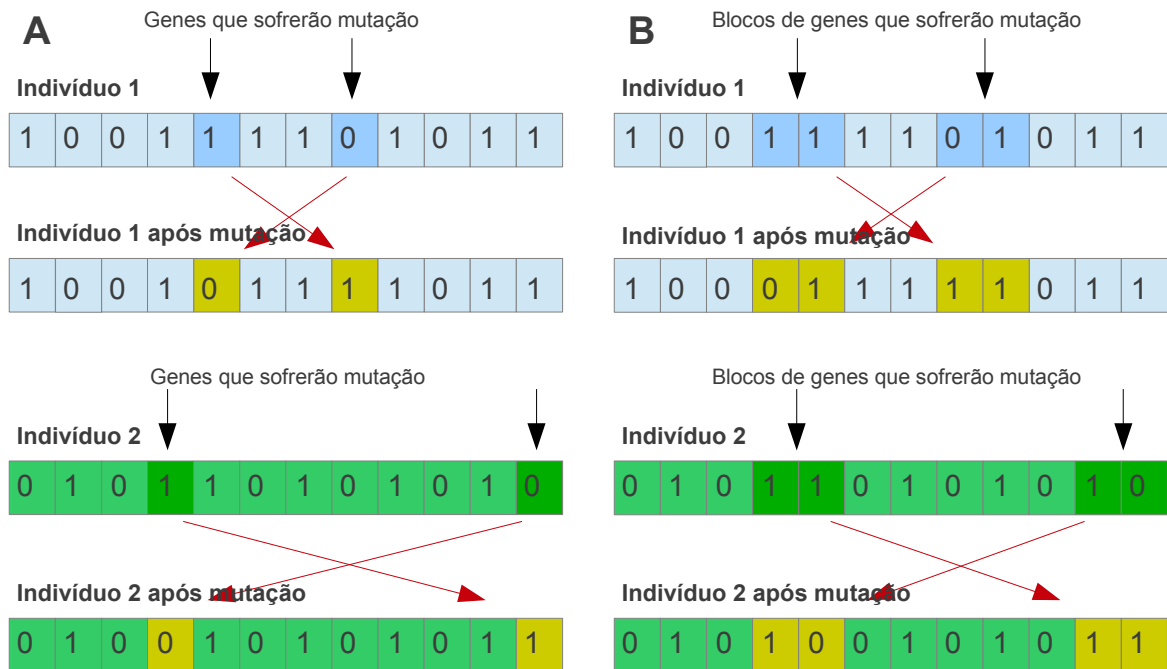
Segundo [Alba e Dorronsoro \(2008\)](#), o operador de mutação é o responsável por incluir na população um grau de diversidade considerável. Dessa forma, a mutação é uma importante ferramenta para evitar a convergência prematura.

A seguir são apresentadas três técnicas bem conhecidas na literatura:

- **Troca (*swap*):** Essa técnica consiste em selecionar aleatoriamente genes ou blocos de genes distintos e, trocá-los de posição. Na Figura 6 A, observamos a troca de genes isolados e, na Figura 6 B, a troca de blocos de genes.
- **Inversão:** Essa técnica consiste em selecionar aleatoriamente um bloco de genes e invertê-lo. Se o bloco escolhido fosse ABCD, após a mutação por inversão, o mesmo seria DCBA. Um exemplo deste tipo de mutação pode ser observado na Figura 7 A.
- **Inserção:** Essa técnica consiste em selecionar aleatoriamente um bloco de genes, removê-lo do cromossomo e logo após inseri-lo em outra posição no mesmo cromossomo, conforme pode ser observado na Figura 7 B.

De acordo com [Linden \(2012\)](#), os AG são excelentes candidatos para paralelização, já que trabalham com evoluções de grandes populações e normalmente requerem grande tempo de computação. Nesse sentido este trabalho propõe a paralelização de um AG. Para ajudar no entendimento do trabalho veremos a seguir, alguns conceitos de computação paralela.

Figura 6 – Mutação utilizando a técnica de Troca ou *swap*. Genes isolados em (A) e Blocos de genes em (B).

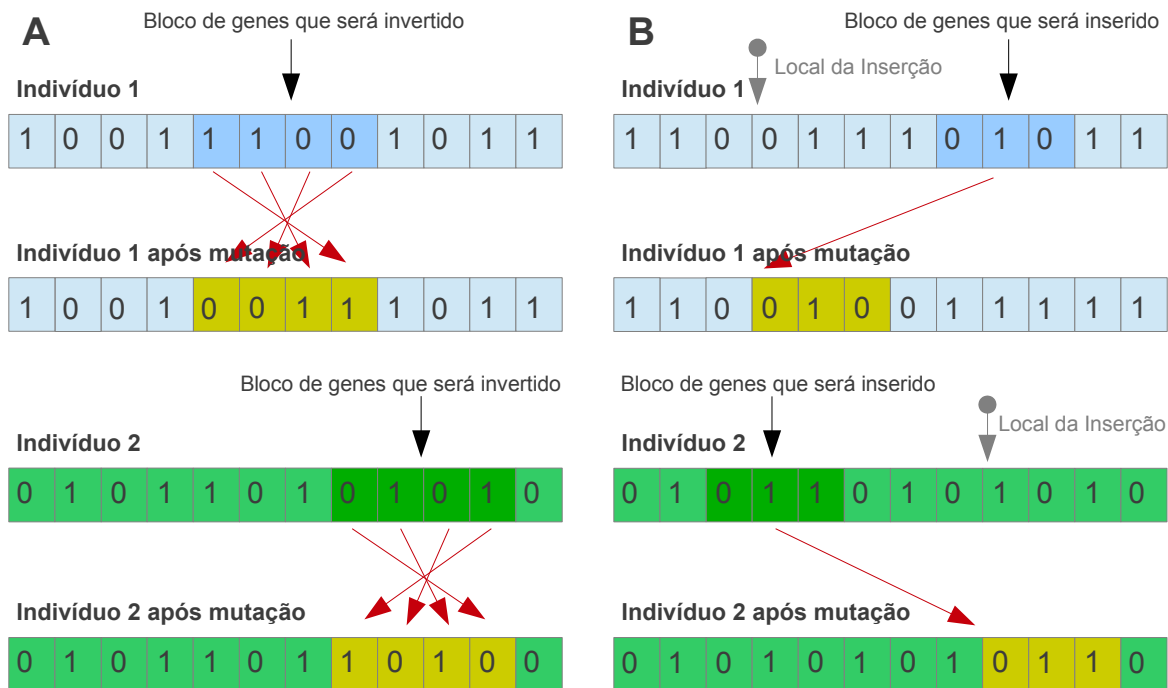


2.5 Computação paralela

Computação paralela é uma forma de computação que consiste em decompor problemas grandes em subproblemas menores que então são resolvidos simultaneamente. Essa ideia não é nova, mas recentemente entrou em evidência devido às limitações físicas para aumento da frequência dos processadores. De acordo com Rauber e Rüniger (2010), os principais fabricantes de *chips* passaram a fabricar processadores com várias unidades de processamento com baixo consumo de energia em um único *chip*. As unidades de processamento possuem controles independentes e podem acessar a mesma memória concorrentemente. Ainda de acordo com Rauber e Rüniger (2010), o termo *core* designa uma unidade de computação e o termo *multicore* designa um processador com várias unidades de computação.

Wilkinson e Allen (2005) nos dizem que existem basicamente dois tipos de computadores paralelos: multiprocessadores com memória compartilhada e multicomputadores com memória distribuída. Ainda, de acordo com Wilkinson e Allen (2005), programar para memória compartilhada, consiste em dividir as tarefas entre vários processadores que operam sobre dados armazenados em uma mesma memória, compartilhada entre todos. Programar para computação distribuída, consiste em dividir as tarefas e enviá-las, usando bibliotecas de passagem de mensagens, para computadores distintos, que computam as tarefas e seus resultados serão agrupados para compor a solução do problema.

Figura 7 – Mutação utilizando a técnica de Inversão (A) e Inserção (B).



Este trabalho usará os conceitos aplicados à memória compartilhada, já que os multiprocessadores com este tipo de memória estão amplamente popularizados.

Existem várias *API* (*Application Program Interface*) para programação paralela com memória compartilhada, entre elas POSIX Threads, Cilk++, Intel TBB e OpenMP. Esta última, é um padrão portátil para programação de sistemas de memória compartilhada aceito e muito difundido, desenvolvido por especialistas da indústria e que já está incorporado em muitos compiladores (WILKINSON; ALLEN, 2005; ZHU et al., 2009; RAUBER; RÜNGER, 2010).

A *API* OpenMP foi escolhida por proporcionar uma interface simples e ser muito flexível para desenvolvimento de programas paralelos.

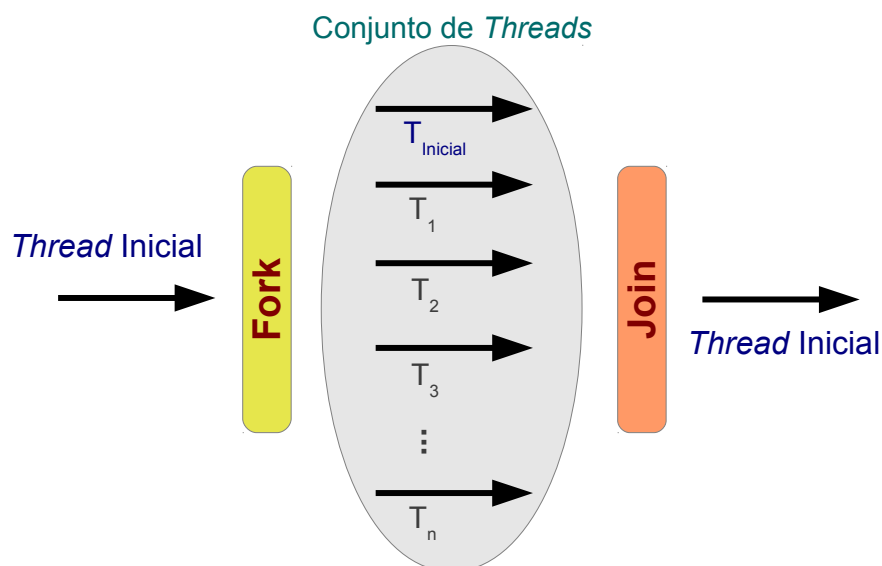
2.5.1 OpenMP

OpenMP provê diversas diretivas de compilação, biblioteca de rotinas e variáveis de ambiente. Pode ser usado para estender as linguagens de programação C, C++ e Fortran com construções do tipo *SPMD*, o que significa que o mesmo código é usado sobre dados diferentes. Este padrão foi projetado em 1997 e é mantido pela *OpenMP Architecture Review Board* (ARB) e, desde então, muitos desenvolvedores tem inserido esse padrão em seus compiladores (ZHU et al., 2009; RAUBER; RÜNGER, 2010).

Wilkinson e Allen (2005) e Zhu et al. (2009) dizem que o OpenMP segue o modelo

fork-join baseado em *threads*. Eles relatam que as regiões sequenciais do código são executadas por uma *thread* mestre e que as regiões paralelas podem ser executadas por um conjunto de *threads*. O número de *threads* nesse conjunto pode ser determinado por vários fatores. De acordo com [Zhu et al. \(2009\)](#) e [Rauber e Rüniger \(2010\)](#), a *thread* inicial executa o código sequencialmente até encontrar a primeira construção paralela. Neste momento, implicitamente, a *thread* inicial cria um conjunto de *threads* constituído de novas *threads* e a ela mesma que passa então a agir como mestre das novas *threads* criadas. O código dentro do construtor paralelo é chamado região paralela, por ser executado em paralelo pelas *threads* do conjunto. Conforme descrevem [Rauber e Rüniger \(2010\)](#) o modo de execução paralelo pode ser SPMD ou, então, diferentes *tasks* (tarefas) para diferentes *threads*. Ao final da região paralela existe uma barreira de sincronização, a partir da qual, somente a *thread* mestre prossegue. Este modelo pode ser visualizado na Figura 8.

Figura 8 – Representação do modelo *fork-join*.



Fonte: Adaptação de ([RAUBER; RÜNIGER, 2010](#)).

[Rauber e Rüniger \(2010\)](#) explicam que o OpenMP trabalha com dois modelos e memória diferentes: memória privada e memória compartilhada. Quando a memória é compartilhada, todas as *threads* tem acesso ao seu conteúdo. Em oposição, quando a memória é privada o acesso é limitado a uma única *thread*.

2.5.2 Diretivas OpenMP utilizadas neste trabalho

Para as linguagens C e C++ as diretivas OpenMP são inseridas no código fonte com a declaração `#pragma`. A forma geral de declaração de uma diretiva é a seguinte:

```
#pragma omp nome_diretiva [cláusulas [ ] ...]
```

As cláusulas são opcionais e diferentes para as diferentes diretivas. A diretiva atua sobre a próxima linha do código fonte, ou sobre o bloco delimitado por parênteses, declarado imediatamente após a diretiva (RAUBER; RÜNGER, 2010; WILKINSON; ALLEN, 2005).

2.5.2.1 Regiões Paralelas

Uma região paralela é um trecho de código que deve ser executado por um conjunto de *threads*. Este conjunto é criado imediatamente antes da região paralela e, é destruído assim que todas as *threads* chegam a barreira implícita que delimita o final da região paralela (WILKINSON; ALLEN, 2005; RAUBER; RÜNGER, 2010).

A declaração usa a seguinte sintaxe:

```
#pragma omp parallel [cláusulas [ ] ...]
{ //trecho de código... }
```

De acordo com Rauber e Rüniger (2010) o construtor paralelo cria um conjunto de *threads* mas não distribui o trabalho. Caso não exista uma outra diretiva distribuindo o trabalho, o trecho de código é então executado por todas as *threads*.

2.5.2.2 Sections

Sections é um tipo de construtor paralelo usado para distribuir tarefas não iterativas às *threads*. Dentro deste construtor, diferentes blocos de código são chamados de *section* e, são distribuídos a diferentes *threads*.

A declaração usa a seguinte sintaxe:

```
#pragma omp sections [cláusula [cláusula] ...]
{
    [#pragma omp section]
    { //trecho de código...}
    [#pragma omp section]
    { //trecho de código...}
    .
    .
    .
}
}
```


2.5.2.3 Laços Paralelos

Um laço paralelo é um laço em que as iterações são distribuídas entre as *threads* de uma região paralela (WILKINSON; ALLEN, 2005; RAUBER; RÜNGER, 2010).

A declaração usa a seguinte sintaxe:

```
#pragma omp for [cláusulas [ ] ...]
for(inicialização; controle; incremento){
    //trecho de código...
}
```

É possível combinar o construtor de regiões paralelas com o construtor de laços paralelos, usando a seguinte sintaxe:

```
#pragma omp parallel for [cláusulas [ ] ...]
{
    for(inicialização; controle; incremento){
        //trecho de código...
    }
}
```

2.5.2.4 Cláusulas

De acordo com Rauber e Rüniger (2010), várias cláusulas podem ser definidas na declaração de um construtor paralelo. Entre elas:

- `private(lista_de_variáveis)`: define que cada *thread* trabalhará com uma cópia privada das variáveis listadas.
- `shared(lista_de_variáveis)`: define que as *threads* compartilharão as variáveis listadas.
- `schedule(tipo[, bloco])`: define a maneira como as iterações de um laço paralelo serão divididas entre as *threads*. O tipo deve ser definido como `static`, `dynamic`, `guided`, `auto` ou `runtime`.

Static: A distribuição das iterações é feita de forma estática, em tempo de compilação. O `bloco` determina o tamanho da fatia de iterações que cada *thread* receberá.

Dynamic: A distribuição das iterações é feita de forma dinâmica, em tempo de execução. O `bloco` determina o tamanho da fatia de iterações que cada *thread* receberá.

Guided: A distribuição das iterações é feita de forma dinâmica, em tempo de execução. O `bloco` representa a fatia mínima que cada *thread* receberá. As fatias começam grandes, e vão diminuindo gradativamente até o valor definido no bloco.

Runtime: As decisões são feitas em tempo de execução, e são dependentes das especificações da implementação da biblioteca OpenMP.

Auto: As decisões sobre a distribuição das iterações são delegadas ao compilador.

Caso seja declarado um laço paralelo sem a cláusula `schedule`, valores padrão para `tipo` e tamanho de `bloco` serão assumidos. Esses valores podem variar conforme as especificações da implementação da biblioteca OpenMP (RAUBER; RÜNGER, 2010).

2.6 Trabalhos Relacionados

Com o objetivo de melhorar o desempenho dos Algoritmos Genéticos, a maioria dos autores recorre a hibridização dos mesmos. De acordo com Linden (2012), um Algoritmo Genético Híbrido (AGH), é um algoritmo que combina AG com outra técnica de Inteligência Artificial (IA) para obter em conjunto resultados melhores do que seria alcançados aplicando as técnicas isoladamente. 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.

Nazif e Lee (2012) propuseram um AG com uma técnica de cruzamento otimizado para solucionar o PRV. Enquanto a maioria dos AGs usam métodos estocásticos para realizar os cruzamentos, o cruzamento otimizado usa a função objetivo para auxiliar neste processo. O cruzamento otimizado gera dois descendentes, um otimizado e outro exploratório. O primeiro tenta obter a melhor função objetivo dos descendentes recém criados. O segundo visa manter a diversidade genética. O autor afirma que os melhores resultados são obtidos com uma probabilidade de cruzamento entre 0,6 e 0,95. A mutação é realizada com os operadores de troca e inversão e, os dois são aplicados aleatoriamente a cada descendente. O AG com cruzamento otimizado de Nazif e Lee (2012) usa elitismo e a cada 50 gerações elimina indivíduos repetidos da população. Isso é feito para evitar a convergência prematura e melhor explorar o espaço de busca. Este método é chamado a cada 50 evoluções porque o mesmo tem um custo elevado de computação. A população inicial é gerada randomicamente e, seu tamanho varia conforme o tamanho da entrada. O algoritmo para se em 100 gerações não houver melhora no custo.

Wang e Lu (2010) propuseram um AGH para solucionar o PRV com três estágios distintos. O primeiro estágio consiste em gerar uma população bem estruturada através

de três métodos: incorporação do vizinho mais próximo ao algoritmo de varredura, algoritmo de inserção melhorado e, criação aleatória. O segundo estágio consiste em realizar experimentos para otimizar as probabilidades de cruzamento e mutação no desempenho do AGH. O terceiro e último estágio consiste em um mecanismo que age além dos operadores de cruzamento e mutação. Trata-se de uma heurística que combina o algoritmo de inserção melhorado com inserção randômica de mutação. O AGH para quando em mil gerações não houver ganho ou ao atingir 10 mil gerações. O AGH de Wang e Lu (2010) usa ainda uma estratégia elitista que substitui indivíduos inferiores por superiores. De acordo com os autores, este AGH supera o AG puro e atinge resultados que são no máximo 5% acima da melhor solução conhecida.

Prins (2004) também propôs um AGH para o VRP, que usa a técnica de cruzamento chamada de *Order Crossover* (OX) para criar as novas populações e implementa uma busca local através da substituição do operador de mutação tradicional. Essa busca é realizada através de movimentos de substituição, troca e inserção no cromossomo. Neste AGH não são permitidas soluções idênticas na população, o que evita a convergência prematura da mesma e assegura uma maior dispersão das soluções. Para não ter que testar todas as soluções em busca de clones, o AGH considera que as soluções devem estar espaçadas por uma constante k . Se uma solução não respeitar esse espaçamento, ela é impedida de compor a população e é descartada. Para o cruzamento, são selecionados dois pares de genitores, onde em cada par, o indivíduo com pior função objetivo é descartado. O cruzamento gera dois novos indivíduos e, um deles é escolhido aleatoriamente para substituir um indivíduo com custo acima da média na população.

Baker e Ayechew (2003) propuseram um AGH simples e competitivo com outras técnicas no que tange a qualidade da solução e tempo de computação. A população inicial é gerada com o auxílio do ângulo polar do cliente. Para o cruzamento, são selecionados 2 indivíduos por torneio binário e, o melhor é usando como genitor, outros 2 indivíduos são selecionados por sorteio aleatório, e o melhor deles também é usando como genitor. O algoritmo incorpora uma busca local implementada através do método *2-optimal* e que é acionado a cada 10.000 gerações. O AG evolui por 20.000 gerações. De acordo com o autor, os melhores resultados foram obtidos com cruzamento de 2 pontos, sorteados randomicamente.

Todos os trabalhos apresentados chegaram a soluções muito próximas da melhor solução conhecida e, para algumas instâncias, chegaram na melhor solução conhecida. No que tange ao tempo de computação necessário para chegar a esse resultado, os autores não apresentam esse dado. Adicionalmente, embora AGs sejam paralelizáveis, esse opção é pouco explorada.

Conforme aumenta a entrada do AG, aumenta o tempo de computação necessário para encontrar uma solução adequada. Muitos esforços para acelerar AG tem sido dis-

pendidos e a paralelização é tida como uma escolha promissora (YUSSOF; RAZALI; SEE, 2009).

O paralelismo intrínseco dos AG embora seja citado por muitos autores, é pouco explorado nos trabalhos relacionados. Autores como Yussuf, Razali e See (2009) e Zhu-rong et al. (2011) fizeram uso deste paralelismo e, propuseram Algoritmo Genético Paralelo (AGP) com intuito de melhorar o tempo de computação e acelerar a convergência da população.

Neste trabalho, exploraremos o paralelismo inerente aos AG para realizar tarefas sem dependências simultaneamente. O objetivo é implementar um AG capaz de computar mais gerações em um período de tempo menor, acelerando o tempo de resposta. O AGP deve ser capaz de obter uma solução de melhor qualidade quando executado pelo mesmo tempo que a sua versão sequencial. No próximo capítulo são apresentados os passos para construção do algoritmo e, os testes que delinearam a escolha dos parâmetros do mesmo.

2.7 Considerações sobre a Revisão Bibliográfica

Neste capítulo, foi apresentado o PRV, que é o problema de otimização combinatória alvo deste trabalho. Verificamos que é inviável realizar uma busca exaustiva pois este problema é intratável. Logo, faz-se necessário o uso de uma meta-heurística para obter em um tempo razoável uma solução aceitável para o problema.

As meta-heurísticas são uma forma de encontrar uma solução para um problema, abstraindo boa parte da teoria matemática que o envolve. Foram apresentadas as três meta-heurísticas mais populares: *Simulated Annealing*, *Tabu Search*, e *Genetic Algorithm*.

O foco deste trabalho está voltado para implementação de um AG para resolver o PRV. Para tanto, foi apresentado o funcionamento básico de um AG, onde foi possível verificar que as possíveis soluções são codificadas em cromossomos e, agrupadas em populações. Essas populações sofrerão ações dos operadores genéticos que conduzirão as evoluções até que a condição de parada seja atingida. Vimos o funcionamento dos operadores genéticos, começando pelo de seleção, responsável por selecionar os indivíduos para o cruzamento, seguido do operador de cruzamento, responsável por combinar a carga genética dos indivíduos selecionados, formando novos indivíduos que herdaram características de ambos os genitores. Vimos ainda o funcionamento do operador genético de mutação, responsável por alterar, com certa probabilidade, parte dos genes dos novos indivíduos criados. Durante todo o processo de evolução, para distinguir os indivíduos mais aptos dos menos aptos e, atribuir-lhes um custo, é usada a função objetivo, que é a única ligação entre o AG e o problema que está sendo resolvido.

Foram apresentados também os conceitos de computação paralela com memória

compartilhada e distribuída. Como os multiprocessadores com memória compartilhada estão amplamente difundidos, foi escolhida a [API OpenMP](#) para a paralelização do código do [AG](#), visando melhor aproveitar este tipo de arquitetura. Também foram apresentadas as diretivas e cláusulas que serão utilizadas para a paralelização do [AG](#).

Nos trabalhos relacionados, ficou evidenciado que os autores tem utilizado [AG](#) híbridos, que combinam outras técnicas heurísticas para melhorar a qualidade da solução.

O próximo capítulo apresenta a construção do [AG](#) sequencial puro. Também apresenta testes com parâmetros como tamanho da população, número de evoluções, técnica de cruzamento e mutação.

3 Implementação do Algoritmo Genético

3.1 Introdução

Como observado no capítulo anterior, é inviável fazer uma busca exaustiva para calcular o custo de todas as rotas para descobrir a menor em todo espaço de busca (mínimo global). Visando encontrar uma solução em tempo razoável, os **AG** são uma excelente alternativa, pois mesmo que não garantam que a melhor solução seja encontrada, conduzem a uma solução factível num período de tempo consideravelmente menor se comparado a uma busca exaustiva.

Este capítulo tem por objetivo demonstrar como foi implementado o **AG**. Ele está estruturado da seguinte maneira: a Seção 2 apresenta a definição das características do **PRV** que será utilizado para testar a implementação do **AG**, a Seção 3 apresenta uma visão geral do algoritmo implementado e como foram definidos os parâmetros do mesmo, a Seção 4 apresenta o tempo de computação do **AG** implementado e a Seção 5 apresenta as considerações sobre o capítulo.

3.2 Definição das Características do PRV

Para validação da implementação do **AG** foi usado parte do *Benchmark* de **Christofides et al. (1979)**, que reúne 14 instâncias do **PRV** com o número de clientes (cidades) variando entre 50 e 199. Metade das instâncias possuem restrição quanto a capacidade de transporte do veículo e a outra metade, além de restrição de capacidade de transporte, também possui restrições quanto a distância percorrida por rota. Este *Benchmark* é largamente utilizado (**KHEIRKHAZHADDEH; BARFOROUSH, 2009; JOZEFOWIEZ; SEMET; TALBI, 2009; LIN et al., 2009; PRINS, 2004; LAPORTE et al., 2000**).

Neste trabalho, são utilizadas três instâncias com restrição da capacidade de transporte, que estão disponíveis para *download* na *internet*¹. Foram escolhidas as instâncias *c50*, *c100* e *c120*, respectivamente com 50, 100 e 120 cidades. A escolha levou em consideração o tempo de computação que o algoritmo implementado consome para processá-las.

Cada instância é um arquivo texto contendo as seguintes informações:

número de cidades, menor custo (em Km) conhecido

capacidade de transporte do veículo

¹Disponível em <http://mistic.heig-vd.ch/taillard/problemes.dir/vrp.dir/vrp.html> (último acesso em janeiro de 2013).

coordenadas do depósito
para cada cidade: número, coordenadas, demanda

A Tabela 2 apresenta os menores custos (em Km) conhecidos para as instâncias selecionadas.

Tabela 2 – Menores custos (em Km) conhecidos para as instâncias usadas neste trabalho.

<i>Instância</i>	<i>Menor Custo (em Km)</i>
c50	524,61
c100	826,14
c120	1042,11

3.3 Algoritmo Genético Implementado

O AG implementado toma por base o AG básico visto no capítulo anterior. Foram feitas pequenas modificações que podem ser conferidas no Algoritmo 2. A seguir daremos uma visão geral do mesmo.

3.3.1 Visão Geral

Conforme podemos observar no Algoritmo 2, primeiramente é gerada uma população inicial (linha 2). A população é composta por vários indivíduos, sendo que cada indivíduo é uma possível solução para o PRV. Para gerar um indivíduo, uma função escolhe qual a primeira cidade a ser visitada aleatoriamente. A partir deste ponto, busca-se sempre a cidade mais próxima até que todas as cidades, sem repetição, tenham sido visitadas. Este 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. 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). Uma nova geração começa a ser construída pela perpetuação de um percentual dos indivíduos mais aptos da geração atual (linha 5). Até preencher a nova geração com mesmo número de indivíduos da geração anterior, indivíduos da população anterior são sorteados para o cruzamento. Metade dos cruzamentos são realizados entre um dos melhores indivíduos e um indivíduo sorteado randomicamente (linha 10). A outra metade dos cruzamentos são realizados entre dois indivíduos distintos selecionados randomicamente (linha 17). Esta escolha aleatória visa a manutenção da diversidade genética, de forma que a população não venha a convergir rapidamente, aumentando assim as chances de produzir indivíduos mais aptos. Cada cruzamento retorna dois descendentes.

Algoritmo 2 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
```

tes. Com certa probabilidade, cada indivíduo retornado sofre mutação em parte de seus genes (linhas 12 e 19). Os descendentes então são avaliados e adicionados à nova geração (linhas 14 e 21). Quando a nova geração estiver completa, ela substitui a população anterior. Isso se repete até que o número de evoluções pré-determinado seja atingido e então o algoritmo retorne o melhor indivíduo encontrado.

Como pode-se perceber na explicação do AG acima, sua implementação depende da definição de uma série de parâmetros do algoritmo. Nas subseções a seguir, explicaremos o impacto desses parâmetros na qualidade das soluções, quais valores foram definidos para eles em nossa implementação e, como chegou-se a tais valores. Sempre que foi necessário executar testes, foram coletados os resultados de 30 execuções para cada configuração.

3.3.2 Tamanho da População e Número de Evoluções

O tamanho da população é um escalar que representa o número de indivíduos que compõem uma população, enquanto que o número de evoluções é um escalar que repre-

Tabela 3 – Impacto do aumento da população e número de evoluções na qualidade da solução.

<i>Entrada</i>	<i>População</i>	<i>Evoluções</i>	<i>Solução Média (Km)</i>
c50	N	$N \times N$	586,17
		$N \times N \times 10$	565,81
	$N \times 10$	$N \times N$	559,30
		$N \times N \times 10$	544,07
c100	N	$N \times N$	1029,00
		$N \times N \times 10$	983,56
	$N \times 10$	$N \times N$	997,08
		$N \times N \times 10$	940,62

senta o número de vezes que será criada uma nova geração. Estes parâmetros influenciam diretamente na qualidade da solução e no tempo de computação. Uma população muito pequena pode não fornecer diversidade genética suficiente para conduzir o AG a boas soluções. Por outro lado, se a população for grande demais, podemos estar nos aproximando de uma busca exaustiva (LINDEN, 2012).

No Algoritmo 2, o tamanho da população é usado na criação da população inicial (linha 2), para definir a porcentagem de indivíduos que serão perpetuados (linha 5) e também para o controle do laço que completa a nova geração com cruzamentos (linhas 6 a 23). Já o número de evoluções é o número de vezes que será repetido o laço que controla as evoluções (linhas 4 a 27).

Neste trabalho, consideramos dois tamanhos de população: N e $N \times 10$. Também consideramos dois números de evolução: $N \times N$ e $N \times N \times 10$. Para ambos os casos, N é o número de cidades na entrada, e 10 é um escalar que representa 10% de N . Esses valores foram escolhidos com o intuito de manter a proporção conforme aumenta-se a entrada.

A Tabela 3, mostra o impacto do tamanho da população e do número de evoluções na qualidade das soluções encontradas. Nela percebe-se que quanto maior o valor desses parâmetros, menor é o custo (em Km) obtido pelo AG.

3.3.3 Taxa de Cruzamento e Probabilidade de Mutação

Taxa de cruzamento é a quantidade de indivíduos da nova geração que serão formados a partir de cruzamentos. Probabilidade de mutação, é a probabilidade de um indivíduo recém gerado por um cruzamento sofrer mutação em alguns de seus genes. No Algoritmo 2, a taxa de cruzamento é utilizada para determinar quantas vezes o laço dos cruzamentos será executado (linha 6 a 23). A probabilidade de mutação é utilizada nos testes das linhas 11 e 18.

Autores como Michalewicz e Fogel (2004) e Linden (2012) dizem que a taxa de cruzamento deve variar entre 60 e 95% e que é comum que se considere probabilidade

de mutação como: 100% - taxa de cruzamento. Os mesmos autores afirmam ainda que isso não é uma regra e, que se pode adotar estratégias em que os dois operadores sejam independentes.

Neste trabalho foram testadas duas possibilidades de taxa de cruzamento. A primeira versão, chamada de Sequencial 1, considera uma taxa de cruzamento de 100% - 1 indivíduo. Essa versão perpetua somente o melhor indivíduo da população anterior. A segunda versão, chamada de Sequencial 2, considera uma taxa de cruzamento de 80% (aproximadamente a média dos valores sugeridos por Michalewicz e Fogel (2004)). Essa versão preserva 20% dos indivíduos da população anterior, sendo que metade desses indivíduos são aleatórios e a outra metade são os melhores indivíduos. A identificação dos melhores indivíduos garante que cada um deles tem características distintas, ou seja, não há indivíduos repetidos entre os melhores. Com essa medida elitista, a nova população terá, na pior das hipóteses, os mesmos melhores indivíduos que a população anterior.

Para as duas versões a probabilidade de mutação considerada foi de 20% (100% - aproximadamente a média dos valores sugeridos por Michalewicz e Fogel (2004) para taxa de cruzamento). Os testes foram executados submetendo as instâncias c50 e c100 às versões Sequencial 1 e Sequencial 2 do programa, com diferentes tamanhos de população e número de evoluções. Os resultados podem ser vistos nas Figuras 9 e 10, onde têm-se os custos em Km obtidos em cada versão. Para cada configuração de parâmetros, temos valores dentro de um intervalo onde: o melhor custo (parte inferior do intervalo) é a menor distância em Km encontrada, o pior custo (parte superior do intervalo) é a maior distância em Km encontrada e, o custo médio (figura geométrica dentro do intervalo) é média de todas as distâncias em Km.

Conforme podemos observar nas Figuras 9 e 10, a versão Sequencial 2 (intervalo com losango laranja) encontrou melhores custos médios que a versão Sequencial 1 (intervalo com quadrado azul) em todos os casos de teste. Na Figura 9 percebemos que a melhor solução encontrada foi muito próxima em todos os testes, mas o valor médio das soluções encontradas foi ligeiramente melhor na versão Sequencial 2. Na Figura 10 percebemos que a diferença entre as versões Sequencial 1 e Sequencial 2 fica mais nítida. Melhores custos médios são obtidos pela versão Sequencial 2 em todos os testes. Esta versão também encontrou a maioria dos menores custos obtidos no teste.

Com base nos resultados apresentados, fixamos a taxa de cruzamento em 80% . A probabilidade de mutação foi definida em 20% levando em consideração o que foi sugerido por Michalewicz e Fogel (2004) e Linden (2012).

Figura 9 – Instância c50 submetida a versão Sequencial 1 e a versão Sequencial 2: médias, melhores e piores custos (em Km) encontrados, variando o tamanho da população e número de evoluções.

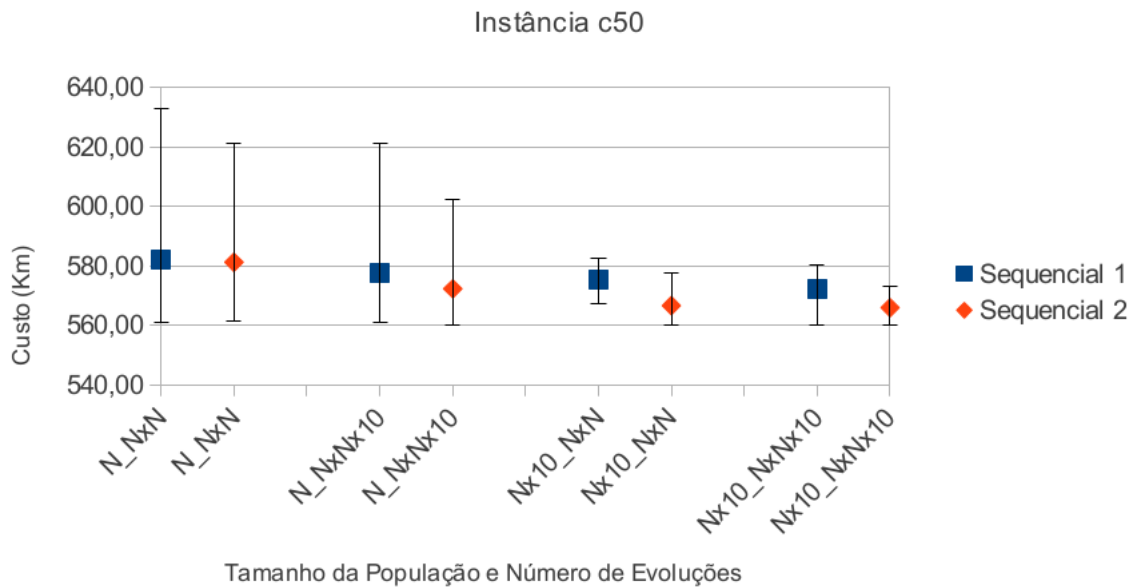
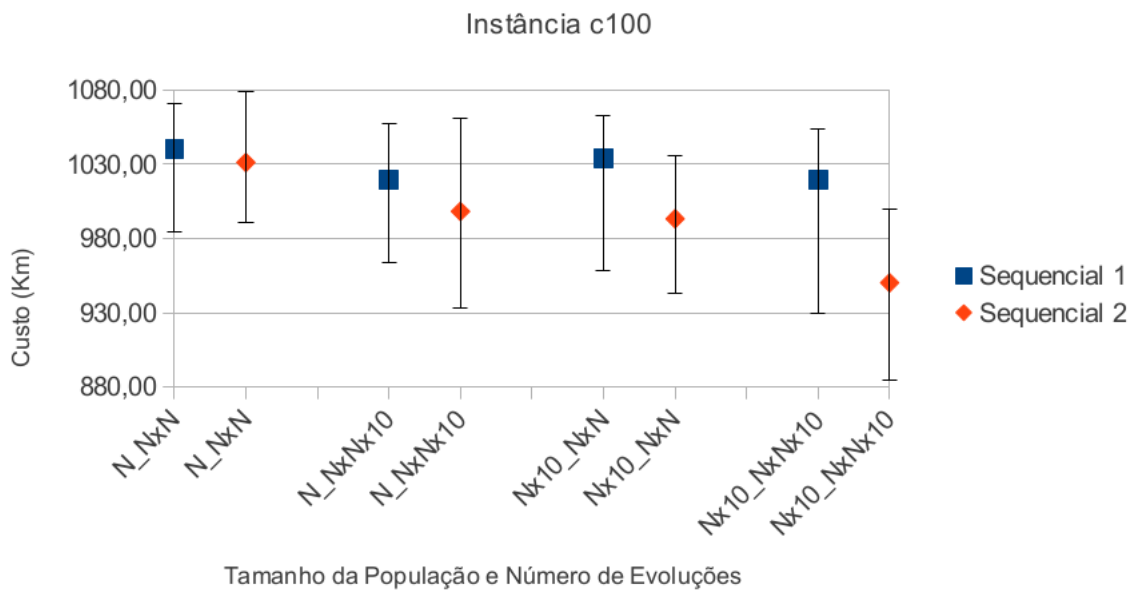


Figura 10 – Instância c100 submetida a versão Sequencial 1 e a versão Sequencial 2: médias, melhores e piores custos (em Km) encontrados, variando o tamanho da população e número de evoluções.



3.3.4 Taxa de Mutação

A taxa de mutação é a quantidade de genes do cromossomo que sofrerão alguma alteração. No Algoritmo 2, a taxa de mutação refere-se às seleções realizadas nas linhas 12 e 19. Autores como Wang e Lu (2009) afirmam que a taxa de mutação e de cruzamento afetam significativamente a capacidade de resolução do AG.

De acordo com Rayward-Smith et al. (1996), Michalewicz e Fogel (2004) e Linden

(2012), por razões históricas, quando estamos trabalhando com cromossomos binários, a taxa de mutação geralmente é algo 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 que uma taxa de mutação fixa, já que com o avanço das evoluções as novas populações tentem a estagnação. Neste sentido, uma taxa de mutação maior nas últimas evoluções das populações, compensaria essa tendência.

Para definição da taxa de mutação, foram testadas 3 possibilidades: mutação em 4%, 10% e variando de 4% a 10% dos cromossomos. Esta última começa com 4% e vai aumentando gradativamente até chegar a 10%, antes que se esgote o número total de evoluções.

As Figuras 11 e 12 apresentam as médias, os melhores e os piores custos na execução das instâncias c50 e c100, para as 3 possibilidades de taxas de mutação, variando-se o tamanho da população e número de evoluções. Os piores custos médios foram encontrados quando a taxa de mutação é fixa com valor 10% (representado nas Figuras 11 e 12 pelo intervalo com triângulo amarelo). Quando a taxa de mutação é fixa, com valor 4% (intervalo com losango laranja), os custos médios são melhores que os obtidos com taxa 10%. Contudo, no geral, os melhores custos foram encontrados com taxa de mutação variável, de 4% a 10% (intervalo com quadrado azul). Esse comportamento foi descrito nos trabalhos de Rayward-Smith et al. (1996), Michalewicz e Fogel (2004) e Linden (2012).

Com esse resultado, fica definido que a taxa de mutação usada nos próximos testes será variável, de 4 a 10%. O próximo passo será definir a técnica de mutação que será adotada.

3.3.5 Técnica de Mutação

Este trabalho analisou cinco técnicas de mutação, baseadas nas três técnicas básicas apresentadas no capítulo anterior: troca, inversão e inserção. São elas a mutação por troca de genes, por troca de bloco de genes, por inserção de genes, inversão de genes e ainda uma randômica, que sorteia qual das 4 técnicas anteriores será utilizada. Para simplificação, essas técnicas de mutação são chamadas de troca, troca bloco, inversão, inserção e randômica.

Para definir qual a melhor técnica de mutação, as instâncias c50 e c100 foram submetidas a cada uma das cinco técnicas de mutação acima citadas, variando-se o tamanho da população e número de evoluções. Os custos obtidos nos testes podem ser conferidos nas Figuras 13 e 14.

Na Figura 13 percebemos que para instância c50, os custos médios da mutação randômica (intervalo com quadrado azul) são ligeiramente menores quando o número de evoluções é menor. Quando o número de evoluções é maior, a técnica de mutação com

Figura 11 – Instância c50 submetida a versões com diferentes taxas de mutação: médias, melhores e piores custos (em Km) encontrados, variando o tamanho da população e número de evoluções.

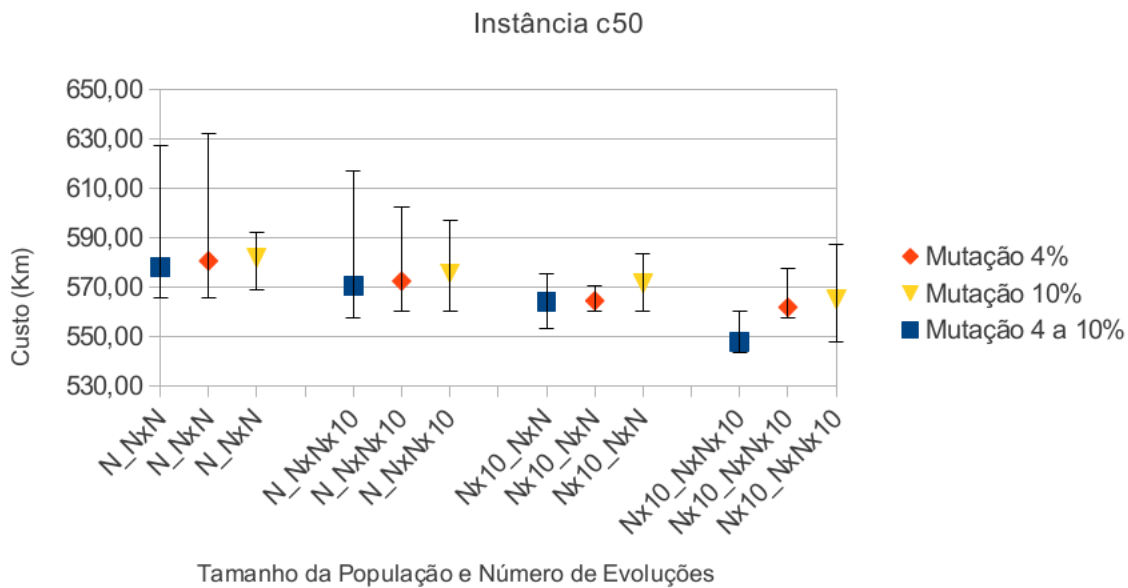
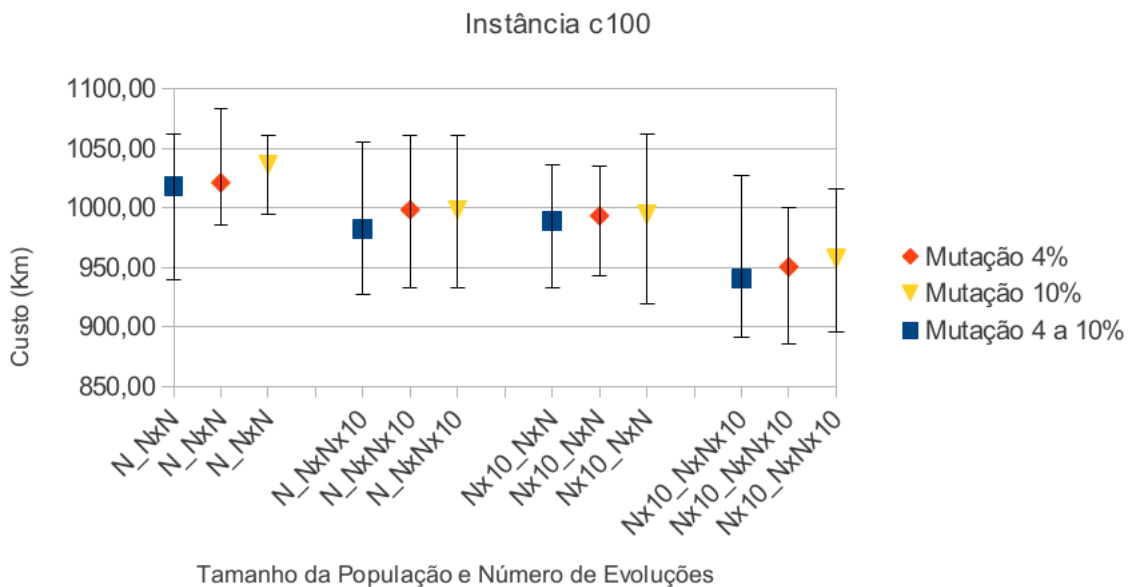


Figura 12 – Instância c100 submetida a versões com diferentes taxas de mutação: médias, melhores e piores custos (em Km) encontrados, variando o tamanho da população e número de evoluções.



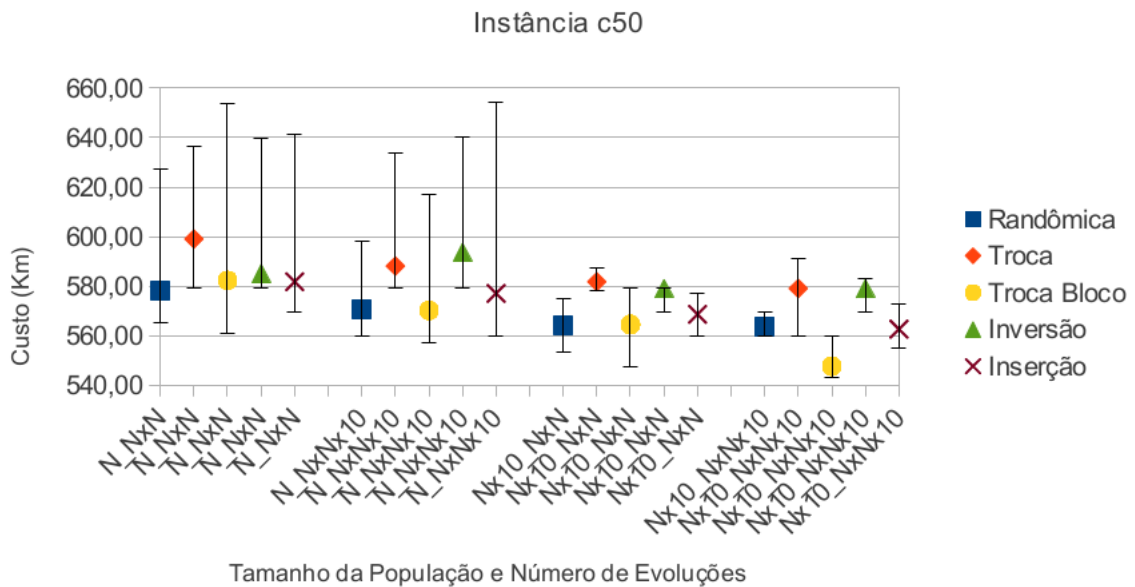
custos médios menores passa a ser a troca bloco (intervalo com círculo amarelo). Em todos os casos de teste para a instância c50, o menor custo de cada configuração foi encontrado utilizando a mutação troca bloco, o que pode ser conferido na Tabela 4.

Na Figura 14 percebemos que para a instância c100, a mutação randômica (intervalo com quadrado azul) obteve menores custos médios para os menores tamanhos de população e número de evoluções. Quando a população e/ou número de evoluções eram

Tabela 4 – Técnica de mutação que conduziu as instâncias c50 e c100 a menores custos médios e a menores custos, no teste para definir qual a melhor técnica de mutação.

<i>Instância</i>	<i>População</i>	<i>Evoluções</i>	<i>Custo médio</i>	<i>Menor custo</i>
c50	N	$N \times N$	Randômica	Troca Bloco
		$N \times N \times 10$	Troca Bloco	Troca Bloco
	$N \times 10$	$N \times N$	Randômica	Troca Bloco
		$N \times N \times 10$	Troca Bloco	Troca Bloco
c100	N	$N \times N$	Randômica	Randômica
		$N \times N \times 10$	Troca Bloco	Troca Bloco
	$N \times 10$	$N \times N$	Troca Bloco	Troca Bloco
		$N \times N \times 10$	Troca Bloco	Troca Bloco

Figura 13 – Instância c50 submetida a versões com diferentes técnicas de mutação: médias, melhores e piores custos (em Km) encontrados, variando o tamanho da população e número de evoluções.



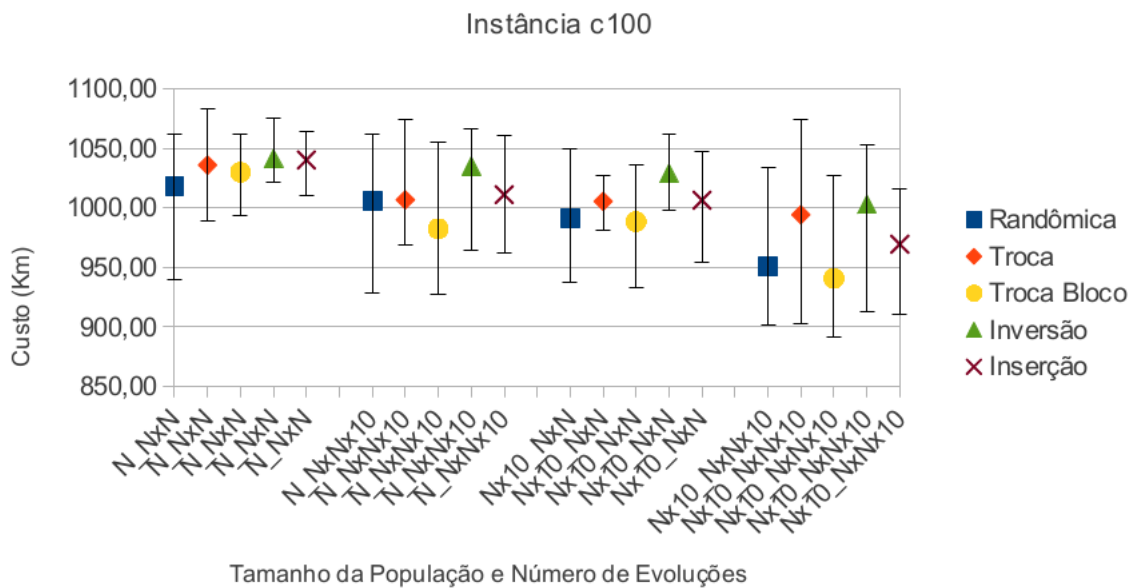
maiores, a técnica de mutação que obteve menores custos foi a troca bloco (intervalo com círculo amarelo). O menor custo de cada versão para a instância c100 teve o mesmo comportamento do custo médio, conforme observamos na Tabela 4.

Consideramos como melhor técnica de mutação aquela que mais vezes encontrou menores custos médios e a que encontrou as melhores soluções (menores custos). Observando a Tabela 4 percebemos que a técnica de mutação mais eficiente foi a troca bloco. O próximo passo será a escolha da melhor técnica de cruzamento.

3.3.6 Técnica de Cruzamento

Para definirmos qual a melhor técnica de cruzamento, foram feitos testes com as três técnicas básicas de cruzamento descritas no capítulo anterior (uniforme, de 1

Figura 14 – Instância c100 submetida a versões com diferentes técnicas de mutação: médias, melhores e piores custos (em Km) encontrados, variando o tamanho da população e número de evoluções.



ponto e de 2 pontos) e 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.

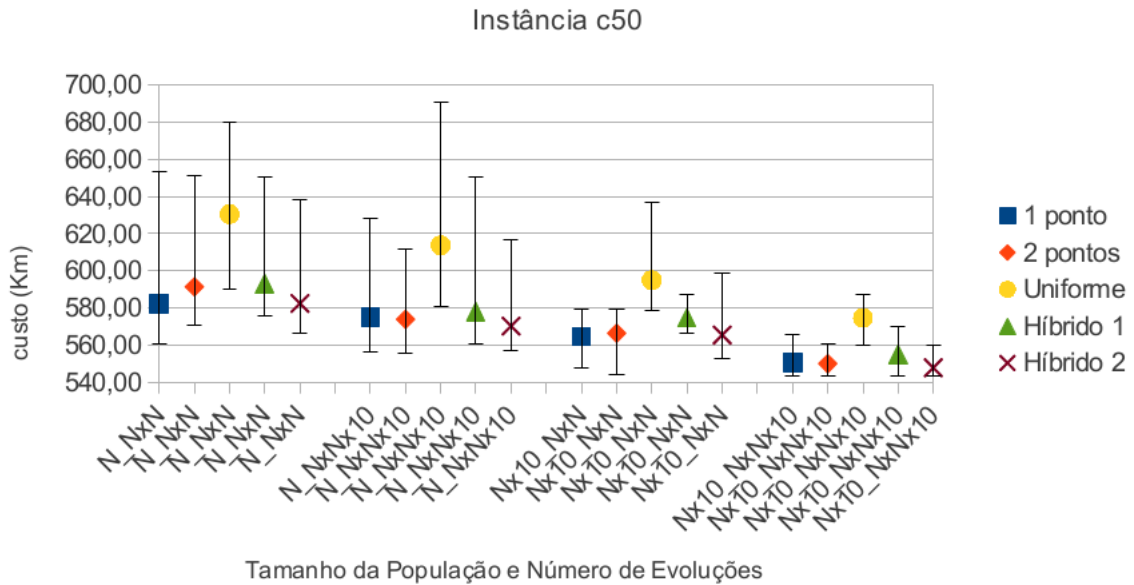
A descrição das técnicas híbridas é dada a seguir:

Híbrida 1: consiste em selecionar randomicamente, no momento do cruzamento, qual das técnicas básicas (cruzamento uniforme, de 1 ponto ou de 2 pontos) será usada para os indivíduos selecionados. Assim, pode-se chegar a melhores soluções pela diversidade genética decorrente do uso aleatório das técnicas básicas de cruzamento.

Híbrida 2: analisa a cada evolução o custo da melhor solução e, caso não haja melhoria em $N/5$ rodadas, troca a técnica de cruzamento usando uma fila circular com as técnicas básicas. Onde N é o número de cidades na entrada. Assim pode-se chegar a melhores soluções alternando a técnica de cruzamento sempre que houver estagnação na qualidade da solução encontrada.

As Figuras 15 e 16 apresentam as médias, melhores e piores custos obtidos para

Figura 15 – Instância c50 submetida a versões com diferentes técnicas de cruzamento: médias, melhores e piores custos (em Km) encontrados, variando o tamanho da população e o número de evoluções.



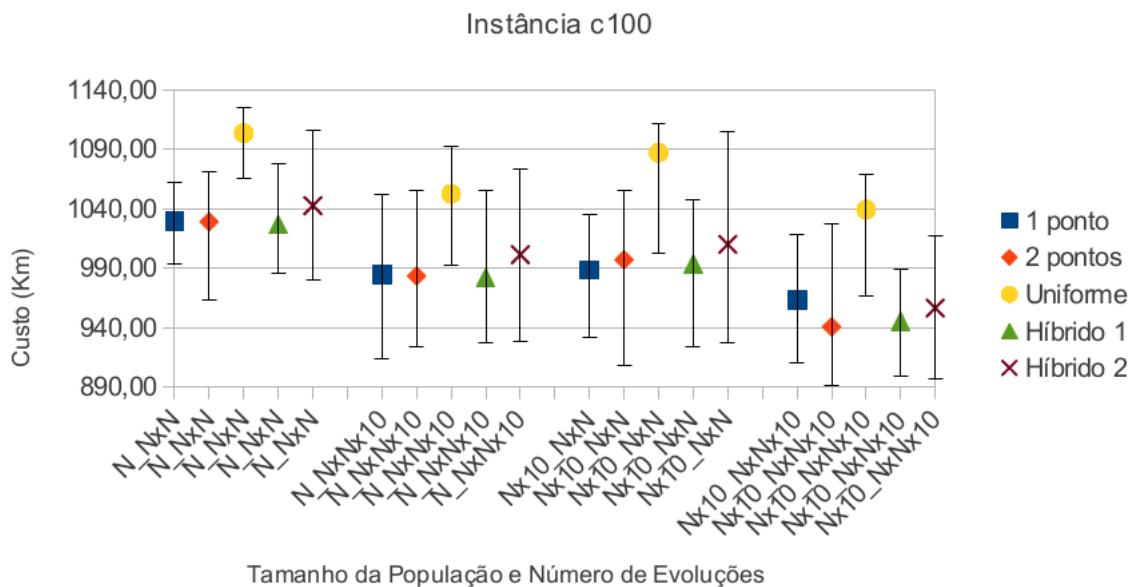
cada técnica de cruzamento, variando-se o tamanho da população e o número de evoluções. A Tabela 5 apresenta os menores custos médios e qual a técnica que os obteve. Conforme observamos na Figura 15, para a instância c50 as melhores técnicas de cruzamento foram o cruzamento de 1 ponto (intervalo com quadrado azul) e o cruzamento híbrido 2 (intervalo com xis marrom). Para a instância c100, de acordo com a Figura 16, as melhores técnicas de cruzamento foram o cruzamento de 1 ponto (intervalo com quadrado azul) e o cruzamento de 2 pontos (intervalo com losango laranja).

Tabela 5 – Técnica de cruzamento que conduziu as instâncias c50 e c100 a menores custos médios no teste de definição da melhor técnica.

Instância	População	Evoluções	Custo médio	Técnica de Cruzamento
c50	N	$N \times N$	582,42	1 ponto
		$N \times N \times 10$	570,23	híbrido 2
	$N \times 10$	$N \times N$	564,12	1 ponto
		$N \times N \times 10$	547,79	híbrido 2
c100	N	$N \times N$	1029,01	2 pontos
		$N \times N \times 10$	984,47	1 ponto
	$N \times 10$	$N \times N$	988,45	1 ponto
		$N \times N \times 10$	940,63	2 pontos

A definição da melhor técnica de cruzamento depende da instância utilizada e também de parâmetros como o tamanho da população e número de evoluções. Conforme observamos na Tabela 5 e nas Figuras 15 e 16, o aumento do número de indivíduos na população e o aumento do número de evoluções implicam em uma melhora significativa na qualidade da solução.

Figura 16 – Instância c50 submetida a versões com diferentes técnicas de cruzamento: médias, melhores e piores custos (em Km) encontrados, variando o tamanho da população e o número de evoluções.



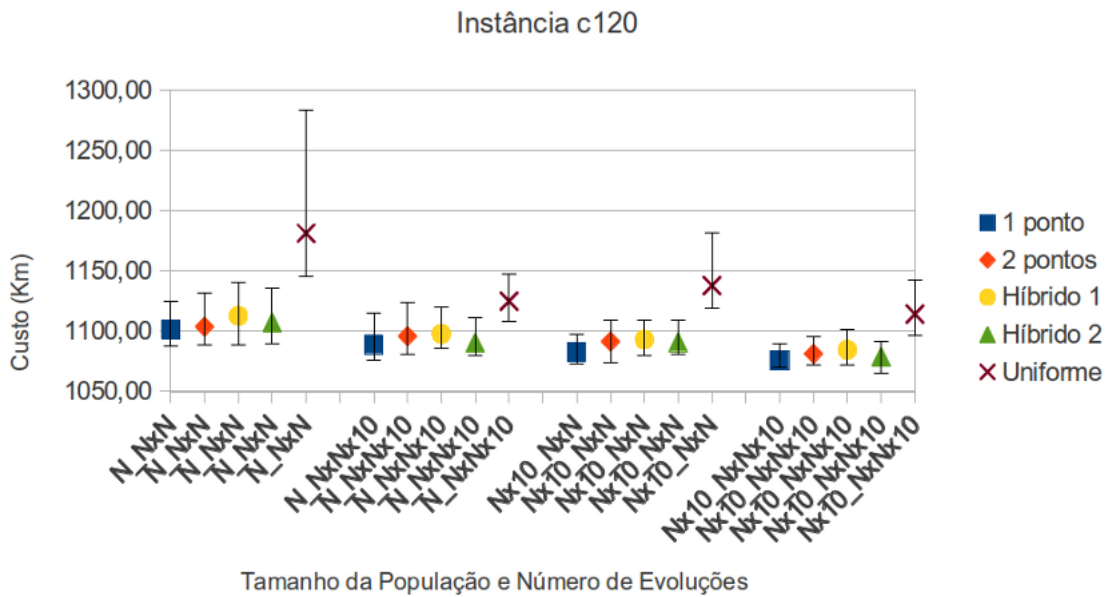
Para a instância c50, a técnica que conduziu ao menor custo médio foi a técnica de cruzamento híbrido 2, quando associada a uma população de tamanho $N \times 10$ e a um número de evoluções igual a $N \times N \times 10$. Com a instância c100, a técnica que conduziu ao menor custo médio foi a técnica de cruzamento de 2 pontos, quando associada a uma população de tamanho $N \times 10$ e a um número de evoluções igual a $N \times N \times 10$. Essas duas técnicas serão adotadas como melhor técnica de cruzamento para suas respectivas instâncias em nossos próximos testes.

3.3.7 Parâmetros para Instância c120

Com base nos resultados apresentados até agora, pode-se perceber que parâmetros como taxa de cruzamento, probabilidade de cruzamento, taxa de mutação, tamanho da população e número de evoluções, tiveram um comportamento parecido, independente da instância alvo. Por isso, os valores fixados para as instâncias c50 e c100 serão também adotados para a instância c120, uma vez que os testes para essa instância consumiram um grande tempo de processamento. O parâmetro que apresentou comportamento dependente da instância foi a técnica de cruzamento. Por este motivo testou-se as cinco técnicas de cruzamento para a referida instância.

A Figura 17 apresenta as médias, melhores e piores custos obtidos para cada técnica de cruzamento, variando-se o tamanho da população e o número de evoluções. Os custos médios obtidos foram muito parecidos e, para facilitar a análise, foram transcritos para a Tabela 6, que mostra o custo médio e a técnica de cruzamento que levou a este resultado para a instância c120.

Figura 17 – Instância c120 submetida a versões com diferentes técnicas de cruzamento: médias, melhores e piores custos (em Km) encontrados, variando o tamanho da população e o número de evoluções.



Na Tabela 6 verificamos que para todas as combinações de tamanho de população e número de evoluções, a técnica de cruzamento de 1 ponto conduziu a menores custos médios. Dessa forma, consideramos que a melhor técnica de cruzamento para a instância c120 é o cruzamento de 1 ponto.

Na próxima seção serão apresentados os tempos médios de computação para as instâncias alvo.

3.4 Tempo de Computação

Durante todo o processo de definição de parâmetros, o tempo de computação não foi apresentado. As definições foram feitas levando em conta sempre a qualidade das soluções. A Tabela 7 apresenta o tempo de computação para as instâncias c50, c100 e c120, considerando todos os parâmetros definidos nesse capítulo.

Durante nossa investigação o tamanho de população $N \times 10$ e o número de evoluções $N \times N \times 10$ sempre conduziram o AG implementado aos melhores resultados dos testes. Em contrapartida, conforme pode ser visto na Tabela 7, essa combinação é a que resulta num maior tempo de computação.

Existem duas possibilidades para reduzirmos o tempo total de execução: usar um processador melhor, ou usar melhor os recursos que os processadores atuais nos oferecem. A primeira possibilidade, embora seja factível, não será considerada, porque a infraestrutura de testes que temos a disposição é um multiprocessador Intel® Core 2 Quad Q8300 @ 2.50GHz, que possui 4 núcleos físicos. Este processador executando um algo-

Tabela 6 – Custo médio e a técnica de cruzamento empregada para a instância c120, variando-se o tamanho da população e o número de evoluções.

<i>População</i>	<i>Evoluções</i>	<i>Solução média</i>	<i>Cruzamento</i>
N	$N \times N$	1101,14	1 ponto
		1103,72	2 pontos
		1107,06	híbrido 2
		1112,73	híbrido 1
		1181,31	uniforme
N	$N \times N \times 10$	1089,02	1 ponto
		1090,28	híbrido 2
		1095,81	2 pontos
		1097,74	híbrido 1
		1124,92	uniforme
$N \times 10$	$N \times N$	1082,59	1 ponto
		1090,93	híbrido
		1091,37	2 pontos
		1093,07	híbrido 1
		1137,86	uniforme
$N \times 10$	$N \times N \times 10$	1075,74	1 ponto
		1078,86	híbrido 2
		1081,22	2 pontos
		1084,47	híbrido 1
		1114,16	uniforme

Tabela 7 – Tempo médio de computação para as instâncias c50, c100 e c120.

<i>Entrada</i>	<i>População</i>	<i>Evoluções</i>	<i>Tempo Médio (s)</i>
c50	N	$N \times N$	0,43
		$N \times N \times 10$	2,17
	$N \times 10$	$N \times N$	2,15
		$N \times N \times 10$	11,30
c100	N	$N \times N$	6,60
		$N \times N \times 10$	65,84
	$N \times 10$	$N \times N$	71,27
		$N \times N \times 10$	719,73
c120	N	$N \times N$	13,11
		$N \times N \times 10$	156,42
	$N \times 10$	$N \times N$	177,61
		$N \times N \times 10$	2094,30

ritmo sequencial, ocupa apenas 1 núcleo, ficando 3 núcleos ociosos. A solução mais viável é paralelizar o AG tirando melhor proveito da arquitetura multiprocessador com memória compartilhada. As questões envolvendo a paralelização do nosso AG serão apresentadas no próximo capítulo.

3.5 Considerações sobre o Algoritmo Genético Implementado

Neste capítulo definimos as características do **PRV** que foi utilizado para validar a implementação do **AG**, onde escolhemos 3 instâncias com 50, 100 e 120 cidades. Em seguida passamos a uma visão geral do algoritmo que foi implementado, descrevendo seu funcionamento. A qualidade das soluções encontradas pelos **AG** dependem de parâmetros específicos. Passamos então para a fase de definição de parâmetros. Com base na literatura, foi definida a probabilidade de mutação em 20% e, foram realizados testes para definir os melhores valores para parâmetros como a taxa de cruzamento e taxa de mutação, que ficaram em 80% e 4 a 10%, respectivamente. Foram realizados testes relacionados a técnica de mutação, que definiu a mutação do tipo **troca bloco** como melhor técnica. Nossos testes mostraram que a escolha da técnica de cruzamento depende da característica da instância de entrada. Para a instância c50 a melhor técnica de cruzamento foi o cruzamento híbrido 2, para a instância c100, o cruzamento de 2 pontos e para a instância c120, o cruzamento de 1 ponto.

Como os menores custos foram encontrados nas versões que necessitam de maior tempo de computação, uma opção viável para diminuir o tempo necessário é a paralelização. Isso além de diminuir o tempo de computação, aproveitará melhor o potencial da plataforma de testes.

O próximo capítulo apresenta os passos necessários para a paralelização da versão sequencial do **AG**.

4 Paralelização do Algoritmo Genético

4.1 Introdução

De acordo com Linden (2012), AG são excelentes candidatos a paralelização. Isso porque evoluem grandes populações de indivíduos e normalmente requerem muito tempo de execução. Outro fato que contribui para a paralelização dos AG é que muitas tarefas são realizadas sobre dados independentes e portanto podem ser executadas em paralelo. Para descobrirmos onde investir esforços de paralelização, devemos investigar o código sequencial, para descobrir quantas chamadas há para cada função, o tempo de execução dessas funções e quanto tempo a soma de suas chamadas consome em relação ao tempo total de execução do programa. Não adianta investir esforços em funções que recebem poucas chamadas e tenham um tempo de execução muito pequeno. Funções triviais que recebem muitas chamadas, ao invés de terem seu código paralelizado, devem ter suas chamadas paralelizadas. Neste capítulo veremos os passos que foram seguidos para obter a versão paralela do AG implementado.

O capítulo está organizado da seguinte forma: a Seção 2 apresenta uma análise da versão sequencial, a Seção 3 apresenta uma investigação sobre o que pode ser paralelizado, a Seção 4 apresenta a implementação da versão paralela, a Seção 5 apresenta a qualidade da solução na versão paralela, a Seção 6 apresenta a qualidade da solução na versão paralela quando executada pelo tempo da versão sequencial e a Seção 7 apresenta as considerações sobre a paralelização do AG implementado.

4.2 Análise da Versão Sequencial do Algoritmo Genético

Com todos os parâmetros definidos no capítulo anterior, esta seção irá avaliar o comportamento da nossa versão sequencial do AG. Antes de partirmos para verificação do perfil da execução propriamente dito, vamos descrever as funções em uso.

4.2.1 Funções Implementadas

Nesta seção, quando mencionamos função, estamos nos referindo a trechos de código que executam tarefas independentes em nossa implementação. A seguir são descritas as principais funções usadas pelo Algoritmo Genético implementado:

População inicial: Esta função gera todos os indivíduos da população inicial.

Para cada indivíduo, ela sorteia a primeira cidade a ser visitada e, a partir dela, busca sempre a cidade mais próxima, usando a função `Vizinho próximo`, até que tenha visitado todas as cidades.

Avaliação: Esta função é responsável por mensurar a aptidão de cada indivíduo. Essa medida é feita, considerando a distância percorrida para visitar as cidades e retornar ao depósito, quantas vezes for necessário, até que todas as rotas tenham sido atendidas e não existam mais cidades a serem visitadas. O indivíduo mais apto é aquele que conseguir atender aos requisitos, acumulando a menor distância percorrida possível. A função `Avaliação` faz chamadas sucessivas à função `Distância`.

Distância: Esta é a função que recebe dois pontos quaisquer, e retorna a distância euclidiana entre eles, usando a seguinte fórmula:

$$D = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (4.1)$$

Onde as coordenadas x e y representam a localização das cidades e são obtidas a partir do arquivo de entrada (instâncias alvo).

Nova geração: Esta função recebe uma população como entrada e constrói a nova geração utilizando os operadores genéticos. Ela escolhe aleatoriamente um indivíduo entre os 10% melhores para cruzar com um indivíduo aleatório da população. O cruzamento só ocorre se os dois indivíduos forem distintos, caso contrário outro sorteio será realizado. Esse processo se repete até que metade dos cruzamentos necessários para formar a nova geração tenham sido realizados. A outra metade da nova população é formada com cruzamentos usando indivíduos aleatórios como pais. Em todos os cruzamentos, indivíduos que tenham apenas 10% de diferença entre seus custos são considerados iguais. Cada descendente gerado, com 20% de probabilidade, sofre mutação em 4 a 10% de seus genes. Criar uma nova geração implica em várias chamadas às funções `Cruzamento`, `Mutação` e `Avaliação`, que por sua vez, faz várias chamadas à função `Distância`.

Cruzamento: Esta função recebe dois indivíduos e cruza-os utilizando a técnica de cruzamento previamente selecionada. É comum em um cruzamento que alguns genes desapareçam e que outros fiquem duplicados. Para contornar isso, essa função faz uma verificação para descobrir e corrigir inconsistências em cada um dos descendentes.

Mutação: Esta função recebe um cromossomo, aplica mutação utilizando a técnica de mutação previamente selecionada e, retorna o cromossomo alterado.

Compara solução: Esta função auxiliar compara dois custos e determina qual é menor. Ela é passada por parâmetro para a função `qsort()` que é utilizada para ordenar a população logo após sua criação.

Número randômico: Esta função auxiliar tem como objetivo gerar números pseudo-

aleatórios. Tendo em vista que a função `rand()` (função fornecida pelo *GNU Compiler Collection* (**GCC**), na biblioteca padrão, para fornecer números pseudo-aleatórios) não trabalha bem com mais de um fluxo de execução, se fez necessária a criação de uma função que suprisse essa necessidade, já que o objetivo do trabalho é paralelizar o código. Essa função foi criada com base na função apresentada na página de manual do `srand()` (função fornecida pelo **GCC** para semear a nova sequência pseudo-aleatória), que pode ser consultada na *internet*¹ ou em um terminal linux usando o comando `man srand`.

Vizinho próximo: Esta função verifica a partir de um ponto, qual é o ponto mais próximo. Ela é usada pela função **População inicial** na construção da primeira população de indivíduos.

Calcula tempo: Esta função calcula o tempo de execução do programa. Os tempos gastos com entrada e saída não são contabilizados como tempo de execução.

Na próxima seção, faremos um estudo do impacto de cada função no tempo total de execução do programa.

4.2.2 Perfil da Execução

Para gerar o perfil da execução (*profiling*) da versão sequencial foi utilizada a ferramenta **gprof** que além de ser livre, já vem incorporada ao **GCC** bastando acrescentar a opção “-pg” ao comando de compilação. O programa gerado após a compilação, sempre que executado, cria um arquivo com o perfil da execução. Para esta análise, o programa foi executado tendo como entrada a instância `c100` e, usando os parâmetros do **AG** definidos no capítulo anterior, fixando o tamanho da população em $N \times 10$ e o número de evoluções em $N \times N \times 10$.

O tempo de execução do programa com o **gprof** foi de 682,39 segundos, ou seja, 11 minutos e 22,39 segundos. A Tabela 8 apresenta o consumo de tempo de cada função do programa. Conforme podemos observar, a função **População inicial** é chamada uma única vez e seu tempo de execução é de 0,05 s. Esse tempo representa apenas 0,01% do tempo total de execução do **AG** (682,39 s).

A função **Nova geração**, foi chamada 100.000 vezes ($100 \times 100 \times 10$), e consome 3,59% do tempo total de execução. Foram feitas 40.000.000 de chamadas para a função **Cruzamento** e 80.000.000 de chamadas à função **Avaliação**. As funções **Cruzamento** e **Avaliação** usaram respectivamente 26,16% e 16,80% do tempo total de execução do programa. Nesta execução, a função **Distância** foi chamada 4.349.587.223 vezes, e usou 49,48% do tempo total de execução do programa.

A função **Número randômico** foi chamada 771.296.353 vezes, consumindo 8,77 s, o que representou 1,29% do tempo total de execução. Outra função que recebeu muitas

¹Disponível em <http://unixhelp.ed.ac.uk/CGI/man-cgi?srand+3> (acessado em janeiro de 2013).

Tabela 8 – Impacto das funções no tempo total de execução do programa, obtido pelo *gprof*.

<i>Função</i>	<i>Número de chamadas</i>	<i>Soma das chamadas (s)</i>	<i>% do Tempo total</i>
Distância	4.349.587.223	337,48 s	49,48%
Cruzamento	40.000.000	178,40 s	26,16%
Avaliação	80.000.000	114,61 s	16,80%
Nova geração	100.000	24,49 s	3,59%
Compara Solução	-	14,06 s	2,06%
Número randômico	771.296.353	8,77 s	1,29%
Mutação	16.002.992	2,48 s	0,36%
Vizinho próximo	1000	1,20 s	0,18%
Calcula tempo	1	0,82 s	0,12%
População inicial	1	0,05 s	0,01%

chamadas foi a função *Mutação*. Suas 16.002.992 chamadas consumiram 2,48 s, o que representou 0,36% do tempo total de execução.

Observando o perfil da execução, pode-se concluir que o nosso programa gastou mais tempo calculando as distâncias entre as cidades do que realizando procedimentos relacionados ao *AG*. Um agravante é que o número de chamadas da função *Distância* varia a cada execução, pois para cada indivíduo avaliado, essa função pode ser chamada $N + 1$ vezes no melhor caso (a capacidade do veículo é suficiente para atender todas as cidades sem retornar ao depósito), ou $2N$ no pior caso (a capacidade do veículo esgota-se a cada cidade, necessitando de N retornos ao depósito). Na próxima subseção demonstraremos as melhorias realizadas na versão sequencial para reduzir o impacto da função *Distância* no tempo total de execução.

4.2.3 Melhorando o Desempenho Sequencial

A função *Distância* consome uma parcela significativa do tempo total de execução do programa (conforme se observa na Tabela 8). Isto porque o algoritmo sempre calcula a distância entre dois pontos, mesmo que já tenha calculado essa distância em outro momento. Neste sentido, implementamos uma solução que calcula as distâncias uma única vez e as armazena numa matriz para consultas futuras.

O custo para armazenar as distâncias em uma matriz é pequeno quando comparado ao ganho de tempo que proporciona. Para armazenar todas as distâncias entre N cidades, é necessário criar uma matriz com $N + 1$ linhas e $N + 1$ colunas, pois temos que armazenar também a posição do depósito. Cada elemento da matriz é um *double*, logo a quantidade de memória alocada para essa matriz é $(N + 1) \times (N + 1) \times 8$ (um *double* ocupa 8 Bytes de memória). A Tabela 9 apresenta a quantidade de memória necessária para alocar matrizes de distância, considerando as instâncias que serão utilizadas neste trabalho.

Com a matriz criada, surge a necessidade de novas funções para fazerem a interface

Tabela 9 – Quantidade de memória necessária para alocar a matriz distância.

<i>Entrada</i>	<i>Tamanho em KBytes</i>
c50	20,32
c100	79,69
c120	114,38

Tabela 10 – Impacto das funções no tempo total de execução da nova versão do programa, obtido pelo *gprof*.

<i>Função</i>	<i>Número de chamadas</i>	<i>Soma das chamadas (s)</i>	<i>% do Tempo total</i>
Cruzamento	40.000.000	179,99 s	37,42%
Busca distância	4.349.590.210	125,49 s	26,09%
Avaliação	80.000.000	122,05 s	25,37%
Nova geração	100.000	23,72 s	4,93%
Compara Solução	-	14,44 s	3,00%
Número randômico	769.795.125	7,53 s	1,57%
Calcula Distâncias	1	3,46 s	0,72%
Mutação	15.996.602	2,12 s	0,44%
Vizinho próximo	1000	1,42 s	0,30
Calcula tempo	1	0,94 s	0,20%
População inicial	1	0,05 s	0,01%
Distância	5050	0,00 s	0,00%

da mesma:

Calcula distância: Função que faz uso da antiga função **Distância** para preencher a matriz com as distâncias entre os pontos.

Busca distância: Função que recebe dois pontos como parâmetro, consulta a matriz distâncias e retorna a distância que está armazenada na linha e coluna correspondentes.

O perfil da execução da nova versão sequencial pode ser visto na Tabela 10. O uso da matriz de distâncias impactou no tempo total de execução, reduzindo de 682,39 s para 481,25 s. O novo mecanismo usou apenas 26,81% do tempo de execução (tempo da **Calcula distância** + tempo da **Busca distância**) enquanto que a versão antiga usava 49,48%. A função **Calcula distância** levou apenas 3,46 s para preencher a matriz. As funções **Cruzamento** e **Avaliação** passaram a consumir 37,42% e 25,37% do tempo de execução, respectivamente. As 5.050 chamadas da função **Distância** consumiram tão pouco tempo que duas casas decimais não foram suficientes para registrá-lo.

A Tabela 11 apresenta os tempos médios de computação da versão sequencial, sem o uso do **gprof**, para as instâncias c50, c100 e c120, obtidos com o novo mecanismo para o cálculo das distâncias e, para comparação, os obtidos com o mecanismo antigo.

Na próxima seção investigaremos o que pode ser paralelizado.

Tabela 11 – Comparação entre os tempos médios de computação utilizando os dois diferentes mecanismos para cálculo das distâncias.

<i>Entrada</i>	<i>Tempo médio com cálculo prévio</i>	<i>Tempo médio com cálculo sob demanda</i>
c50	7,54 s	11,30 s
c100	517,83 s	719,13 s
c120	1553,98 s	2094,30 s

4.3 O que pode ser paralelizado?

Para respondermos a essa pergunta, precisamos analisar o algoritmo sequencial e o perfil da execução em conjunto.

Observando novamente a Tabela 10, verificamos que a função **Cruzamento** recebeu 40.000.000 chamadas, que representaram 37,42% do tempo total de execução. Esta função pode ser paralelizada pois não possui dependência de dados. Ela recebe dois indivíduos previamente selecionados, os processa e retorna outros dois indivíduos. Por outro lado, ela é chamada diversas vezes em cada geração, mas cada chamada consome apenas 0,0000045 segundos (4,5 μ s). O grande número de chamadas é que faz com que essa função tenha grande impacto no tempo total de execução, sendo mais viável que receba chamadas paralelas ao invés de ter seu código paralelizado.

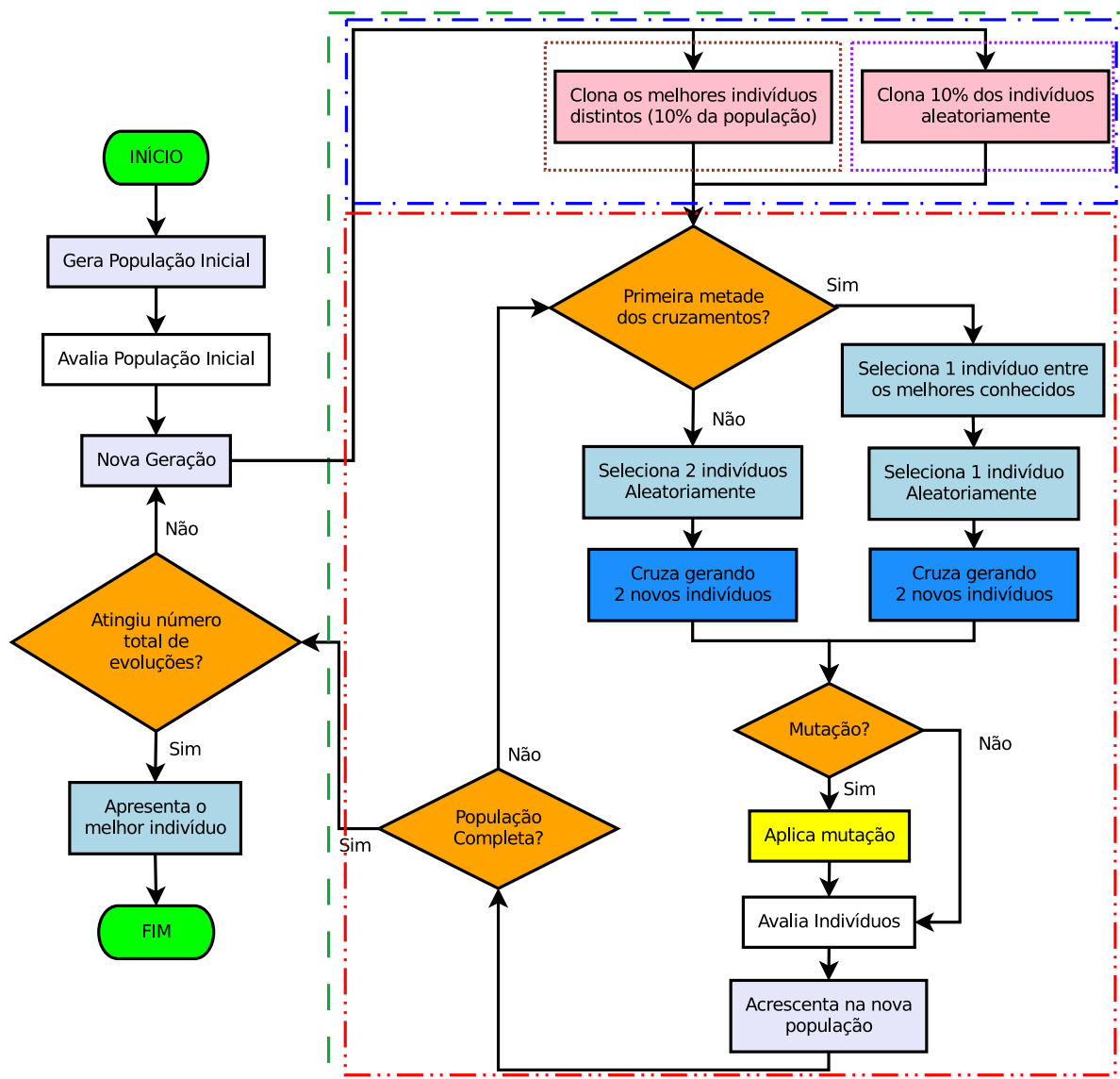
A função **Avaliação** recebeu 80.000.000 chamadas, que representaram 25,37% do tempo total de execução. Ela recebe o dobro do número de chamadas da função **Cruzamento**, mas cada chamada consome apenas 0,0000015 segundos (1,5 μ s). Essa função faz chamadas sucessivas à função **Busca distância** e não é paralelizável pois possui dependência de dados. Ela começa o cálculo do custo do indivíduo computando a distância entre o depósito e a primeira cidade a ser visitada. Se após atender essa cidade, ainda houver capacidade de atendimento para a próxima, a função computa a distância entre a cidade atual e a próxima a ser visitada. Caso não haja capacidade de atendimento, é computada a distância entre a cidade atual e o depósito e entre o depósito e a próxima cidade. Esse processo se repete até que a última cidade tenha sido visitada e tenha sido computada a distância entre ela e o depósito. O que impede que o cálculo da distância total percorrida seja efetuado em paralelo, é que o cromossomo não armazena o momento de retornar ao depósito. Este momento é calculado com base na capacidade de atendimento do veículo. Desta forma, esta função é candidata a receber chamadas paralelas.

A função **Busca distância** recebeu 4.349.590.210 chamadas, que representaram 26,09% do tempo total de execução. Cada chamada consumiu apenas 0,000000029 segundos (29 ns). Esta função não possui dependência de dados. Ela recebe dois inteiros que representam a linha e a coluna que serão consultadas na matriz com as distâncias previ-

amente calculadas. Faz um acesso à memória e retorna o valor armazenado na posição solicitada. Desta forma, não é viável que receba esforços de paralelização pois ela apenas realiza um acesso à memória.

A função *Nova geração* recebeu 100.000 chamadas, que representaram apenas 4,93% do tempo total de execução. A cada geração, esta função é chamada uma única vez, portanto suas chamadas não podem ser paralelizadas. Suas instruções, por outro lado, são fortes candidatas a paralelização já que após essa função executar um trecho de código dedicado a perpetuação de indivíduos, passa a executar um laço que faz chamadas diretas às funções *Cruzamento* e *Avaliação* e indiretas à função *Busca distância*, que juntas são responsáveis por 88,88% do tempo de execução do programa.

Figura 18 – Fluxograma do algoritmo genético sequencial implementado.



A Figura 18 apresenta o fluxograma do algoritmo sequencial apresentado no capítulo anterior. As funções que mais impactam no tempo total de execução recebem

chamadas na função `Nova geração`, representada pela área envolvida por um retângulo verde (tracejado). A área envolvida por um retângulo azul (tracejado separado por ponto) representa o trecho de código que perpetua 10% dos melhores indivíduos e 10% de indivíduos aleatórios. A área envolvida por um retângulo vermelho (tracejado separado por dois pontos) representa o laço que controla as chamadas das funções `Cruzamento` e `Avaliação`. Portanto, se paralelizarmos as instruções da função `Nova geração`, estaremos fazendo chamadas diretas às funções `Cruzamento` e `Avaliação` e indiretas à função `Busca distância`.

4.4 Implementação da versão Paralela

Para iniciarmos a implementação da versão paralela com OpenMP, devemos identificar as regiões paralelas na função alvo e qual a melhor diretiva de paralelização a ser aplicada.

4.4.1 Definindo as Regiões Paralelas

Na Figura 18, a função `Nova geração` (retângulo tracejado verde) inicia procurando por 10% dos melhores indivíduos distintos e os copiando para a nova geração. Em seguida, copia outros 10% de indivíduos aleatórios. Embora essas tarefas estejam em uma sequência cronológica, elas podem ser feitas ao mesmo tempo ou na ordem inversa, pois não possuem dependência de dados. Esta é a primeira região a ser paralelizada.

Imediatamente após perpetuar os indivíduos, a função inicia o laço que controla os cruzamentos e faz chamadas às funções `Cruzamento` e `Avaliação`. Este laço é executado até que sejam criados todos os indivíduos necessários para completar a nova geração. As operações executadas neste laço também não possuem dependência de dados, e cada iteração consome em média 0,000008 segundos (8 μ s). Este tempo é a soma de 1 chamada direta à função `Cruzamento`, 2 chamadas diretas à função `Avaliação`, N chamadas indiretas à função `Busca distância` e algumas operações de controle. Esta é a segunda região a ser paralelizada.

Ambas as regiões paralelas foram delimitadas usando a diretiva `#pragma omp parallel`. Essa diretiva informa ao compilador que a região pode ser executada em paralelo. O tipo de paralelização aplicado a cada região depende das características do bloco de instruções que será executado.

A primeira região paralela possui dois blocos de código que podem ser executados em paralelo. Para paralelização de blocos não iterativos, o OpenMP provê a diretiva `#pragma omp sections`. Usando esta diretiva, as tarefas são divididas, ficando cada bloco de código dentro de uma *section*. Cada *section* é executada por uma *thread* diferente.

Neste caso, como são dois blocos de código, essas tarefas serão sempre executadas por duas *threads*.

A segunda região paralela possui tarefas que são controladas por um laço. Para paralelização de laços, o OpenMP provê a diretiva `#pragma omp for`. Usando esta diretiva, as iterações do laço paralelizado são executadas por um número definido de *threads*, que pode ser setado usando a rotina `omp_set_num_threads(int num_threads)`.

Com essas diretivas inseridas no código, já é possível testar o desempenho da versão paralela.

4.4.2 Métricas de Desempenho

As métricas de desempenho utilizadas para análise dos resultados foram o *Speedup* e a Eficiência. De acordo com [Wilkinson e Allen \(2005\)](#) um dos primeiros pontos de interesse quando se desenvolve programas para multiprocessadores é saber quão rápido o multiprocessador resolve o problema considerado. A métrica utilizada para quantificar essa rapidez do multiprocessador chama-se *speedup* e compara o tempo obtido por um processador sequencial ao tempo obtido por um multiprocessador.

Conforme [Rauber e Rünger \(2010\)](#) relatam, o *speedup* $S_p(n)$ é dado por:

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \quad (4.2)$$

onde $T^*(n)$ é o melhor tempo de execução sequencial, $T_p(n)$ é o tempo de execução paralelo e p é o número de processadores usados para resolver o problema. O *speedup* ideal é obtido quando temos seu valor igual a p , ou seja, o ganho é proporcional ao número de processadores utilizados. Entretanto, existem casos em que o *speedup* é maior que p . Nesses casos de acordo com [Wilkinson e Allen \(2005\)](#) temos um *superlinear speedup* o que significa que nosso algoritmo sequencial não é ótimo e que pode ser melhorado.

Segundo [Rauber e Rünger \(2010\)](#), uma métrica alternativa para medir o desempenho é a eficiência, que nos diz o quanto o processador foi usado durante a execução do programa. A eficiência $E_p(n)$ é dada por:

$$E_p(n) = \frac{T^*(n)}{p \times T_p(n)} \quad (4.3)$$

onde $T^*(n)$ é o melhor tempo de execução sequencial, $T_p(n)$ é o tempo de execução paralelo e p é o número de processadores usados para resolver o problema. [Rauber e Rünger \(2010\)](#) dizem que o *speedup* ideal $S_p(n) = p$ corresponde a $E_p(n) = 1$.

4.4.3 Variando o Número de Threads

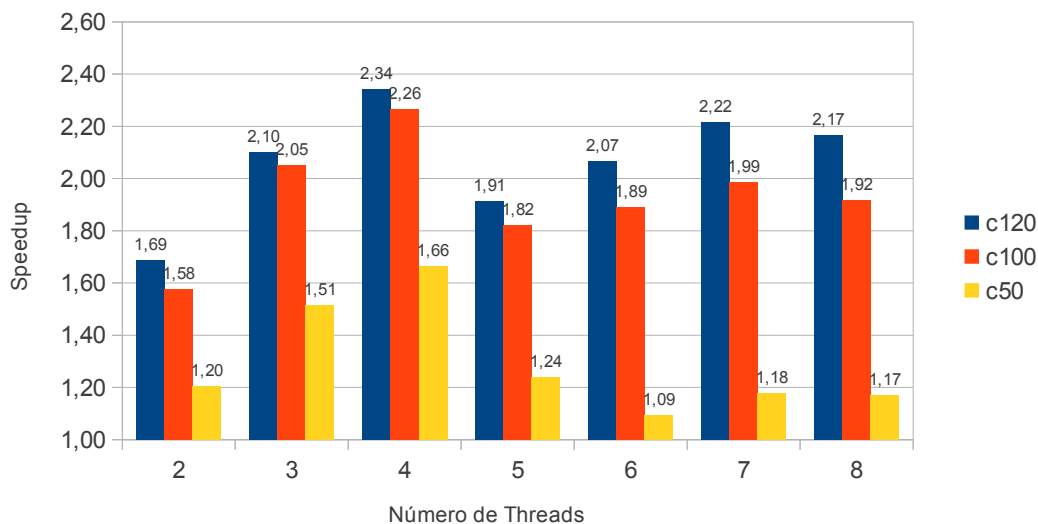
O microcomputador usado nos testes está equipado com um multiprocessador Intel® Core 2 Quad Q8300, que possui 4 núcleos físicos. O primeiro teste realizado, con-

sistiu em executar o código paralelo, variando o número de *threads* de 2 a 8, considerando o *schedule* padrão (tipo *static*, *chunk* 1). Os resultados deste teste podem ser conferidos nas Figuras 19 e 20. Nela percebemos que para a instância c50, embora tenha existido algum ganho no tempo de execução, ele foi abaixo do ideal. Para 2, 3 e 4 *threads*, os *speedups* foram de apenas 1,2, 1,51 e 1,66 enquanto que o *speedup* ideal seria 2, 3 e 4, respectivamente. Para 5, 6, 7 e 8 *threads*, os *speedups* foram de 1,24, 1,09, 1,18 e 1,17, enquanto que o ideal seria 5, 6, 7 e 8, respectivamente. A eficiência também ficou abaixo do ideal, sendo de 0,6, 0,5, 0,42 para 2, 3 e 4 *threads*, respectivamente, e de 0,25, 0,18, 0,17 e 0,15 para 5, 6, 7 e 8 *threads*, respectivamente, enquanto que o valor ideal seria 1 para qualquer número de *threads*.

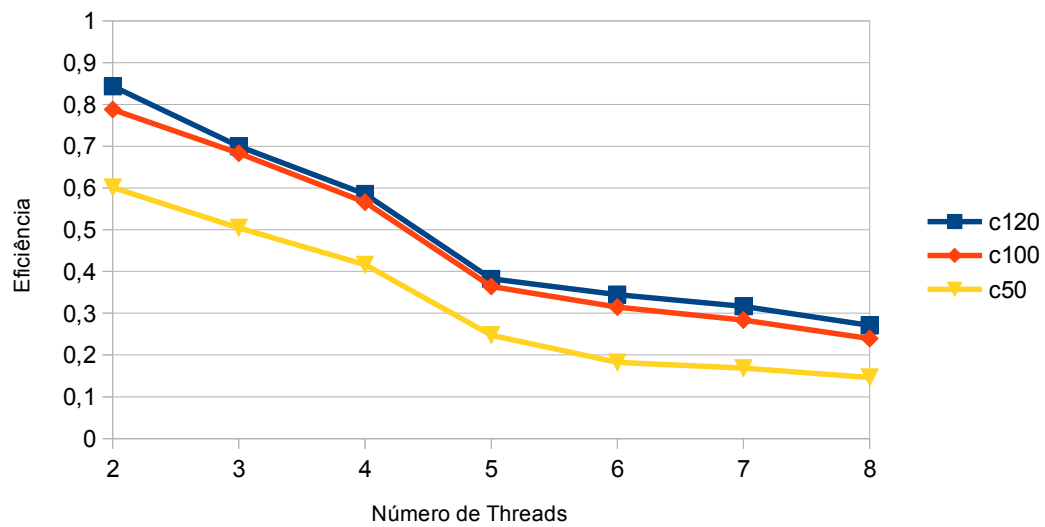
Para a instância c100, os ganhos no tempo de execução foram pouco maiores. Com 2, 3 e 4 *threads*, os *speedups* foram de 1,58, 2,05 e 2,26, respectivamente e com 5, 6, 7 e 8 *threads*, os *speedups* foram de 1,82, 1,89, 1,99 e 1,92. A eficiência foi de 0,79, 0,68, 0,57 para 2, 3 e 4 *threads*, respectivamente, e de 0,36, 0,31, 0,28 e 0,24 para 5, 6, 7 e 8 *threads*, respectivamente.

Para a instância c120, com 2, 3 e 4 *threads*, o *speedup* foi de 1,69, 2,10 e 2,34, respectivamente. Para 5, 6, 7 e 8 *threads*, os *speedups* foram de 1,91, 2,07, 2,22 e 2,17, respectivamente. A eficiência foi de 0,84, 0,70, 0,59 para 2, 3 e 4 *threads*, respectivamente, e de 0,38, 0,34, 0,32 e 0,27 para 5, 6, 7 e 8 *threads*, respectivamente.

Figura 19 – Instâncias c50, c100 e c120 submetidas às versões paralelas até 8 *threads*: *Speedup*.



Na Figura 19 percebemos que os melhores *speedups* foram encontrados usando até 4 *threads*, enquanto que na Figura 20 percebemos que a eficiência cai muito a partir de 4 *threads*. Por esse motivo, em todos os próximos testes, o número de *threads* considerado foi 2, 3 e 4.

Figura 20 – Instâncias c50, c100 e c120 submetidas às versões paralelas até 8 *threads*: Eficiência.

Para melhorar o *speedup*, ainda é possível testar diferentes tipos de *schedule* juntamente com diferentes tamanhos de *chunk*.

4.4.4 Variando a Forma de Escalonamento

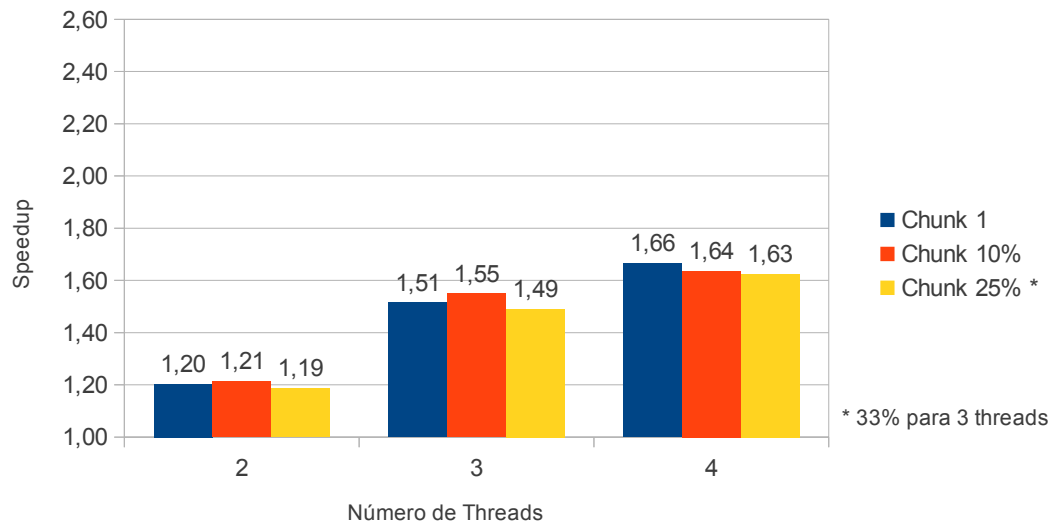
É possível configurar a forma de escalonamento das iterações entre as *threads* e definir o tamanho dos blocos de iterações distribuídos a cada uma, usando a cláusula `schedule(type [, chunk])`. O *type* é o tipo de escalonamento e o *chunk* é tamanho dos blocos de iterações que pode ser passado como parâmetro à cláusula e que deve ser um inteiro maior que zero.

Foram feitos testes com escalonamento estático, dinâmico e guiado. Para cada tipo, foram testados 3 tamanhos de *chunk*. Para 2 e 4 *threads* foram testados dos tamanhos 1, 10% e 25% do bloco de iterações. Para 3 *threads* foram testados os tamanhos 1, 10% e 33% do bloco de iterações. O *chunk* 1 é a granularidade fina, onde as iterações são atribuídas às *threads* uma a uma. O *chunk* 10% entrega às *threads* blocos com 10% do total de iterações, representando uma granularidade média. Já o *chunk* 25% estabelece blocos com 25% do total de iterações visando testar um bloco de granularidade grossa. É importante notar que o *chunk* 25% é compatível com números pares de *threads*. Para nossos testes com 3 *threads*, a granularidade grossa usa blocos com 33% do total de iterações.

As Figuras 21, 22 e 23 apresentam a comparação entre os *speedups* obtidos usando escalonamento estático, dinâmico e guiado, variando o tamanho dos blocos de iterações distribuído às *threads* para a instância c50. O número total de iterações para essa instância é 200.

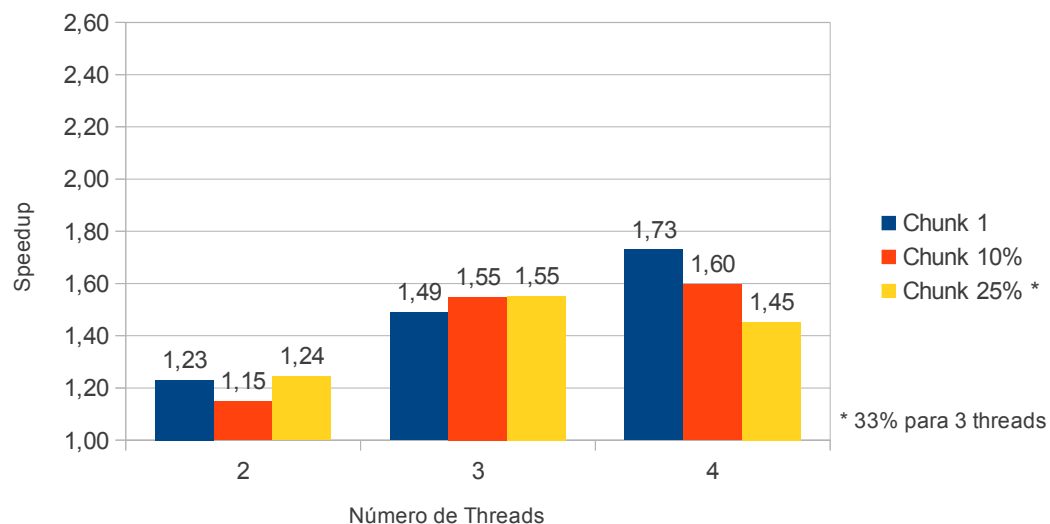
Os *speedups* da instância c50 com *schedule* estático são apresentados na Figura

Figura 21 – Instância c50 submetida à versão paralela com *schedule* estático: *Speedup* variando o *chunk* e o número de *threads*.



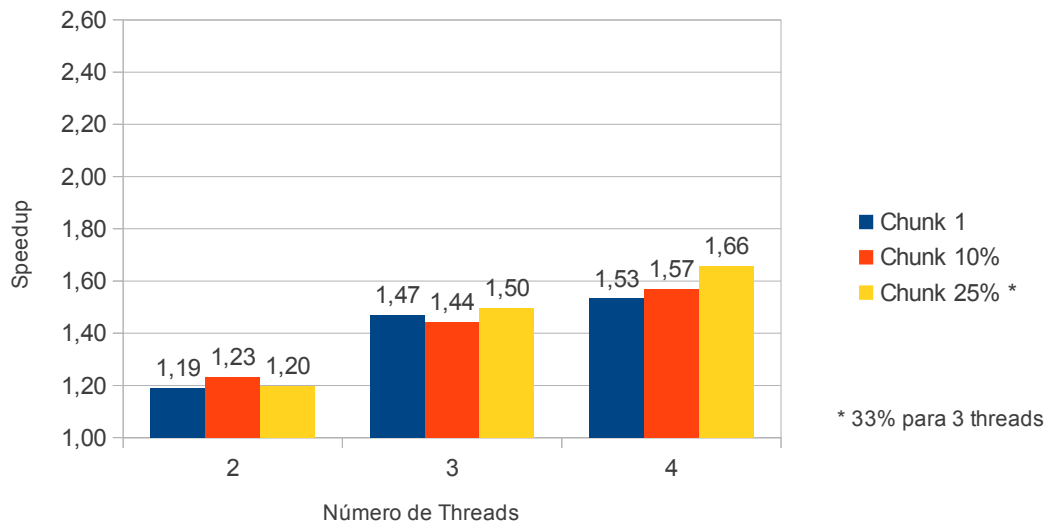
21. Nela percebemos que para 2 e 3 *threads* os maiores valores foram obtidos usando granularidade média. Já para 4 *threads*, o maior valor foi obtido usando granularidade fina.

Figura 22 – Instância c50 submetida à versão paralela com *schedule* dinâmico: *Speedup* variando o *chunk* e o número de *threads*.



Os *speedups* da instância c50 com *schedule* dinâmico são apresentados na Figura 22. Nela percebemos que para 2 *threads* o maior valor foi obtido usando granularidade grossa. Para 3 *threads*, a granularidade média e a grossa obtiveram o maior valor. Já para 4 *threads*, o maior valor foi obtido usando granularidade fina.

Figura 23 – Instância c50 submetida à versão paralela com *schedule* guiado: *Speedup* variando o *chunk* e o número de *threads*.



Os *speedups* da instância c50 com *schedule* guiado são apresentados na Figura 23. Nela percebemos que para 2 *threads* o maior valor foi obtido usando granularidade média. Para 3 e 4 *threads*, foi obtido com granularidade grossa.

Para a instância c50, o escalonamento dinâmico conduziu aos maiores *speedups* independente do número de *threads*. O tamanho do bloco que proporcionou maior eficiência com escalonamento dinâmico variou conforme o número de *threads*. Para 2 e 3 *threads* foi a granularidade grossa e, para 4 *threads*, a granularidade fina. Vale ressaltar que para o escalonamento dinâmico, a granularidade grossa tende a ser mais eficiente quando as iterações tem o tempo de computação igual ou muito parecidos. Assim, as *threads* terminam a execução de seus blocos aproximadamente ao mesmo tempo e o custo adicional para distribuir os blocos é pequeno para poucos blocos. A granularidade fina com escalonamento dinâmico tende a ser mais eficiente quando o tempo de execução de cada iteração é diferente. Assim, sempre que uma *thread* termina seu trabalho, recebe uma nova iteração para computar, equilibrando a distribuição da carga entre as *threads*.

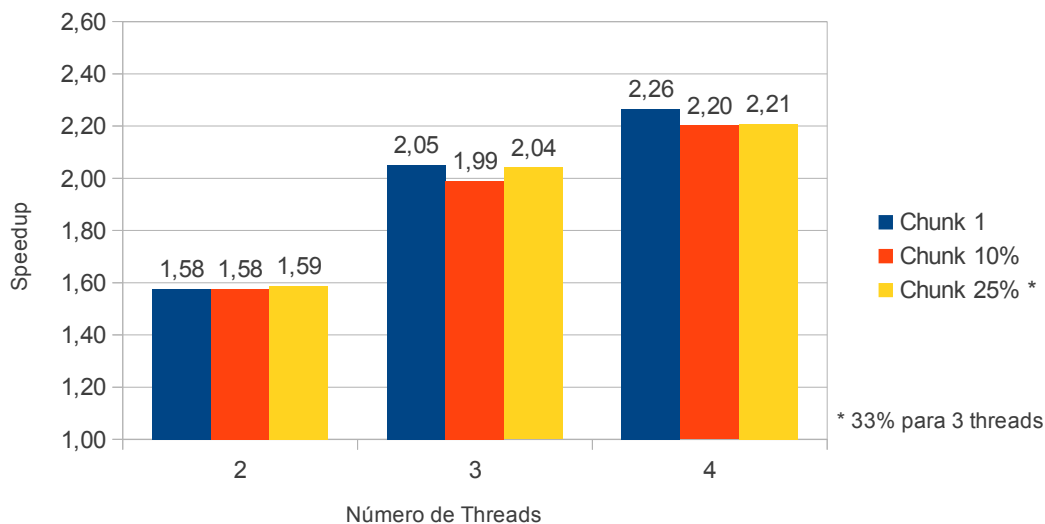
O tempo de computação de cada iteração nesta aplicação, independente da entrada, não é constante. No início, quando a população é diversificada, esse tempo tende a ser muito parecido. Durante as evoluções, conforme as populações comecem a convergir para estabilidade, existe uma chance maior de os indivíduos sorteados para o cruzamento serem considerados iguais. Ocorrendo isso, novos sorteios são realizados até que sejam selecionados indivíduos distintos. Dessa forma, o comportamento da execução vai depender do momento em que a população começar a convergir. Por duas razões, quanto mais tarde a população convergir, melhor. Com uma população diversificada, aumentam as chances de se encontrar boas soluções e, havendo poucos sorteios para seleção de indivíduos dis-

tintos, mais rápida se torna a execução. Como o tempo de computação da instância c50 é muito pequeno, essa análise fica um pouco dificultada.

O melhor *speedup* para a instância c50 foi obtido usando escalonamento dinâmico, 4 *threads* e granularidade fina. Essa combinação fez com que o tempo de execução baixasse de 7,54 s na versão sequencial para 4,36 s nesta versão.

As Figuras 24, 25 e 26 apresentam a comparação entre os *speedups* obtidos usando escalonamento estático, dinâmico e guiado, variando o tamanho dos blocos de iterações distribuído às *threads* para a instância c100. O número total de iterações para essa instância é 800.

Figura 24 – Instância c100 submetida à versão paralela com *schedule* estático: *Speedup* variando o *chunk* e o número de *threads*.



Os *speedups* da instância c100 com *schedule* estático são apresentados na Figura 24. Nela percebemos que para 2 *threads* o maior valor foi obtido usando granularidade grossa. Para 3 e 4 *threads*, foi obtido usando granularidade fina.

Os *speedups* da instância c100 com *schedule* dinâmico são apresentados na Figura 25. Nela percebemos que para 2 *threads* o maior valor foi obtido usando granularidade fina. Para 3 e 4 *threads* foi obtido usando granularidade grossa.

Os *speedups* da instância c100 com *schedule* guiado são apresentados na Figura 26. Nela percebemos que para 2 *threads* o maior valor foi obtido usando granularidade média. Para 3 e 4 *threads*, foi obtido usando granularidade grossa.

Para a instância c100, o escalonamento estático conduziu a melhores *speedups*, independente do número de *threads*. O tamanho do bloco que proporcionou maior eficiência com escalonamento estático variou conforme o número de *threads*. Para 2 *threads* foi a granularidade grossa, já para 3 e 4 *threads*, a granularidade fina. A distribuição dos blocos

Figura 25 – Instância c100 submetida à versão paralela com *schedule* dinâmico: *Speedup* variando o *chunk* e o número de *threads*.

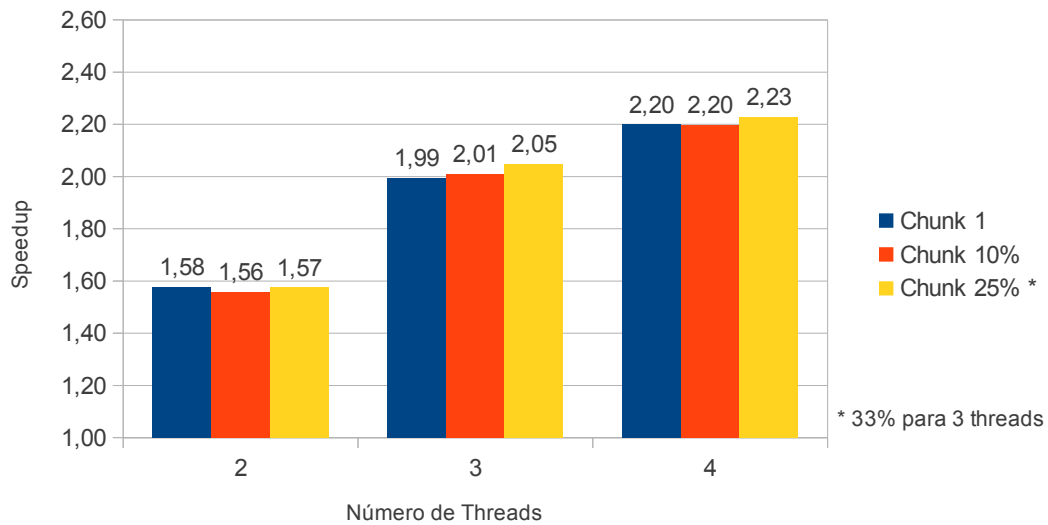
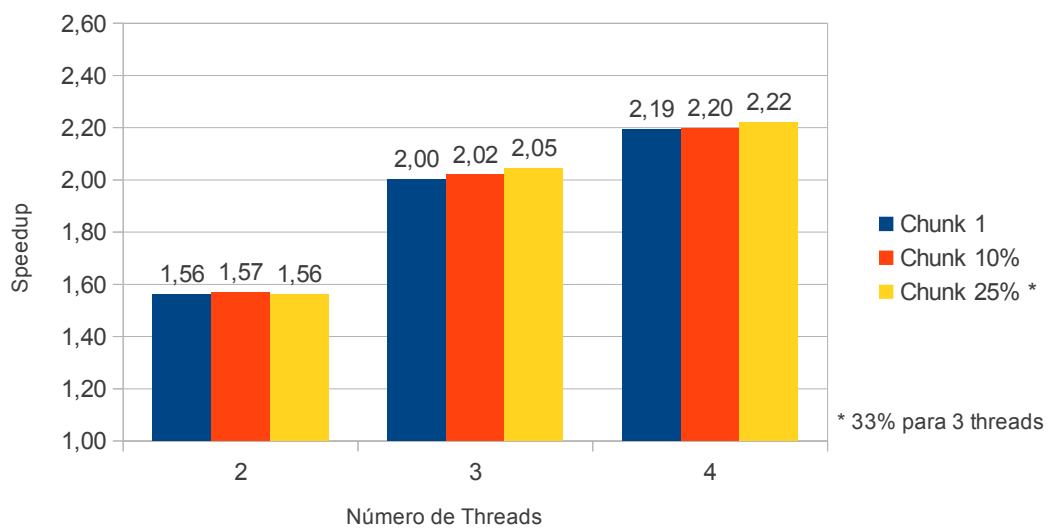


Figura 26 – Instância c100 submetida à versão paralela com *schedule* guiado: *Speedup* variando o *chunk* e o número de *threads*.

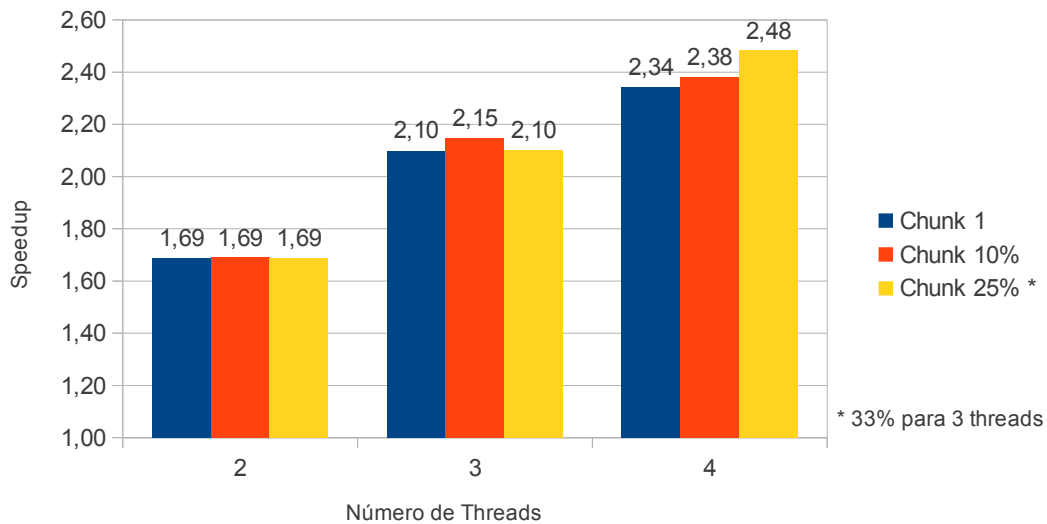


de iterações às *threads* usando escalonamento estático é feita em tempo de compilação, o que não acarreta em custo adicional para distribuição da carga, conforme ocorre com o escalonamento dinâmico. A melhor eficiência se dá quando o tempo de computação de cada iteração é igual ou muito parecido, pois assim as *threads* tendem a terminar o trabalho ao mesmo tempo.

O melhor *speedup* para a instância c100 foi obtido usando escalonamento estático, 4 *threads* e granularidade fina. Essa combinação fez com que o tempo de execução baixasse de 517,83 s na versão sequencial para 228,71 s nesta versão.

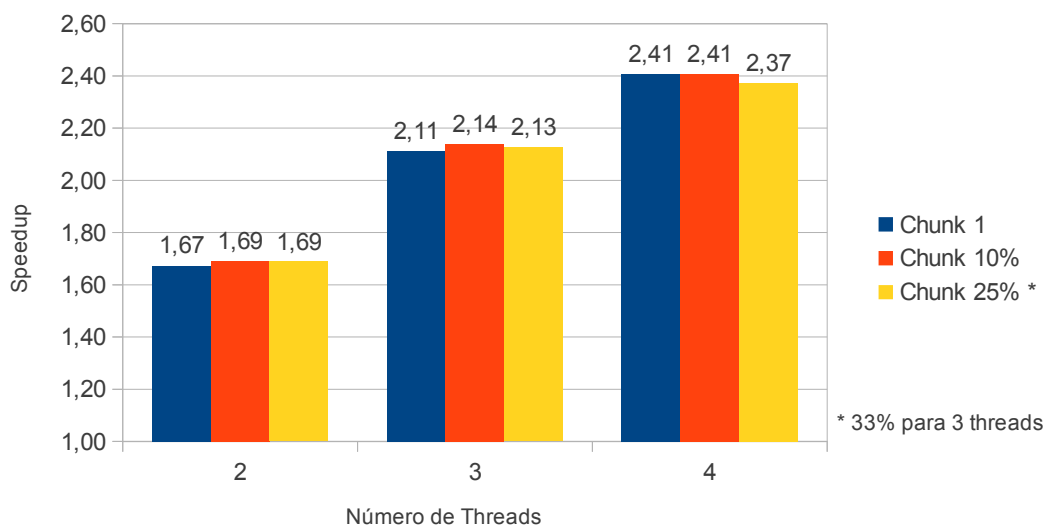
As Figuras 27, 28 e 29 apresentam a comparação entre os *speedups* obtidos usando escalonamento estático, dinâmico e guiado, variando o tamanho dos blocos de iterações distribuído às *threads* para a instância c120. O número total de iterações para essa instância é 1152.

Figura 27 – Instância c120 submetida à versão paralela com *schedule* estático: *Speedup* variando o *chunk* e o número de *threads*.



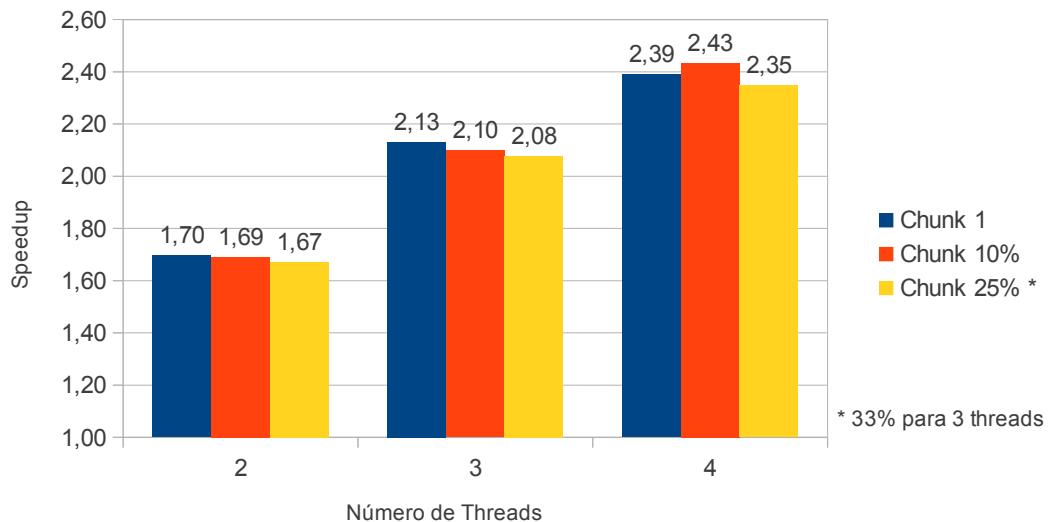
Os *speedups* da instância c120 com *schedule* estático são apresentados na Figura 27. Nela percebemos que para 2 *threads*, o tamanho do bloco não interferiu no *speedup*. Para 3 *threads*, o maior valor foi obtido usando granularidade média. Para 4 *threads*, o maior valor foi obtido usando granularidade grossa.

Figura 28 – Instância c120 submetida à versão paralela com *schedule* dinâmico: *Speedup* variando o *chunk* e o número de *threads*.



Os *speedups* da instância c120 com *schedule* dinâmico são apresentados na Figura 28. Nela percebemos que para 2 *threads* o maior valor foi obtido granularidade média e grossa. Para 3 *threads*, foi usando granularidade média e para 4 *threads*, usando granularidade fina e média.

Figura 29 – Instância c120 submetida à versão paralela com *schedule* guiado: *Speedup* variando o *chunk* e o número de *threads*.



Os *speedups* da instância c120 com *schedule* guiado são apresentados na Figura 29. Nela percebemos que para 2 e 3 *threads* o maior valor foi obtido usando granularidade fina. Para 4 *threads*, foi usando granularidade média.

Para a instância c120, o escalonamento estático conduziu a melhores *speedups*, usando 3 e 4 *threads*. Para 2, o melhor *speedup* foi alcançado com escalonamento guiado. O tamanho do bloco que proporcionou maior eficiência variou conforme o número de *threads*. Para 2 *threads* foi a granularidade fina. A distribuição dos blocos de iterações às *threads* usando escalonamento guiado é feita em tempo de execução, começando por blocos grandes que diminuem gradativamente até o tamanho do *chunk* especificado. Para *chunk* 1, este modelo é parecido com o dinâmico, mas tem menor custo de distribuição da carga, pois o tamanho do bloco começa grande e vai diminuindo, o que implica em menos blocos para distribuir. Para 3 *threads*, a granularidade média conduziu ao melhor *speedup*. Para entendermos esse comportamento, seria necessário realizar testes mais detalhados, já que com escalonamento estático, uma das 3 *threads* receberia 4 blocos de iteração enquanto as outras duas receberiam apenas 3 e, ainda assim essa configuração conduziu ao melhor *speedup*. Não foram conduzidos testes detalhados porque o melhor *speedup* para a instância c120 foi obtido usando escalonamento estático, 4 *threads* e granularidade grossa. Essa combinação fez com que o tempo de execução baixasse de 1553,98 s na versão sequencial para 626,09 s nesta versão.

A Tabela 12 apresenta as instâncias e a configuração das versões que obtiveram o melhor *speedup*, bem como o tempo de computação médio que cada configuração consumiu. Observamos que nesta aplicação, o *speedup* é maior conforme aumentamos o número de *threads* (2 a 4) e é também maior conforme aumentamos a entrada.

Tabela 12 – Configuração das versões paralelas que obtiveram os melhores *speedups* e tempos de execução.

<i>Instância</i>	<i>Versão</i>	<i>Schedule</i>	<i>Chunk</i>	<i>Speedup</i>	<i>Tempo (s)</i>
c50	2 <i>threads</i>	dinâmico	25%	1,24	6,06
	3 <i>threads</i>	estático	10%	1,55	4,87
	3 <i>threads</i>	dinâmico	10%	1,55	4,87
	3 <i>threads</i>	dinâmico	33%	1,55	4,86
	4 <i>threads</i>	dinâmico	1	1,73	4,36
c100	2 <i>threads</i>	estático	25%	1,59	326,07
	3 <i>threads</i>	estático	1	2,05	252,62
	3 <i>threads</i>	dinâmico	33%	2,05	252,82
	3 <i>threads</i>	guiado	33%	2,05	253,16
	4 <i>threads</i>	estático	1	2,26	228,71
c120	2 <i>threads</i>	guiado	1	1,70	916,09
	3 <i>threads</i>	estático	10%	2,15	723,71
	4 <i>threads</i>	estático	25%	2,48	626,09

4.5 Qualidade da Solução

Desde os testes iniciais, a definição de parâmetros foi guiada pela qualidade da solução. Esta qualidade era verificada, considerado a média dos custos encontrados em 30 execuções. Nesta seção, a análise não considera apenas a média, considera também a frequência com que os custos são encontrados e considera o menor (melhor) e o maior (pior) custos encontrados.

O menor custo encontrado, considerando todos os testes realizados neste trabalho incluindo as versões sequenciais, para a instância c50 foi encontrado por diversas configurações de tamanho da população, número de evoluções, *schedule* e granularidade. Este custo foi de 543,19 Km e é apenas 18,58 Km maior que a melhor solução conhecida. Para a instância c100, o menor custo foi encontrado pela versão paralela com 3 *threads*, tamanho da população $N \times 10$, número de evoluções $N \times N \times 10$, *schedule* estático e granularidade média. Este custo foi de 868,15 Km e, é apenas 42,01 Km maior que a melhor solução conhecida. Para a instância c120, o menor custo foi encontrado pela versão paralela com 7 *threads*, tamanho da população $N \times 10$, número de evoluções $N \times N \times 10$, *schedule* dinâmico e granularidade fina. Este custo foi de 1064,87 Km e, é apenas 22,76 Km maior que a melhor solução conhecida.

Embora não exista uma relação entre *speedup* e qualidade da solução, a seguir

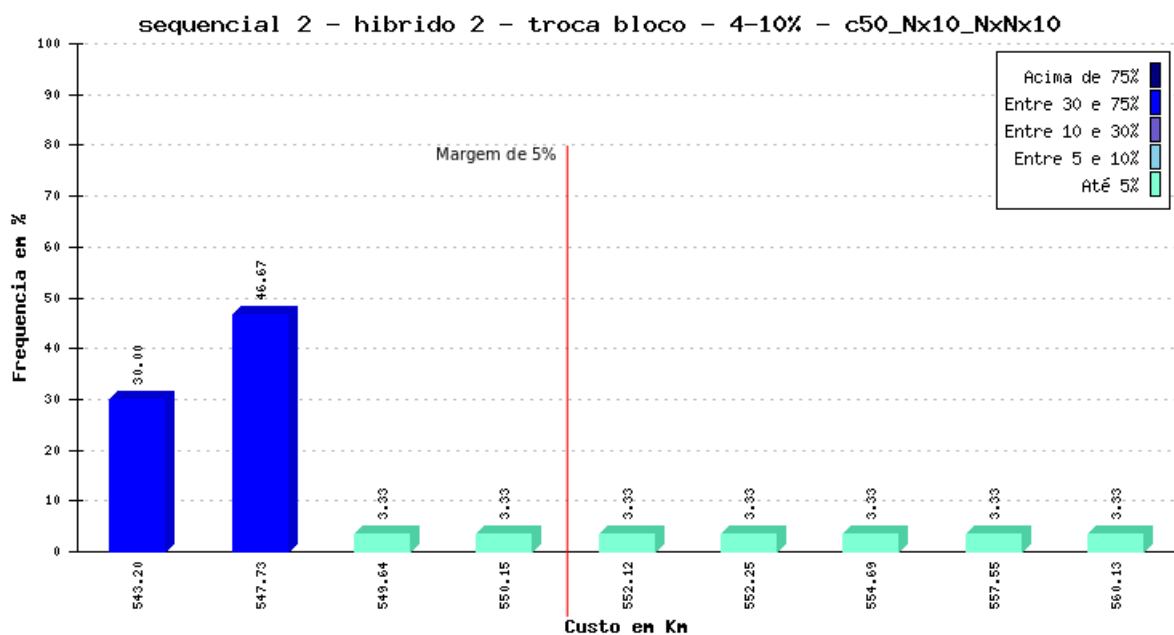
veremos uma comparação entre todos os custos encontrados pelas versões que obtiveram os melhores *speedups* (tempo de execução menor) com os encontrados pela versão sequencial.

4.5.1 Instância c50

Para a instância c50, que tem como melhor custo conhecido o valor de 524,61 Km, os custos encontrados em 30 execuções de cada versão do programa são apresentados a seguir.

A Figura 30 apresenta todos os custos obtidos com a instância c50 submetida à versão sequencial 2. Nela podemos observar que 83,25% dos custos ficaram dentro de uma margem entre a melhor solução conhecida e 5% de tolerância, ou seja, entre 524,11 Km e 550,84 Km. A melhor solução encontrada foi 543,19 Km e, é apenas 18,58 Km maior que a melhor solução conhecida.

Figura 30 – Instância c50 submetida à versão sequencial 2: frequência dos custos encontrados.



A Figura 31 apresenta todos os custos obtidos com a instância c50 submetida a versão paralela com 2 *threads*, *schedule* dinâmico e granularidade grossa. Nela podemos observar que 73,26% dos custos ficaram abaixo dos 550,84 Km. A melhor solução encontrada foi idêntica à solução encontrada pela versão sequencial.

A Figura 32 apresenta todos os custos obtidos com a instância c50 submetida à versão paralela com 3 *threads*, *schedule* dinâmico e granularidade grossa. Nela podemos observar que os 66,6% dos custos ficaram abaixo dos 550,84 Km. A melhor solução encontrada também foi idêntica à melhor solução encontrada pela versão sequencial.

Figura 31 – Instância c50 submetida à versão com 2 threads, schedule dinâmico e chunk 25%: frequência dos custos encontrados.

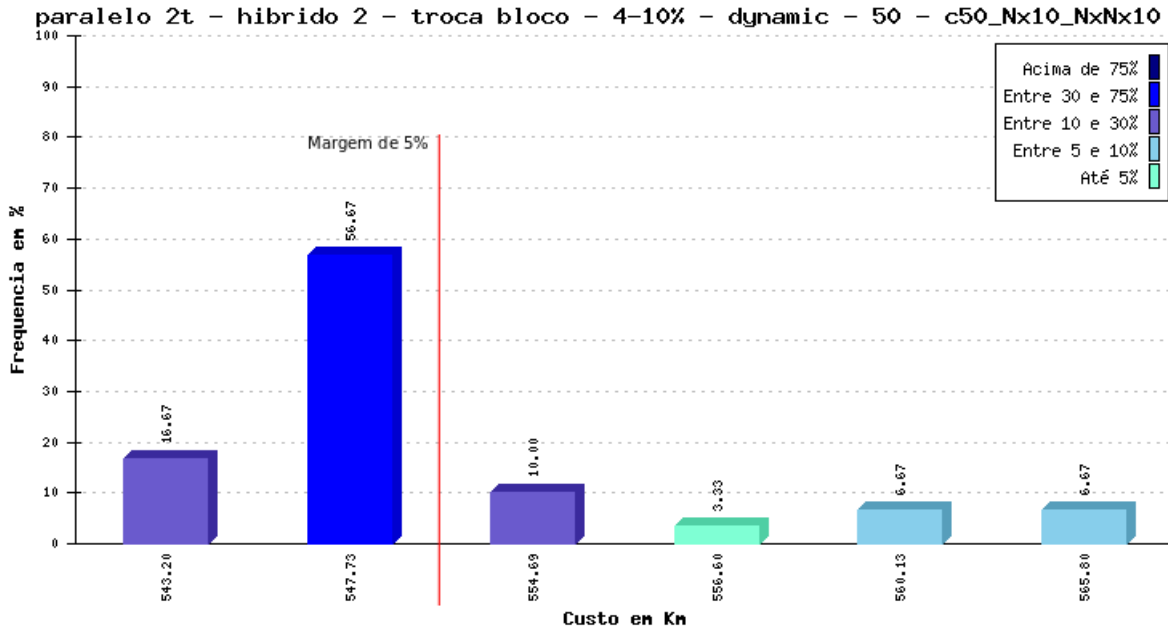
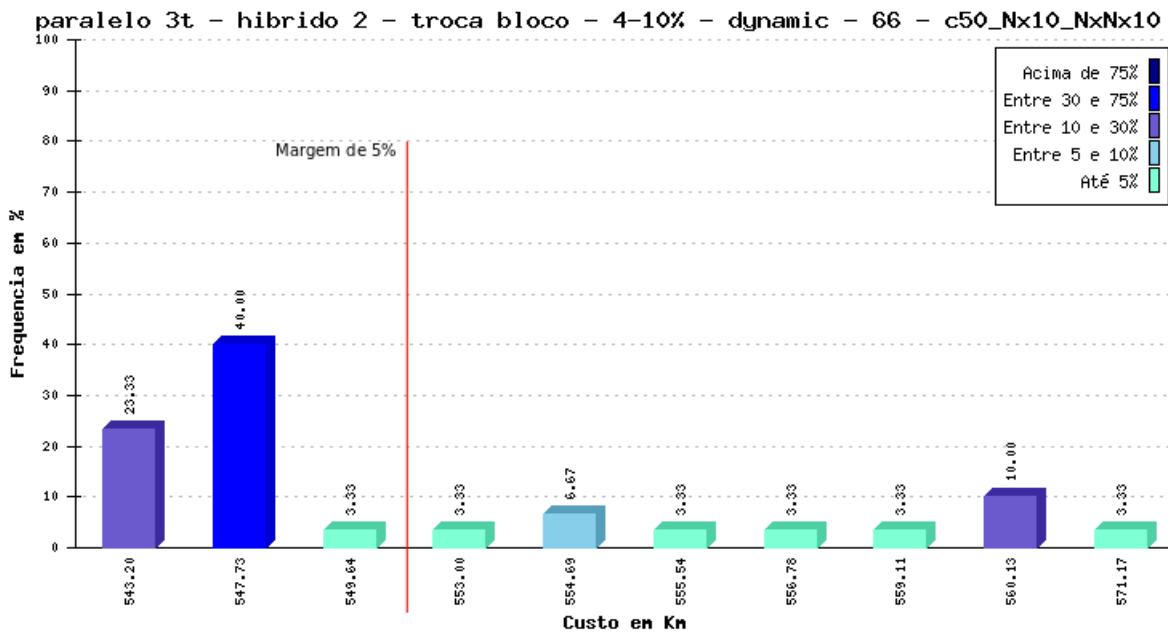
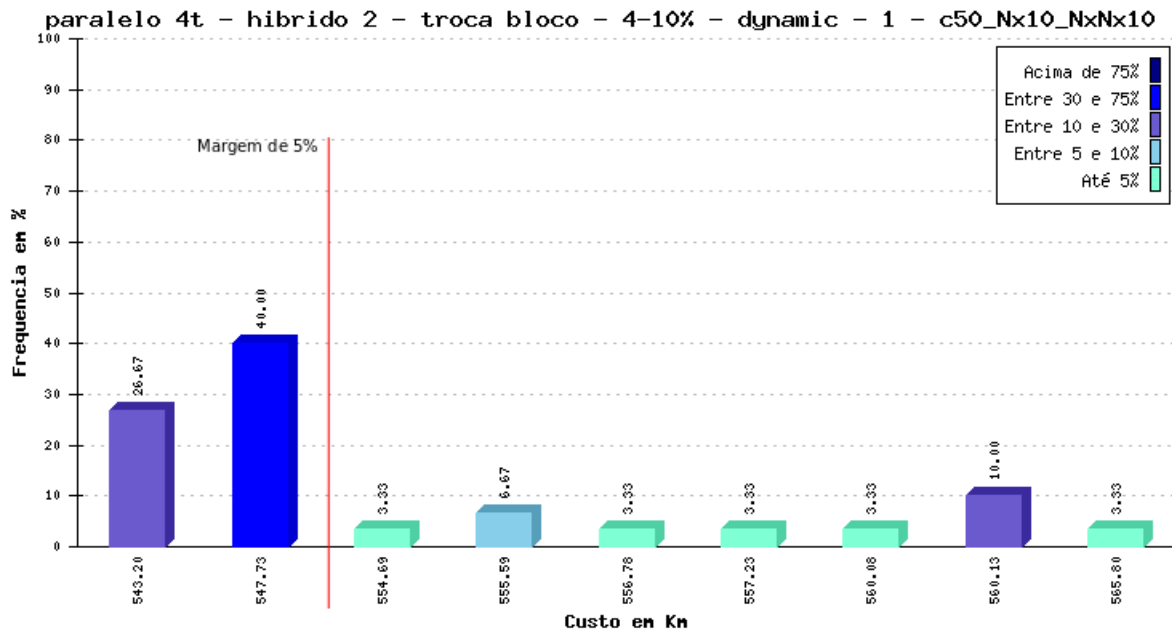


Figura 32 – Instância c50 submetida à versão com 3 threads, schedule dinâmico e chunk 33%: frequência dos custos encontrados.



A Figura 33 apresenta todos os custos obtidos com a instância c50 submetida a versão paralela com 4 *threads*, *schedule* dinâmico e granularidade fina. Nela podemos observar que 66,6% dos custos ficaram abaixo dos 550,84 Km. A melhor solução encontrada também foi idêntica a encontrada pela versão sequencial.

Figura 33 – Instância c50 submetida à versão com 4 *threads*, *schedule* dinâmico e *chunk* 1: frequência dos custos encontrados.



A Tabela 13 apresenta um comparativo entre as versões do programa que obtiveram os melhores *speedups* para instância c50. Nela percebemos que a porcentagem de custos dentro da margem de 5% em relação à melhor solução conhecida foi pior em todas as versões paralelas, quando comparadas à versão sequencial 2. O melhor custo encontrado foi igual em todas as versões e, os custos médios das versões paralelas foram parecidos, mas ligeiramente piores que o da versão sequencial 2. O pior custo encontrado que ficou mais próximo da melhor solução conhecida foi obtido pela versão sequencial 2. Percebemos que para instância c50, nenhuma das versões paralelas de melhor desempenho obteve soluções com qualidade superior às obtidas pela versão sequencial.

Tabela 13 – Comparativo entre as versões com melhor *speedup* para instância c50: porcentagem dentro da margem de 5%, melhor custo, custo médio, pior custo, tipo de escalonamento e tamanho do *chunk*.

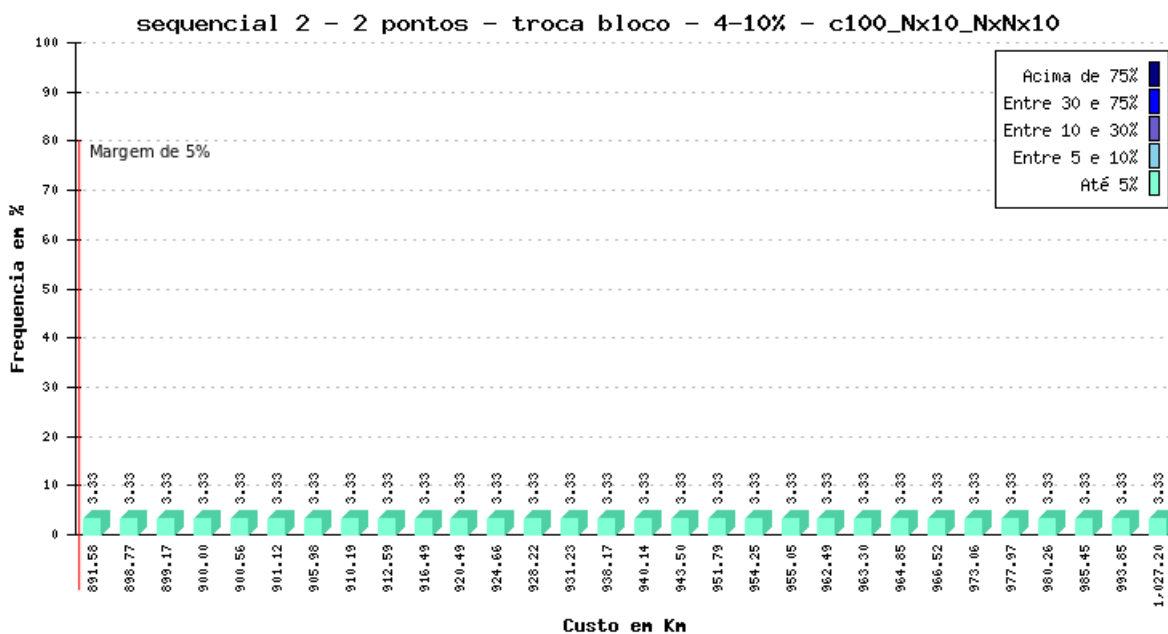
Versão	% dentro margem 5%	Melhor custo	Custo médio	Pior custo	Escalonamento	chunk
Sequencial 2	83,25%	543,19	547,78	560,12	-	-
2 <i>threads</i>	73,34%	543,19	549,99	565,80	dinâmico	25%
3 <i>threads</i>	66,6%	543,19	550,33	571,16	dinâmico	33%
4 <i>threads</i>	66,6%	543,19	550,15	565,80	dinâmico	1

4.5.2 Instância c100

Para a instância c100, que tem como melhor custo conhecido o valor de 826,14 Km, os custos encontrados em 30 execuções de cada versão do programa são apresentados a seguir.

A Figura 34 apresenta todos os custos obtidos com a instância c100 submetida à versão sequencial 2. Nela podemos observar que não há custos dentro de uma margem entre a melhor solução conhecida e 5% de tolerância, que seria entre 826,14 Km e 867,44 Km. A melhor solução encontrada é 65,43 Km maior que a melhor solução conhecida.

Figura 34 – Instância c100 submetida à versão sequencial 2: frequência dos custos encontrados.



A Figura 35 apresenta todos os custos obtidos com a instância c100 submetida à versão paralela com 2 *threads*, *schedule* dinâmico e granularidade grossa. Nela podemos observar que nenhum custo ficou abaixo dos 867,44 Km. A melhor solução encontrada é 75,11 Km maior que a melhor solução conhecida.

A Figura 36 apresenta todos os custos obtidos com a instância c100 submetida à versão paralela com 3 *threads*, *schedule* estático e granularidade fina. Nela podemos observar que nenhum dos custos ficou abaixo dos 867,44 Km. A melhor solução encontrada é 69,81 Km maior que a melhor solução conhecida.

A Figura 37 apresenta todos os custos obtidos com a instância c100 submetida à versão paralela com 4 *threads*, *schedule* estático e granularidade fina. Nela podemos observar que nenhum custo ficou abaixo dos 867,44 Km. A melhor solução encontrada é 71,02 Km maior que a melhor solução conhecida.

A Tabela 14 apresenta um comparativo entre as versões do programa que ob-

Figura 35 – Instância c100 submetida à versão com 2 threads, schedule estático e chunk 25%: frequência dos custos encontrados.

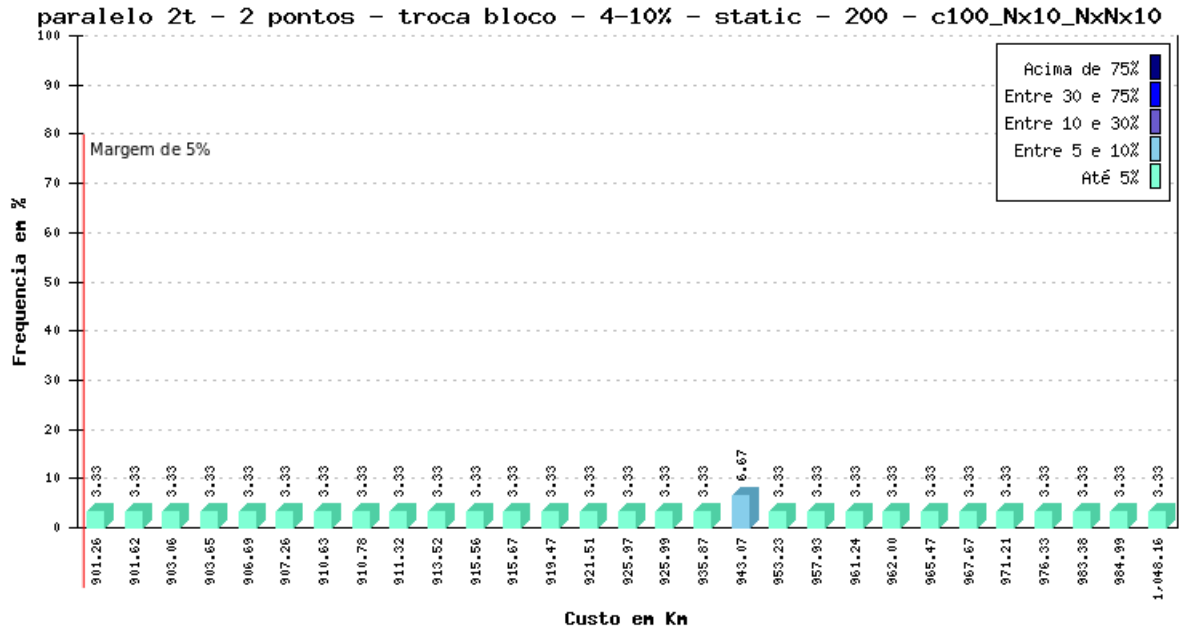


Figura 36 – Instância c100 submetida à versão com 3 threads, schedule estático e chunk 1: frequência dos custos encontrados.

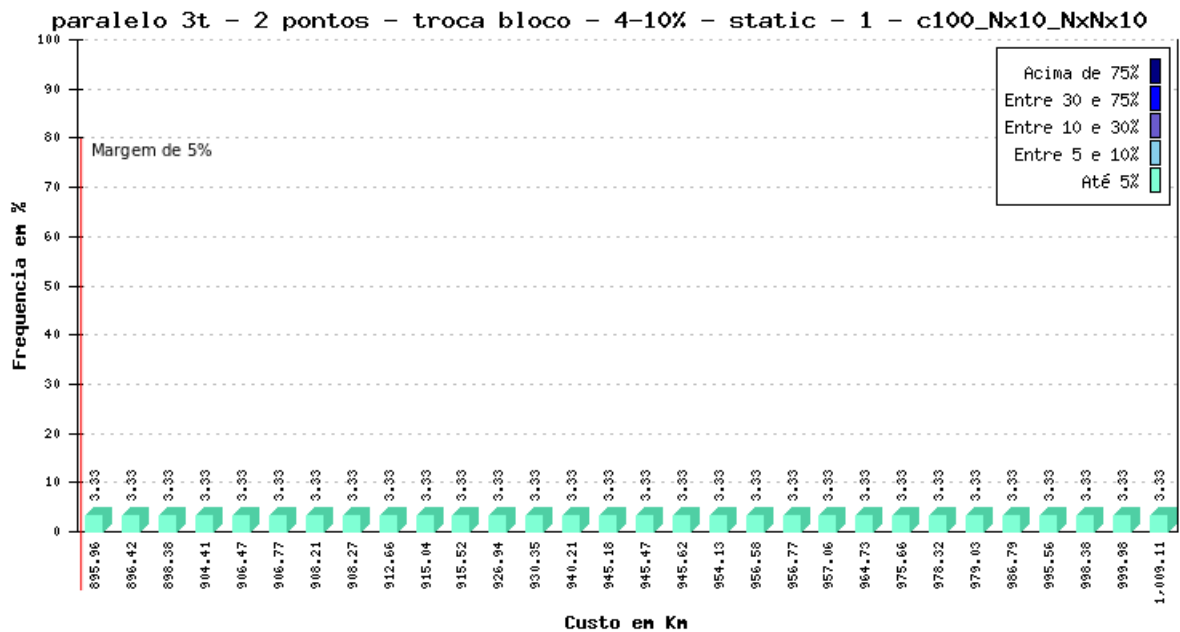
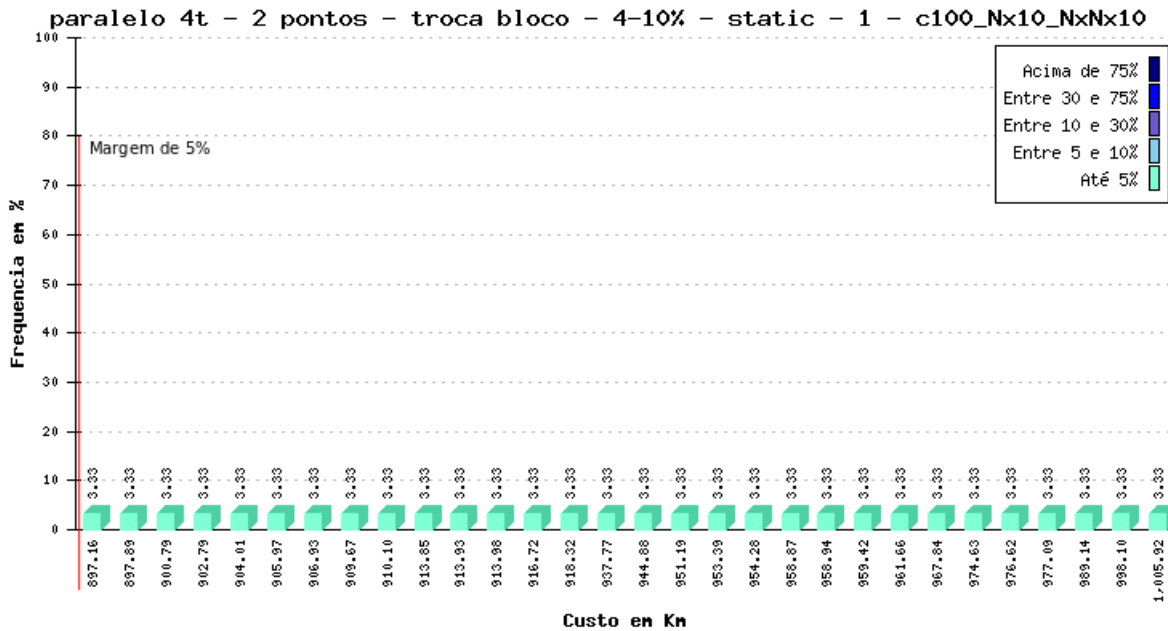


Figura 37 – Instância c100 submetida à versão com 4 *threads*, *schedule* estático e *chunk* 1: frequência dos custos encontrados.



tiveram os melhores *speedups* para instância c100. Percebemos que o melhor custo foi encontrado pela versão sequencial 2. Os custos médios em todas as versões foram parecidos, sendo que 2 versões paralelas obtiveram melhores resultados que a versão sequencial 2. O pior resultado encontrado por cada versão também foi melhor em 2 versões paralelas quando comparados a versão sequencial 2. Percebemos que a versão paralela com 2 *threads* obteve melhor média que a versão sequencial e que o pior custo encontrado pela versão paralela com 4 *threads* é melhor que o obtido pela versão sequencial.

Tabela 14 – Comparativo entre as versões com melhor *speedup* para instância c100: porcentagem dentro da margem de 5%, melhor custo, custo médio, pior custo, tipo de escalonamento e tamanho do *chunk*.

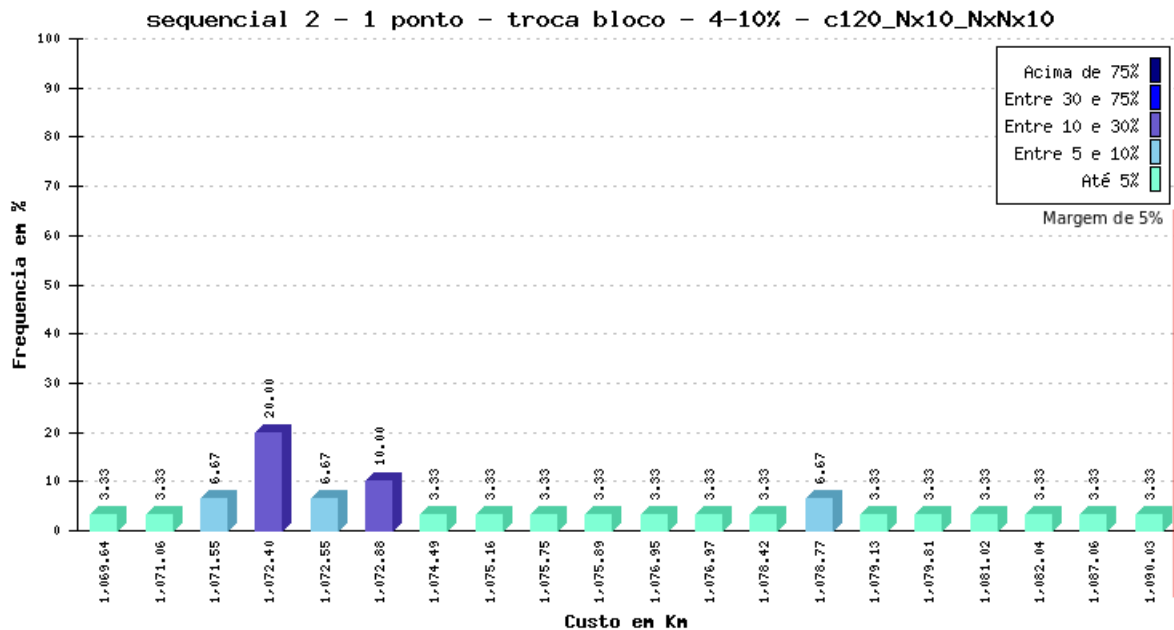
Versão	% dentro margem 5%	Melhor custo	Custo médio	Pior custo	Escalonamento	chunk
Sequencial 2	0%	891,57	940,62	1027,19	-	-
2 threads	0%	901,25	938,25	1048,16	estático	25%
3 threads	0%	895,95	943,80	1009,11	estático	1
4 threads	0%	897,16	939,39	1005,00	estático	1

4.5.3 Instância c120

Para a instância c120, que tem como melhor custo conhecido o valor de 1042,11 Km, os custos encontrados em 30 execuções de cada versão do programa são apresentados a seguir.

A Figura 38 apresenta todos os custos obtidos com a instância c120 submetida à versão sequencial 2. Nela podemos observar que considerando uma margem de 5% de tolerância, ou seja, entre 1042,11 Km e 1094,21 Km, 100% dos custos estão englobados. A melhor solução encontrada é apenas 27,53 Km maior que a melhor solução conhecida.

Figura 38 – Instância c120 submetida à versão sequencial 2: frequência dos custos encontrados.



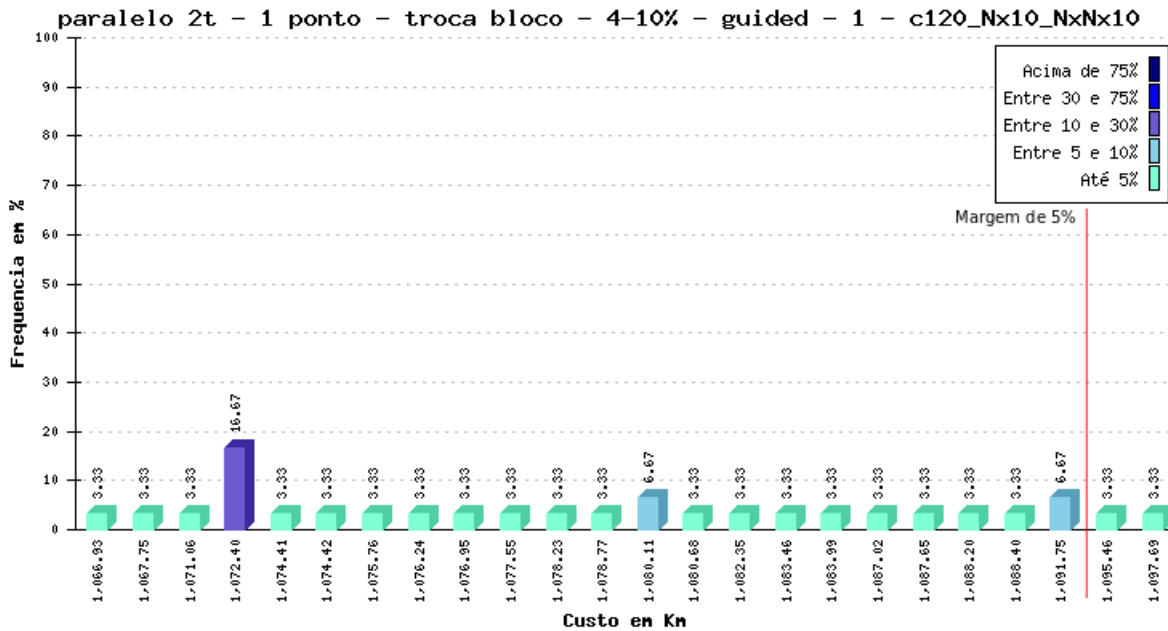
A Figura 39 apresenta todos os custos obtidos com a instância c120 submetida à versão paralela com 2 *threads*, *schedule* guiado e granularidade fina. Nela podemos observar que 93,24% dos custos ficaram abaixo dos 1094,21 Km. A melhor solução encontrada é melhor que a obtida pela versão sequencial, sendo 24,82 Km maior que a melhor solução conhecida.

A Figura 40 apresenta todos os custos obtidos com a instância c120 submetida à versão paralela com 3 *threads*, *schedule* estático e granularidade média. Nela podemos observar que 100% dos custos ficaram abaixo dos 1094,21 Km. A melhor solução encontrada também é melhor que a obtida pela versão sequencia, sendo 26,06 Km maior que a melhor solução conhecida.

A Figura 41 apresenta todos os custos obtidos com a instância c100 submetida à versão paralela com 4 *threads*, *schedule* estático e granularidade grossa. Nela podemos observar que 100% dos custos ficaram abaixo dos 1094,21 Km. A melhor solução encontrada também é menor que a obtida pela versão sequencial, sendo 26,56 Km maior que a melhor solução conhecida.

A Tabela 15 apresenta um comparativo entre as versões do programa que obtiveram os melhores *speedups* para instância c120. Nela percebemos que o melhor custo foi encontrado pela versão paralela com 2 *threads* e que as versões paralelas com 3 e 4

Figura 39 – Instância c120 submetida à versão com 2 threads, schedule guiado e chunk 1: frequência dos custos encontrados.



threads também obtiveram melhores custos que a versão sequencial. Os custos médios em todas as versões paralelas foram parecidos, sendo que o melhor foi encontrado pela versão sequencial 2. O pior resultado encontrado por cada versão também foi melhor na versão sequencial 2.

Tabela 15 – Comparativo entre as versões com melhor speedup para instância c120: porcentagem dentro da margem de 5%, melhor custo, custo médio, pior custo, tipo de escalonamento e tamanho do chunk.

Versão	% dentro margem 5%	Melhor custo	Custo médio	Pior custo	Escalonamento	chunk
Sequencial 2	100%	1069,63	1075,73	1090,02	-	-
2 threads	93,24%	1066,92	1079,95	1097,68	guiado	1
3 threads	100%	1068,18	1077,39	1090,40	estático	10%
4 threads	100%	1068,67	1078,01	1091,45	estático	25%

Como podemos perceber, ora a referência vem da versão sequencial, ora vem da versão paralela. Foi possível, através da paralelização, obter resultados compatíveis com os encontrados pela versão sequencial, num menor período de tempo. Isso significa que conseguimos manter a qualidade da solução, obtendo os resultados com menor tempo de execução. Em todos estes testes, as versões sequenciais e paralelas usavam o mesmo número de evoluções, de acordo com a instância e, este número era a condição parada do programa. A seguir, analisaremos os resultados obtidos com as versões paralelas executadas usando o tempo da versão sequencial como condição de parada.

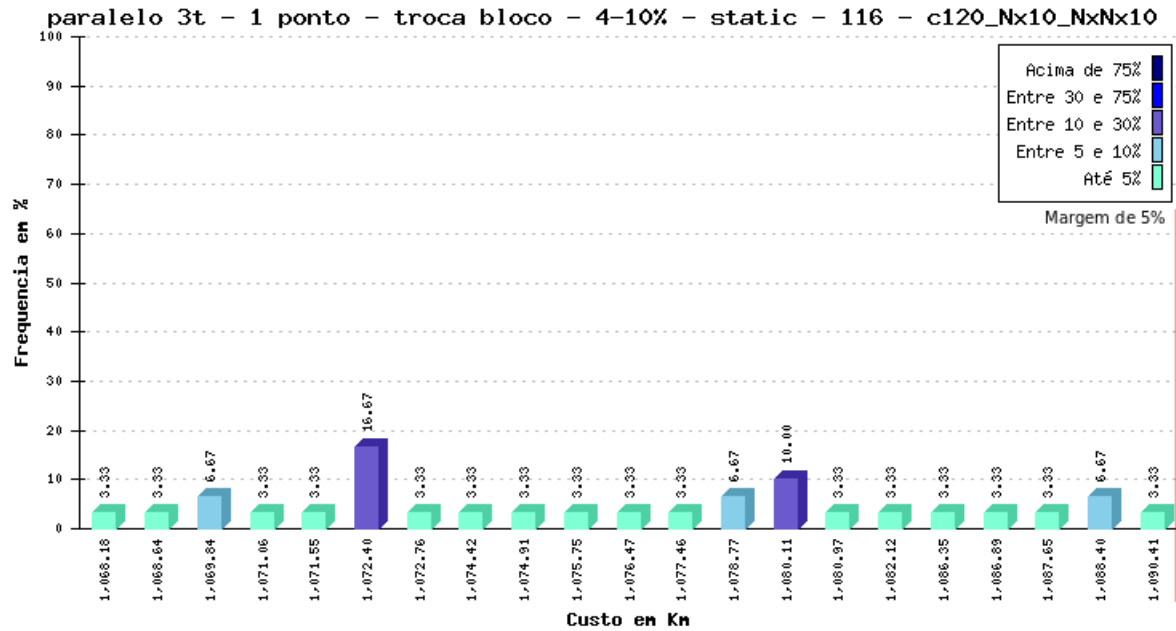


Figura 40 – Instância c120 submetida à versão com 3 threads, schedule estático e chunk 10%: frequência dos custos encontrados.

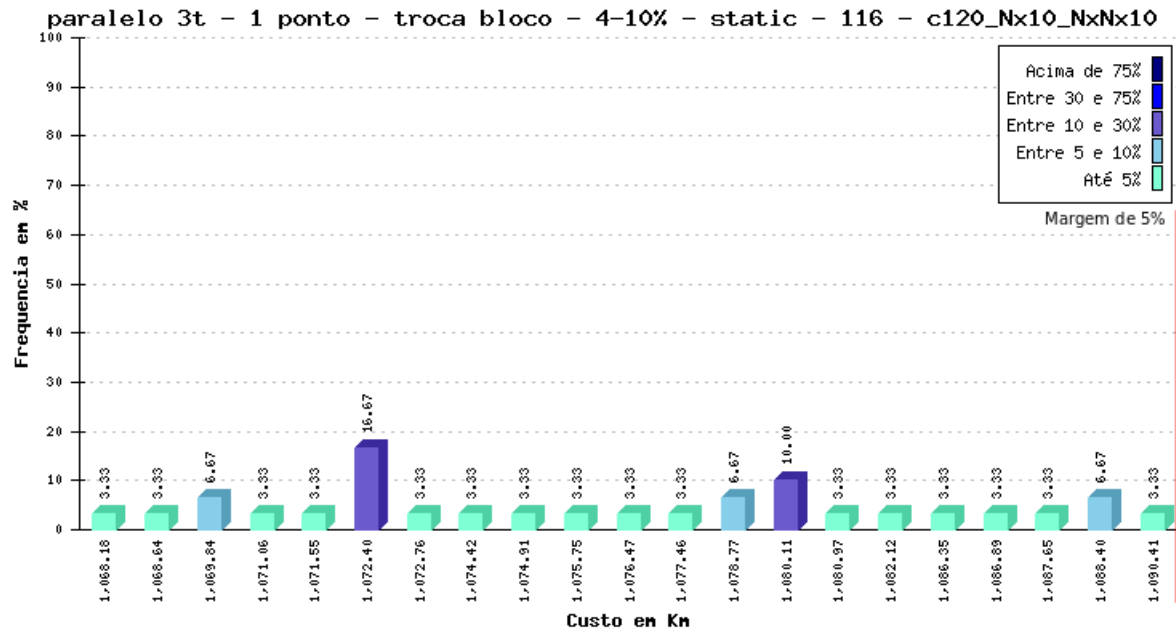
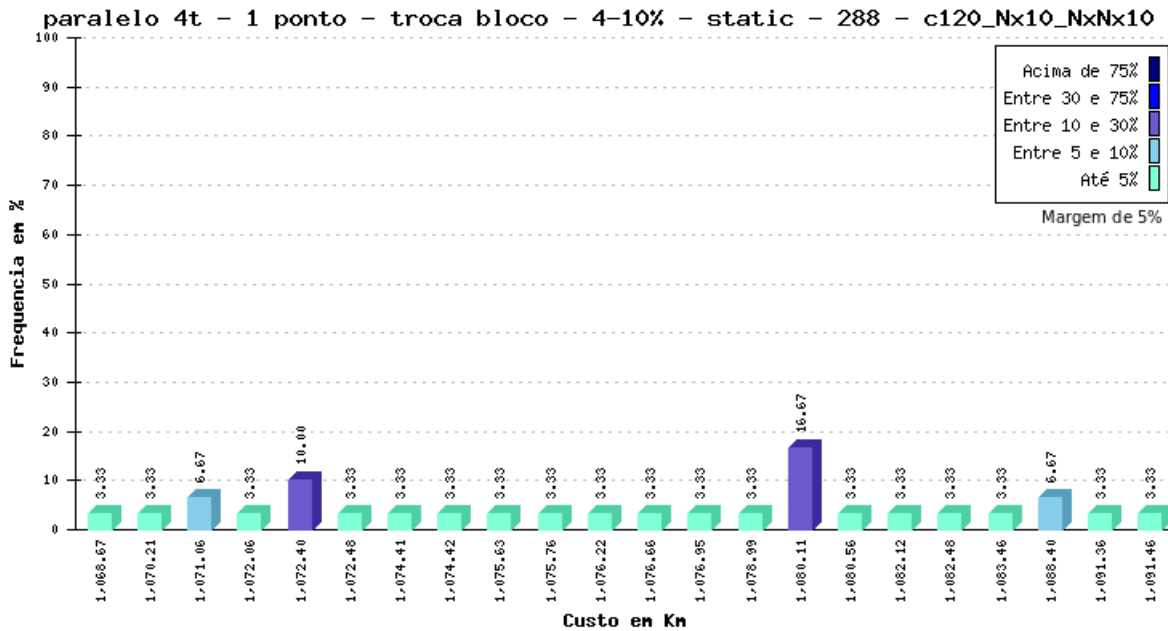


Figura 41 – Instância c120 submetida à versão com 4 *threads*, *schedule* estático e *chunk* 25%: frequência dos custos encontrados.



4.6 Executando a versão Paralela pelo tempo da Sequencial

Nesta seção verificaremos a qualidade das soluções encontradas executando as versões paralelas pelo mesmo período de tempo da versão sequencial 2. Espera-se obter melhores resultados porque aumentamos o número de evoluções executadas em um mesmo intervalo de tempo graças à paralelização.

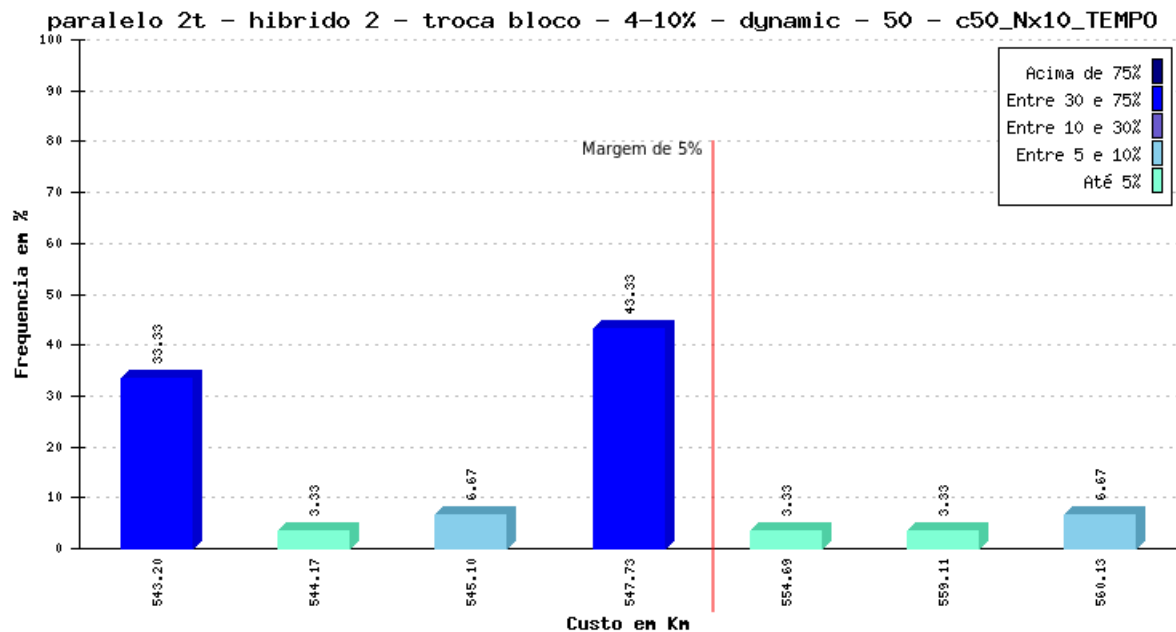
4.6.1 Instância c50

Para a instância c50, os custos encontrados em 30 execuções de cada versão paralela do programa, executadas pelo tempo da versão sequencial 2, são apresentados a seguir.

A Figura 42 apresenta todos os custos obtidos com a instância c50 submetida à versão paralela com 2 *threads*, *schedule* dinâmico e granularidade grossa, executada pelo tempo sequencial. Nela podemos observar que 86,58% dos custos ficaram abaixo dos 550,84 Km. Esta porcentagem de custos dentro da margem de 5% foi melhor que a encontrada pela versão sequencial 2 e pela versão paralela com 2 *threads*. O melhor custo, embora idêntico ao encontrado pela versão sequencial 2, foi encontrado com maior frequência. O pior custo encontrado foi idêntico ao encontrado pela versão sequencial 2. A média de todos os custos foi um pouco menor que a encontrada pelas versões paralela com 2 *threads* e sequencial 2, o que pode ser conferido na Tabela 16.

A Figura 43 apresenta todos os custos obtidos com a instância c50 submetida à versão paralela com 3 *threads*, *schedule* dinâmico e granularidade grossa, executada pelo

Figura 42 – Instância c50 submetida à versão com 2 *threads* executadas pelo mesmo tempo da versão sequencial, *schedule* dinâmico e *chunk* 25%: frequência dos custos encontrados.



tempo sequencial. Nela podemos observar que os custos abaixo dos 550,84 Km foram 86,58%. A porcentagem de custos dentro da margem de 5% foi melhor que a encontrada pela versão sequencial 2. O melhor custo encontrado foi o mesmo encontrado na versão sequencial 2, mas com maior frequência. O pior custo encontrado pela versão com *schedule* dinâmico e granularidade grossa foi idêntico ao encontrado pela versão sequencial 2. A média dos custos foi melhor que a encontrada pela versão sequencial 2, o que pode ser conferido na Tabela 16.

A Figura 44 apresenta todos os custos obtidos com a instância c50 submetida à versão paralela com 4 *threads*, *schedule* dinâmico e granularidade fina, executada pelo tempo sequencial. Nela podemos observar que 93,24% dos custos ficaram abaixo dos 550,84 Km. O melhor custo encontrado foi idêntico ao encontrado pela versão sequencial 2, mas com maior frequência. Já o pior custo encontrado foi melhor nesta versão paralela. A média de todos os custos também foi melhor na versão paralela com 4 *threads* executada pelo tempo sequencial. Mais detalhes podem ser vistos na Tabela 16.

Um comparativo entre a versão sequencial 2 e as versões paralelas que obtiveram os melhores *speedups* para a instância c50, quando executadas pelo tempo sequencial é apresentado na Tabela 16. Percebemos que o melhor custo encontrado foi igual em todas as versões e, é apenas 18,58 Km maior que o menor custo conhecido. Os custos médios das versões paralelas foram parecidos, mas ligeiramente melhores que o da versão sequencial 2, sendo que o melhor custo médio foi encontrado pela versão paralela com 4 *threads*. O pior resultado encontrado por cada versão também foi melhor na versão paralela com 4

Figura 43 – Instância c50 submetida à versão com 3 *threads* executadas pelo mesmo tempo da versão sequencial, *schedule* dinâmico e *chunk* 33%: frequência dos custos encontrados.

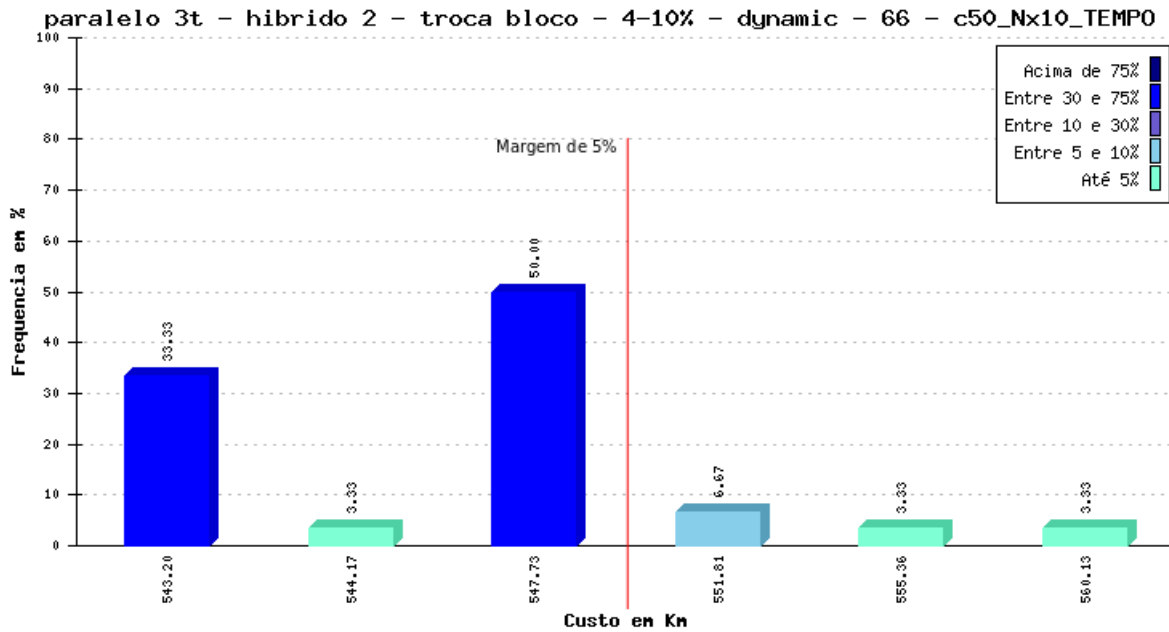
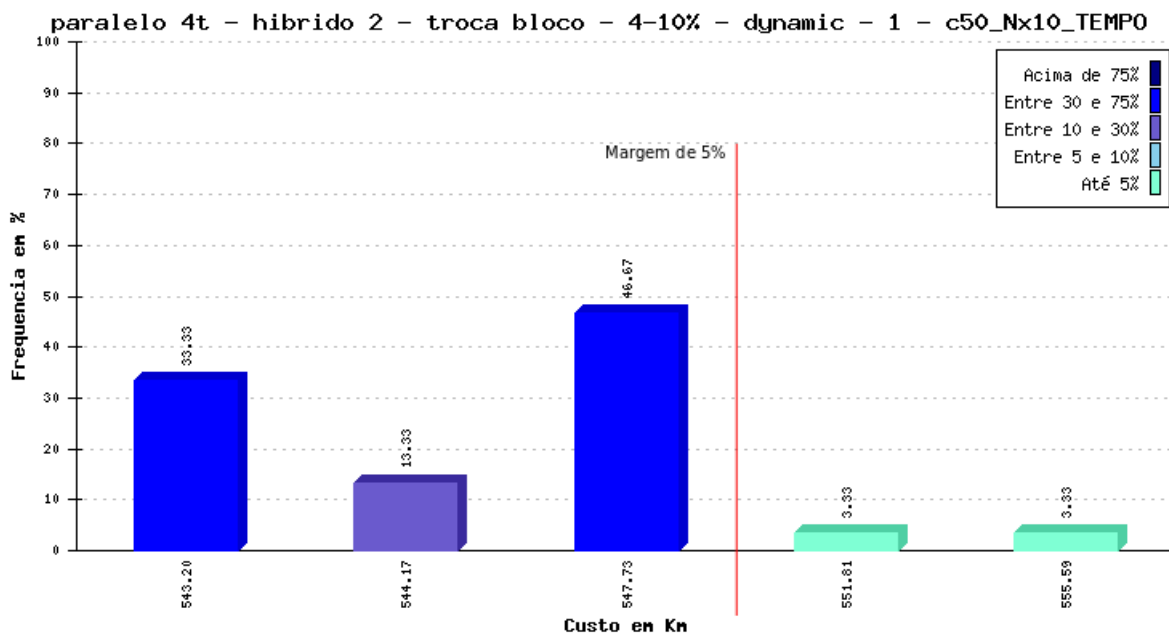


Figura 44 – Instância c50 submetida à versão com 3 *threads* executadas pelo mesmo tempo da versão sequencial, *schedule* dinâmico e *chunk* 1: frequência dos custos encontrados.



threads. Como esperado, as versões com maior número de evoluções obtiveram melhores resultados, embora a diferença tenha sido pequena, para a instância c50.

Usando 4 *threads* foi possível executar um número de evoluções até 72,15% maior que a versão sequencial 2, no mesmo período de tempo. Não foi possível executar mais evoluções porque o tempo de computação dessa instância é muito pequeno, o que prejudica inclusive o ganho que o paralelismo poderia dar a aplicação. A diferença de tempo entre as versões sequencial e a paralela com 4 *threads* foi de apenas 3,18 s e, continuar executando a versão paralela por mais esse período de tempo permitiu que o número de evoluções saltasse de 12500 para 21519. Os ganhos poderiam ser maiores se mais evoluções fossem executadas, porém, o tempo de execução sequencial muito pequeno é que não permitiu. Vale ressaltar que a média dos custos encontrados por essa versão ficou apenas 4,1% maior que a melhor solução conhecida, o que indica que estamos bem perto da solução ótima.

Tabela 16 – Comparativo entre as versões com melhor *speedup* para instância c50, executadas pelo tempo sequencial: porcentagem dentro da margem de 5%, melhor custo, custo médio, pior custo, tipo de escalonamento, tamanho do *chunk* e número de evoluções.

<i>Versão</i>	% dentro margem 5%	<i>Melhor custo</i>	<i>Custo médio</i>	<i>Pior custo</i>	<i>schedule</i>	<i>chunk</i>	<i>Número médio de evoluções</i>
Sequencial 2	83,25%	543,19	547,78	560,12	-	-	12500
2 <i>threads</i>	86,58%	543,19	547,36	560,12	dinâmico	25%	14916
3 <i>threads</i>	86,58%	543,19	547,04	560,12	dinâmico	33%	19588
4 <i>threads</i>	93,24%	543,19	546,14	555,59	dinâmico	1	21519

4.6.2 Instância c100

Para a instância c100, os custos encontrados em 30 execuções de cada versão paralela do programa, executadas pelo tempo da versão sequencial 2, são apresentados a seguir.

A Figura 45 apresenta todos os custos obtidos com a instância c100 submetida à versão paralela com 2 *threads*, *schedule* dinâmico e *chunk* 25%, executada pelo tempo sequencial. O melhor custo encontrado foi 16,41 Km menor que o encontrado pela versão sequencial 2 e é 49,02 Km maior que a melhor solução conhecida. O pior custo e a média de todos os custos também foram melhores nesta versão paralela, o que pode ser conferido na Tabela 17.

A Figura 46 apresenta todos os custos obtidos com a instância c100 submetida à versão paralelas com 3 *threads*, *schedule* estático e granularidade fina, executadas pelo tempo sequencial. O melhor custo encontrado foi 19,32 Km menor que o encontrado pela versão sequencial, sendo 46,11 Km maior que a melhor solução conhecida. O pior custo encontrado foi melhor que o encontrado na versão sequencial, o que pode ser conferido na Tabela 17.

Figura 45 – Instância c100 submetida à versão com 2 threads executadas pelo tempo sequencial, *schedule* estático e *chunk* 25%: frequência dos custos encontrados.

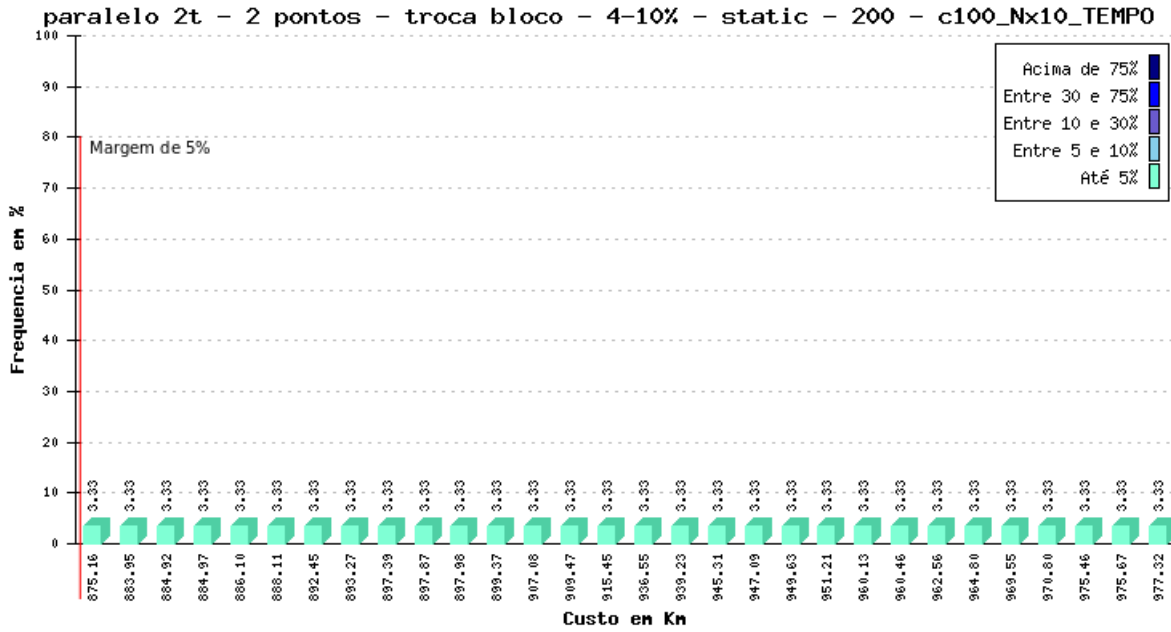
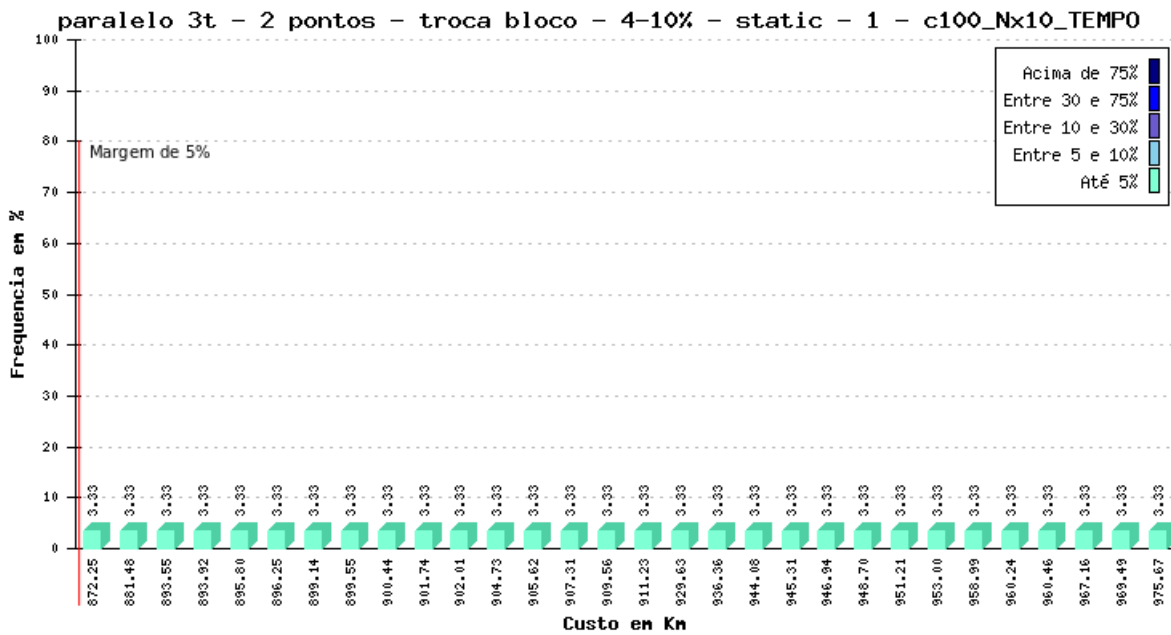
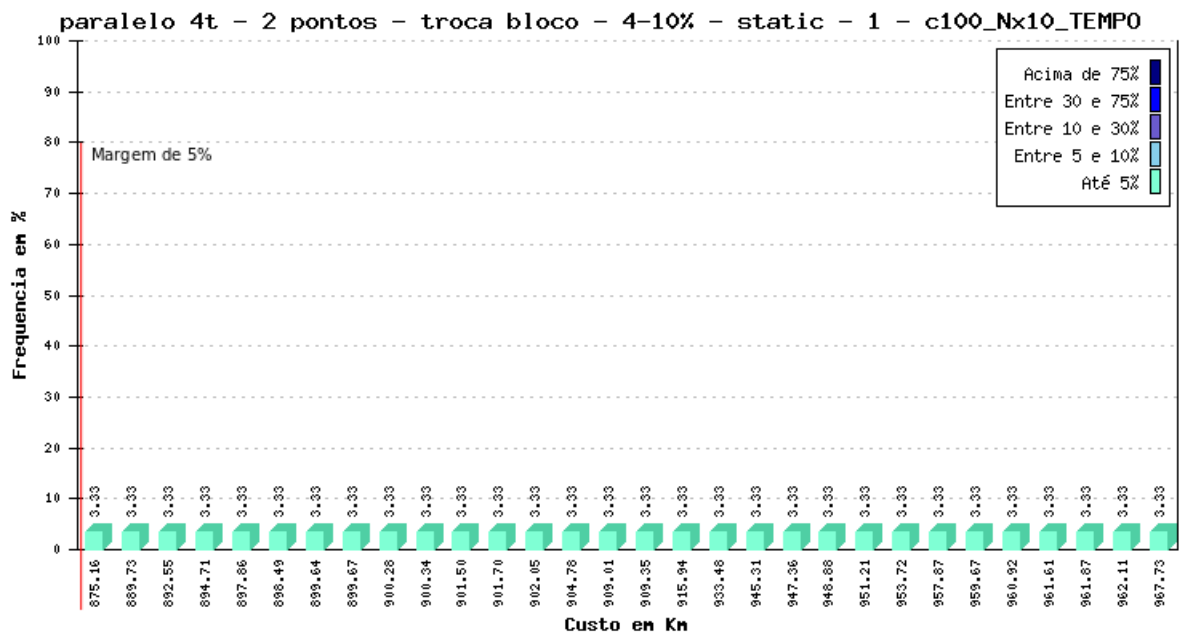


Figura 46 – Instância c100 submetida à versão com 3 threads executadas pelo tempo sequencial, *schedule* estático e *chunk* 1: frequência dos custos encontrados.



A Figura 47 apresenta todos os custos obtidos com a instância c100 submetida à versão paralela com 4 *threads*, *schedule* estático e granularidade fina, executada pelo tempo sequencial. O melhor custo encontrado foi 16,41 Km menor que o encontrado pela versão sequencial 2, sendo que ele é 49,02 Km maior que a melhor solução conhecida. O pior custo encontrado e a média dos custos também foram melhores na versão paralela com 4 *threads* executada pelo tempo sequencial. Mais detalhes podem ser vistos na Tabela 17.

Figura 47 – Instância c100 submetida à versão com 4 *threads* executadas pelo tempo sequencial, *schedule* estático e *chunk* 1: frequência dos custos encontrados.



Um comparativo entre a versão sequencial 2 e as versões paralelas que obtiveram os melhores *speedups* para a instância c100, quando executadas pelo tempo sequencial é apresentado na Tabela 17. Percebemos que o melhor custo foi encontrado pela versão paralela com 3 *threads*, *schedule* estático e granularidade fina. Os custos médios foram melhores em todas as versões paralelas, sendo que o melhor custo médio foi obtido pela versão paralela com 4 *threads*. O pior resultado encontrado por cada versão também foi melhor em todas as versões paralelas quando comparados a versão sequencial 2. Como esperado, os melhores resultados foram obtidos por versões com número de evoluções maior que o número fixo das versões anteriores.

Para a instância c100, usando 4 *threads* foi possível executar um número de evoluções até 111,81% maior que a versão sequencial 2, no mesmo período de tempo. Isso porque esta instância possui tempo de computação maior e a paralelização conseguiu um *speedup* melhor que o obtido pela instância c50. A diferença de tempo entre as versões sequencial e a paralela com 4 *threads* foi de 289,12 s e, continuar executando a versão paralela por mais esse período de tempo permitiu que o número de evoluções saltasse de

100000 para 211817. Os ganhos foram mais expressivos para essa instância, pois foi possível diminuir a distância entre o melhor resultado encontrado e a melhor solução conhecida em 19,32 Km. A diferença entre o menor e o maior custos encontrados também diminuiu nesta versão. Para esta instância, ainda há margem para maiores ganhos, já que a média das soluções encontradas é 11,78% maior que a melhor solução conhecida.

Tabela 17 – Comparativo entre as versões com melhor *speedup* para instância c100: porcentagem dentro da margem de 5%, melhor custo, custo médio, pior custo, tipo de escalonamento, tamanho do *chunk* e número de evoluções.

<i>Versão</i>	% dentro margem 5%	<i>Melhor custo</i>	<i>Custo médio</i>	<i>Pior custo</i>	<i>schedule</i>	<i>chunk</i>	<i>Número médio de evoluções</i>
Sequencial 2	0%	891,57	940,62	1027,19	-	-	100000
2 <i>threads</i>	0%	875,16	926,64	977,31	estático	25%	156511
3 <i>threads</i>	0%	872,25	924,06	975,67	estático	1	198447
4 <i>threads</i>	0%	875,16	923,48	967,72	estático	1	211817

4.6.3 Instância c120

Para a instância c120, os custos encontrados em 30 execuções de cada versão paralela do programa, executadas pelo tempo da versão sequencial 2, são apresentados a seguir.

A Figura 48 apresenta todos os custos obtidos com a instância c120 submetida à versão paralela com 2 *threads*, *schedule* guiado e granularidade fina, executada pelo tempo sequencial. Nela podemos observar que 100% dos custos ficaram abaixo dos 1094,21 Km. O melhor e o pior custo encontrados foram melhores na versão paralela, sendo o melhor custo 23,16 Km maior que a melhor solução conhecida. A média de todos os custos foi melhor na versão sequencial 2, o que pode ser conferido na Tabela 18.

A Figura 49 apresenta todos os custos obtidos com a instância c120 submetida à versão paralela com 3 *threads*, *schedule* estático e granularidade fina. Nela podemos observar que 100% dos custos ficaram abaixo dos 1094,21 Km. O melhor e o pior custo encontrados foram melhores na versão paralela, sendo o melhor custo 23,16 Km maior que a melhor solução conhecida. A média de todos os custos também foi melhor na versão paralela, o que pode ser conferido na Tabela 18.

A Figura 50 apresenta todos os custos obtidos com a instância c100 submetida à versão paralela com 4 *threads*, *schedule* estático e granularidade grossa. Nela podemos observar que 100% dos custos ficaram abaixo dos 1094,21 Km. O melhor e o pior custos encontrados foram melhores na versão paralela, sendo o melhor custo 23,16 Km maior que a melhor solução conhecida. A média de todos os custos também foi melhor na versão paralela, o que pode ser conferido na Tabela 18.

Figura 48 – Instância c120 submetida à versão com 2 threads executadas pelo tempo sequencial, *schedule* guiado e *chunk* 1: frequência dos custos encontrados.

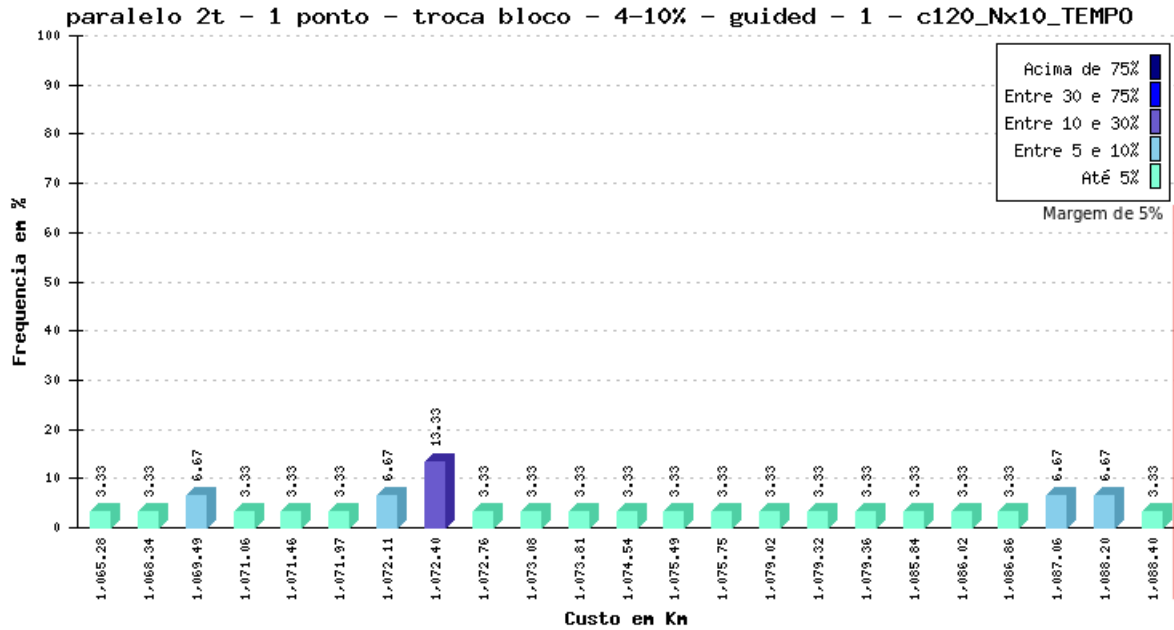


Figura 49 – Instância c120 submetida à versão com 3 threads executadas pelo tempo sequencial, *schedule* estático e *chunk* 10%: frequência dos custos encontrados.

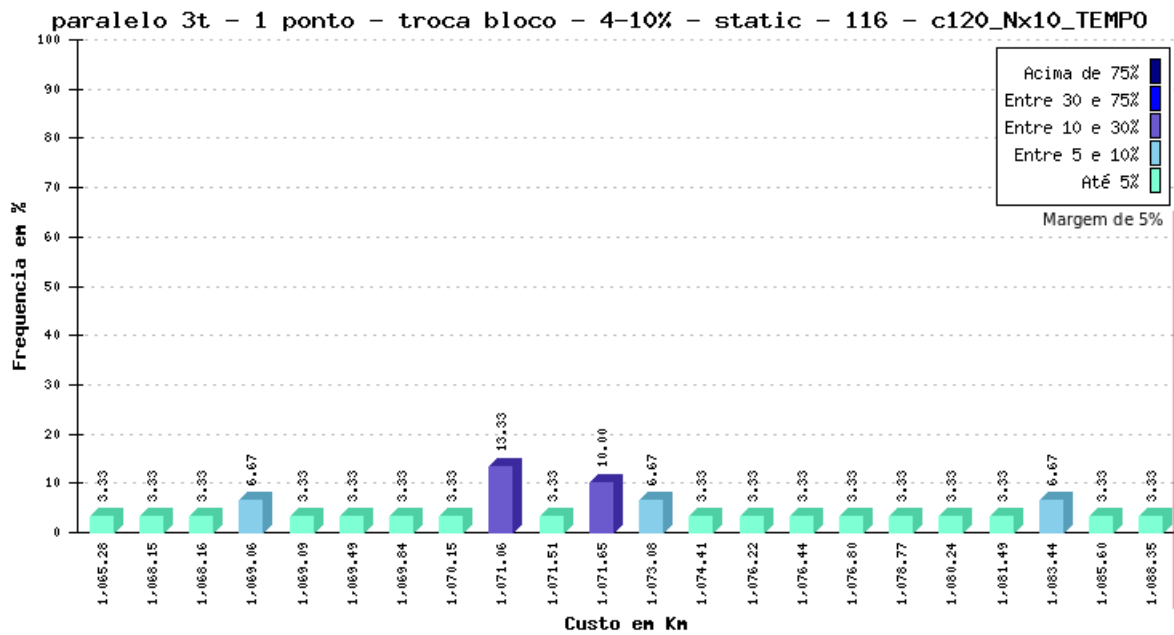
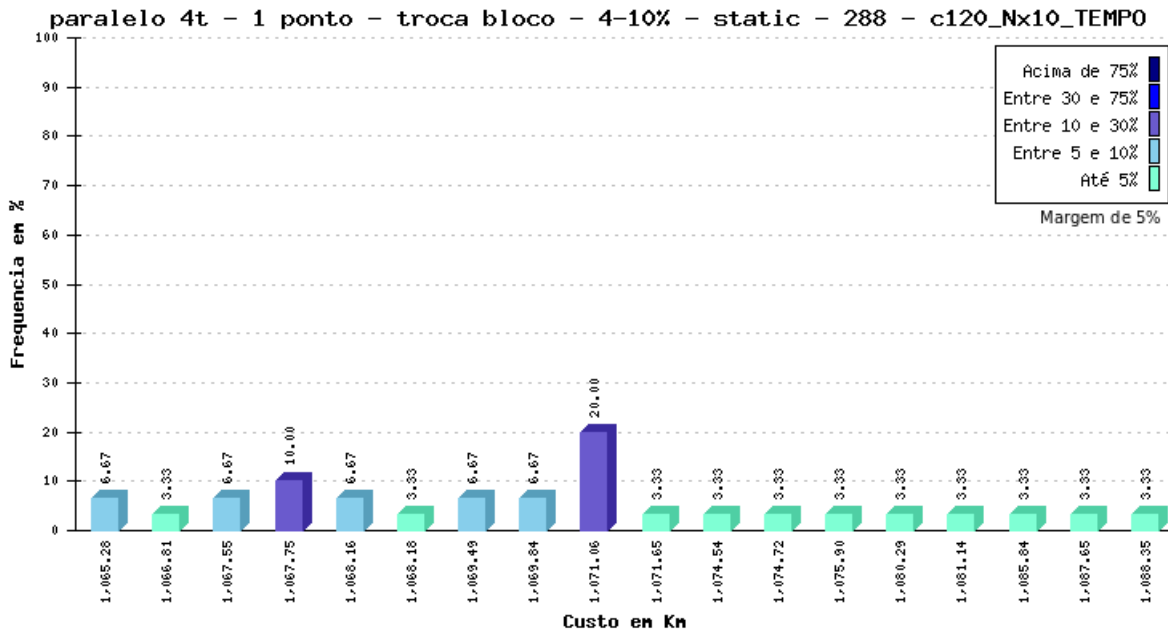


Figura 50 – Instância c120 submetida à versão com 4 *threads* executadas pelo tempo sequencial, *schedule* estático e *chunk* 25%: frequência dos custos encontrados.



Um comparativo entre a versão sequencial 2 e as versões paralelas que obtiveram os melhores *speedups* para a instância c120, quando executadas pelo tempo sequencial é apresentado na Tabela 18. Percebemos também que o melhor custo encontrado por todas as versões paralelas foi igual e, apenas 23,16 Km maior que a melhor solução conhecida. Os custos médios nas versões com 3 e 4 *threads* foram melhores que o encontrado na versão sequencial 2. O pior resultado encontrado por cada versão paralela também foi melhor que o encontrado na versão sequencial 2. Como esperado, as versões com número de evoluções maior que o número fixo das versões anteriores obtiveram melhores resultados.

Para instância c120, usando 4 *threads* foi possível executar um número de evoluções 135,32% maior que a versão sequencial 2, no mesmo período de tempo. Das instâncias testadas, essa é a que possui maior tempo de computação e, a paralelização conseguiu um *speedup* mais alto. A diferença de tempo entre as versões sequencial e a paralela com 4 *threads* foi de 927,89 s e, continuar executando a versão paralela por mais esse período de tempo permitiu que o número de evoluções saltasse de 172800 para 406643. Houve ganho com a execução pelo tempo sequencial, embora menos expressivos que os obtidos pela instância c100. O custo médio ficou apenas 2,88% maior que a menor solução conhecida, o que indica que estamos muito perto da solução ótima.

Executando as versões paralelas pelo tempo da versão sequencial, foi possível melhorar os resultados obtidos pelas versões anteriores. Através da paralelização, foi possível executar mais evoluções, avaliando mais indivíduos e, conseqüentemente aumentando as chances de se encontrar melhores resultados.

Tabela 18 – Comparativo entre as versões com melhor *speedup* para instância c120: porcentagem dentro da margem de 5%, melhor custo, custo médio, pior custo, tipo de escalonamento, tamanho do *chunk* e número de evoluções.

<i>Versão</i>	% dentro margem 5%	<i>Melhor custo</i>	<i>Custo médio</i>	<i>Pior custo</i>	<i>schedule</i>	<i>chunk</i>	<i>Número médio de evoluções</i>
Sequencial 2	100%	1069,63	1075,73	1090,02	-	-	172800
2 <i>threads</i>	100%	1065,27	1076,72	1088,40	guiado	1	281991
3 <i>threads</i>	100%	1065,27	1074,01	1088,35	estático	10%	361782
4 <i>threads</i>	100%	1065,27	1072,17	1088,35	estático	25%	406643

4.7 Considerações sobre a Paralelização do Algoritmo Genético

Neste capítulo, foi feita uma análise da versão sequencial, passando pelas funções implementadas e culminando com a geração do perfil da execução. A partir desta análise foi possível propor um novo mecanismo que acelerou a versão sequencial fazendo com que a mesma passasse a ser executada em aproximadamente 70% do tempo da versão anterior. Foi feita uma análise em busca de trechos de código que pudessem ser paralelizados e, foi verificado que as funções que são responsáveis por mais de 88% do tempo total de execução, recebem chamadas na função `Nova geração`. Explorando esta função, foram definidas duas regiões paralelas: a primeira, responsável pelo trecho que seleciona 10% dos melhores indivíduos e 10% de indivíduos aleatórios para serem perpetuados; a segunda, responsável pelo laço que faz os cruzamentos para completar a nova geração.

Foram feitos testes com número de *threads* entre 2 e 8 onde ficou evidenciado que a partir de 4 a eficiência é muito baixa. Foram feitos testes para verificar qual a melhor forma de escalonamento e qual a melhor divisão dos blocos entre as iterações. Ficou claro que para essa aplicação, a melhor forma de escalonamento depende da instância de entrada e do tamanho do bloco distribuído.

Para a instância c50, o melhor desempenho foi alcançado utilizando *schedule* dinâmico. A granularidade para 2, 3 e 4 *threads* foi grossa, grossa e fina, respectivamente. Para a instância c100, o melhor desempenho foi alcançado utilizando *schedule* estático. A granularidade para 2, 3 e 4 *threads* foi grossa, fina e fina, respectivamente. Para a instância c120, o melhor desempenho foi alcançado utilizando *schedule* guiado para 2 *threads* e estático para 3 e 4 *threads*. A granularidade foi fina, média e grossa, respectivamente.

A qualidade das soluções das versões que obtiveram melhor desempenho foram avaliadas. Percebeu-se que as versões paralelas obtiveram resultados compatíveis com os obtidos pela versão sequencial, mas em um período de tempo menor. Essas mesmas versões foram então executadas pelo tempo da versão sequencial, onde obteve-se resultados melhores que aqueles obtidos pela versão sequencial. Executando uma versão paralela pelo tempo de uma versão sequencial, é possível executar mais evoluções e, por consequência, avaliar mais indivíduos, aumentando as chances de se obter menores custos.

Para as instâncias c50, c100 e c120, foi possível executar até 72,15%, 111,81% e 135,32% mais evoluções, respectivamente. Houve ganho com a execução pelo tempo sequencial, o que refletiu em melhores resultados. Para as mesmas instâncias, os melhores custos encontrados foram 18,58 Km, 46,11 Km e 23,16 Km maiores que as melhores soluções conhecidas, também respectivamente.

O próximo capítulo apresenta a conclusão deste trabalho.

5 Conclusão

Este trabalho apresentou a implementação e paralelização de um **AG** aplicado ao **PRV**, buscando melhores resultados que os obtidos pela versão sequencial. Inicialmente foi implementado um **AG** e, para sua definição, houve uma investigação sobre os diversos parâmetros com intuito de eleger a combinação que encontre os melhores resultados mais vezes. Durante essa investigação, ficou claro que os melhores resultados eram obtidos com populações maiores e com maior número de evoluções, o que implica em um maior tempo de computação.

Para chegarmos à versão paralela do **AG**, o código sequencial foi investigado, a fim de verificar quais trechos deveriam receber esforços de paralelização. Esta investigação foi realizada com o auxílio da ferramenta **gprof**, que gera um perfil da execução e permite analisar quais funções consomem mais recursos. A paralelização deu-se utilizando técnicas de programação para memória compartilhada, utilizando a **API** OpenMP. Foram utilizadas as diretivas de paralelismo em laços **for** e diretivas de paralelismo não iterativo, com uso de *sections*.

Foram usadas 3 instâncias para testar as versões implementadas do **AG**: c50 com 50 cidades, c100 com 100 cidades e c120 com 120 cidades. Os testes com o **AG** paralelo demonstraram que houve ganho no tempo de execução, mas o *speedup* e a eficiência usando os valores padrão para *schedule* e *chunk* ficaram bem abaixo do esperado, principalmente usando mais de 4 *threads*. Foram feitos então, testes com 2, 3 e 4 *threads*, usando diferentes tipos de escalonamento e diferentes granularidades, buscando combinações que obtivessem maiores *speedups*. Para as 3 instâncias testadas, foi possível executar o **AG** em menor tempo, mantendo o resultado compatível com o encontrado pela versão sequencial. Uma vez identificadas essas versões, elas foram executadas pelo tempo de duração da execução sequencial, com o objetivo de executar o maior número de evoluções possível e, com isso, obter menores custos que os obtidos pela versão sequencial.

Usando a versão paralela, para a instância c50 foi possível executar um número de evoluções até 72,15% maior que o avaliado pela versão sequencial no mesmo período de tempo. Embora a melhor solução encontrada tenha sido igual para essa instância ela foi encontrada com maior frequência e, a média de todos os custos encontrados acabou sendo menor. A melhor solução ficou apenas 18,58 Km maior que a melhor solução conhecida para essa instância. Para a instância c100 foi possível executar um número de evoluções até 111,81% maior que o avaliado pela versão sequencial. A média dos custos foi menor que a encontrada pela versão sequencial. O menor custo encontrado foi 46,11 Km maior

que a melhor solução conhecida para essa instância e 19,32 Km menor que o melhor custo encontrado pela versão sequencial. Para a instância c120, foi possível executar um número de evoluções até 135,32% maior que o número de evoluções da versão sequencial. A média dos custos foi mais baixa e o melhor resultado encontrado foi apenas 23,16 Km maior que a melhor solução conhecida para essa instância.

De modo geral, executar a versão paralela pelo tempo de execução da versão sequencial possibilitou que mais evoluções ocorressem e, por consequência, que mais indivíduos fossem avaliados, aumentando as chances de se encontrar melhores soluções. Para todas as instâncias testadas, houve melhora nos resultados, o que indica que o objetivo do trabalho foi atingido.

Pretende-se, em trabalhos futuros, explorar outras APIs de paralelização e usar alguma técnica mais elaborada para semear a população inicial, como o uso de coordenadas polares. Também pretende-se melhorar o mecanismo de seleção, procurando evitar vários sorteios consecutivos para o mesmo cruzamento.

Referências

- 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. Citado na página 34.
- BAKER, B. M.; AYECHIEW, M. A genetic algorithm for the vehicle routing problem. *Computers & Operations Research*, Coventry, UK, v. 30, n. 5, p. 787–800, apr 2003. Citado na página 41.
- BALDACCI, R.; TOTH, P.; VIGO, D. Recent advances in vehicle routing exact algorithms. *4OR*, Berlin, Germany, v. 5, n. 4, p. 269–298, dec 2007. Citado 2 vezes nas páginas 27 e 28.
- BLUM, C. et al. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, Amsterdam, The Netherlands, v. 11, n. 6, p. 4135–4151, sep 2011. Citado na página 29.
- BLUM, C.; ROLI, A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, ACM, New York, NY, USA, p. 268–308, sep 2003. Citado 2 vezes nas páginas 23 e 29.
- CHRISTOFIDES, N. et al. *Combinatorial optimization*. Chichester, UK: John Wiley, 1979. Citado na página 45.
- CUNHA, C. B. d. Experimentos computacionais com heurísticas de melhorias para o problema do caixeiro viajante. In: XVI CONGRESSO DA ANPET - ASSOCIAÇÃO NACIONAL DE PESQUISA E ENSINO EM TRANSPORTES. Natal, RN, Brasil, 2002. Citado na página 26.
- FERNANDES, A. M. d. R. *Inteligência Artificial - Noções Gerais*. Florianópolis, SC, Brasil: Visual books, 2005. Citado 3 vezes nas páginas 30, 31 e 32.
- FREITAS, F. G. de et al. Aplicação de metaheurísticas em problemas da engenharia de software: Revisão de literatura. In: II CONGRESSO TECNOLÓGICO INFOBRASIL. Fortaleza, Brasil, 2009. Citado na página 23.
- GÓMEZ ARTHUR T. E GALAFASSI, C. Uma abordagem aplicada ao problema de roteamento de veículos utilizando a busca tabu. In: XLIII SIMPÓSIO BRASILEIRO DE PESQUISA OPERACIONAL. Ubatuba, SP, Brasil, 2011. Citado na página 29.
- 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. Citado 2 vezes nas páginas 27 e 45.

- 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. Citado na página 45.
- 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. Citado na página 45.
- 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. Citado na página 45.
- LINDEN, R. *Algoritmos Genéticos*. Rio de Janeiro, RJ, Brasil: Ciência Moderna, 3ª Edição, 2012. Citado 11 vezes nas páginas 23, 30, 31, 32, 33, 34, 40, 48, 49, 51 e 61.
- MICHALEWICZ, Z. *Genetic Algorithms + Data Structures=Evolution Programs*. Berlin, Germany: Springer, 3rd Edition, 1996. Citado 2 vezes nas páginas 23 e 32.
- MICHALEWICZ, Z.; FOGEL, D. B. *How to Solve It: Modern Heuristics*. Berlin, Germany: Springer, 2nd Edition, 2004. Citado 5 vezes nas páginas 25, 48, 49, 50 e 51.
- 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. Citado 2 vezes nas páginas 27 e 40.
- OCHI, L. S. et al. A parallel evolutionary algorithm for the vehicle routing problem with heterogeneous fleet. *Future Generation Computer Systems*, Niterói, RJ, Brazil, v. 14, n. 5-6, p. 285–292, dec 1998. Citado na página 27.
- PACHECO, M. A. C. *Algoritmos Genéticos: Princípios e Aplicações*. jul 1999. Disponível em: <<http://www.ica.ele.puc-rio.br/Downloads/38/CE-Apostila-Comp-Evol.pdf>>. Acesso em: dezembro de 2012. Citado na página 30.
- 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. Citado 3 vezes nas páginas 27, 41 e 45.
- RAUBER, T.; RÜNGER, G. *Parallel Programming: for Multicore and Cluster Systems*. Berlin, Germany: Springer, 2010. Citado 8 vezes nas páginas 24, 35, 36, 37, 38, 39, 40 e 69.
- RAYWARD-SMITH, V. J. et al. *Modern Heuristic Search Methods*. Chichester, England: Jhon Wiley & Sons Ltd, 1996. Citado 6 vezes nas páginas 23, 25, 29, 30, 50 e 51.
- SILVEIRA, J. Porto da. *Problema do Caixeiro Viajante*. jun 2000. Disponível em: <<http://www.mat.ufrgs.br/~portosil/caixeiro.html>>. Acesso em: dezembro de 2012. Citado 2 vezes nas páginas 25 e 26.
- STEEMAN, Q. *The Vehicle Routing Problem With Drop Yards: A Dynamic Programming Approach*. 2012. Disponível em: <<http://essay.utwente.nl/61457/>>. Acesso em: dezembro de 2012. Citado 3 vezes nas páginas 25, 27 e 28.

- STÜTZLE, T. *Local Search Algorithms for Combinatorial Problems: Analysis, Improvements, and New Applications*. Washington, D.C., USA: IOS Press, Inc, 1999. Citado 3 vezes nas páginas 23, 29 e 30.
- TSAI, C.-W. et al. A time-efficient method for metaheuristics: Using tabu search and tabu ga as a case. In: *Hybrid Intelligent Systems, 2009. HIS '09. Ninth International Conference on*. Washington, D.C., USA: IEEE Computer Society, 2009. v. 2, p. 24–29. Citado na página 29.
- WANG, C.-H.; LU, J.-Z. A hybrid genetic algorithm that optimizes capacitated vehicle routing problems. *Expert Syst. Appl.*, Tarrytown, NY, USA, v. 36, n. 2, p. 2921–2936, mar 2009. Citado na página 50.
- 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. Citado 3 vezes nas páginas 27, 40 e 41.
- WILKINSON, B.; ALLEN, M. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers 2e*. Upper Saddle River, NJ, USA: Pearson Prentice Hall, 2nd Edition, 2005. Citado 6 vezes nas páginas 24, 35, 36, 38, 39 e 69.
- WINK, S.; BACK, T.; EMMERICH, M. A meta-genetic algorithm for solving the capacitated vehicle routing problem. In: EVOLUTIONARY COMPUTATION (CEC), 2012 IEEE CONGRESS ON. Brisbane, Australia, 2012. Citado na página 27.
- YAGIURA, M.; IBARAKI, T. Metaheuristics as robust and simple optimization tools. In: EVOLUTIONARY COMPUTATION, 1996., PROCEEDINGS OF IEEE INTERNATIONAL CONFERENCE ON. Nayoya University, Japan, 1996. Citado na página 29.
- YUSSOF, S.; RAZALI, R.; SEE, O. H. A parallel genetic algorithm for shortest path routing problem. In: FUTURE COMPUTER AND COMMUNICATION, 2009. ICFCC 2009. INTERNATIONAL CONFERENCE ON. Kuala Lumpur, Malaysia, 2009. Citado na página 42.
- ZHU, H. et al. Paralleling genetic annealing algorithm with openmp. In: PROCEEDINGS OF THE 2009 SECOND INTERNATIONAL CONFERENCE ON INTELLIGENT NETWORKS AND INTELLIGENT SYSTEMS. Washington, DC, USA, 2009. Citado 2 vezes nas páginas 36 e 37.
- ZHU-RONG, W. et al. A study of hybrid parallel genetic algorithm model. In: NATURAL COMPUTATION (ICNC), 2011 SEVENTH INTERNATIONAL CONFERENCE ON. Shanghai, China, 2011. Citado na página 42.