Universidade Federal do Pampa
Gabriella Lopes Andrade

Análise do Desempenho de um Algoritmo Genético Paralelizado com OpenMP Baseado em Rastros de Execução

Alegrete – RS
2016

Gabriella Lopes Andrade

Análise do Desempenho de um Algoritmo Genético Paralelizado com OpenMP Baseado em Rastros de Execução

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

2016

Gabriella Lopes Andrade

Análise do Desempenho de um Algoritmo Genético Paralelizado com OpenMP Baseado em Rastros de Execução

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 29 de novembro de 2016.

Banca examinadora:

Márcia C. Cera

Márcia Cristina Cera
Orientadora

Claudio Schepke

Claudio Schepke
UNIPAMPA

p/ Márcia C. Cera - participou via sistema on-line.

Lucas Mello Schnorr
UFRGS

Esse trabalho é dedicado aos meus pais,
que sempre me apoiaram nessa caminhada!

Agradecimentos

Em primeiro lugar agradeço a Deus, pois sem ele nada sou e nada posso fazer. Ele me sustentou nos momentos mais difíceis e com minha fé nele encontrei forças para continuar.

Aos meus pais Pedro e Rozangila, que sempre me apoiaram em minhas escolhas. Que me consolaram em momentos difíceis e comemoraram todas as minhas conquistas. Que com imenso amor me incentivaram a seguir meus sonhos.

A minha orientadora, Professora Doutora Márcia Cristina Cera, agradeço pela oportunidade e por acreditar na minha capacidade. Agradeço pela paciência, dedicação e competência na orientação. Por me ajudar em cada passo dessa conquista servindo de inspiração e apoio.

Ao meu namorado, companheiro e amigo Gabriel Quintana, que mesmo de longe sempre me incentivou a continuar. Agradeço por todo amor, carinho, compreensão e apoio.

Aos meus colegas e amigos Carolina Ramos, Emanuelle Leães, Jaime Mombach, Rafaela Robe e Sarah Quadros, que me acompanharam nessa caminhada. Agradeço por todos os momentos bons que passamos juntos, pelas conversas e risadas, e pelo o apoio moral com o chimarrão.

Agradeço a todas as pessoas que de alguma forma me apoiaram e torceram por mim. Por todas as orações e pensamentos positivos.

"O que é nascido de Deus vence o mundo;
e esta é a vitória que vence o mundo:
a nossa fé."
(Bíblia Sagrada, I João 5:4)

Resumo

Esse trabalho propõe a realização de uma análise do desempenho de um Algoritmo Genético (AG) paralelizado com a *Application Program Interface* (API) *Open Multi Processing* (OpenMP) a partir de seus rastros de execução. Esse AG é aplicado ao Problema de Roteamento de Veículos (PRV). O PRV é um problema de otimização combinatória, que consiste em rotear veículos com uma certa capacidade de transporte, para atender requisições de um grupo de cidades. Sendo que cada cidade pode ser visitada apenas uma vez e por apenas um veículo. E o custo total da rota não pode exceder a capacidade do veículo. A solução compreende um conjunto de rotas capaz de satisfazer a demanda de todos os clientes com o custo mínimo. O desempenho obtido pela paralelização do AG, embora tenha reduzido o tempo de execução, não foi o idealmente esperado. Logo, o objetivo geral desse trabalho é investigar as causas do baixo desempenho obtido pelo AG, realizando uma análise de desempenho a partir dos rastros de execução. A partir dessa análise, propor melhorias para esse algoritmo. Nesse trabalho a técnica de coleta que será utilizada é o rastreamento, que é a maneira mais detalhada de obter dados sobre o comportamento de aplicações paralelas. Para analisar os dados coletados vamos utilizar a técnica de análise interativa por visualização de rastros. Nessa técnica, os dados coletados a partir do rastreamento são convertidos em representações visuais. A ferramenta que será utilizada para realizar o rastreamento é a Score-P, que é utilizada para rastrear a execução de aplicações de alto desempenho. Para a análise dos dados utilizaremos a ferramenta de visualização Vampir. Nossos resultados mostraram que a paralelização do AG está de acordo para o modelo em que foi implementado e para o conjunto de instâncias do PRV utilizadas, não sendo identificadas possíveis melhorias.

Palavras-chave: Algoritmos Genéticos, Desempenho. OpenMP. Paralelização. PRV. Rastreamento. Score-P. Vampir.

Abstract

This work condutcs an analysis of the performance of a Genetic Algorithm (GA) parallelized with the Application Program Interface (API) Open Multi Processing (OpenMP) from their execution traces. This GA proposing a solution to the Vehicle Routing Problem (VRP). The VRP is a combinatorial optimization problem, which is to route vehicles with a certain transport capacity to meet requests from a group of cities. And the cost each city can be visited only once and only one vehicle. And the total cost of the route can not exceed the capacity of the vehicle. The solution comprises a set of routes capable of satisfying the demand of all the customers with the minimum cost. The results performance obtained by parallelization of GA, although it has reduced the running time, they were not ideally expected. Therefore, the general objective of this work is to investigate the causes of the low performance obtained by the GA, performing a performance analysis from the execution traces. From this analysis, propose improvements to this algorithm. In this work the collection technique used is the trace, which is the most comprehensive way to obtain data on the behavior of parallel applications. To analyze the data collected will use interactive analysis technique for trace display. In this technical, data collected from the tracking are converted into visual representations. The tool that will be used to perform a scan is the Score-P, which is a highly scalable tool for measuring high-performance application performance. For data analysis we used the Vampir visualization tool. Our results showed that the parallelization of GA is according to the model in which it was implemented and to the set of VRP instances used, no possible improvements were identified.

Key-words: Genetic Algorithms, Performance. OpenMP. Paralelization. VRP. Tracking. Score-P. Vampir.

Lista de ilustrações

Lista de tabelas

Lista de siglas

AG Algoritmo Genético

API *Application Program Interface*

CUDA *Compute Unified Device Architecture*

EZTrace *Easy To use Trace Generator*

GCC *GNU Compiler Collection*

Graphviz *Graph Visualization Software*

LEA Laboratório de Estudos Avançados em Computação

MPI *Message Passing Interface*

OpenCL *Open Computing Language*

OpenMP *Open Multi Processing*

OTF *Open Trace Format*

OTF2 *Open Trace Format* Versão 2

Pajé *Paje File Format*

PajeNG *Paje Next Generation*

PAPI *Precision Approach Path Indicator*

PRV Problema de Roteamento de Veículos

Pthreads POSIX *Threads*

SHMEM *Symmetric Hierarchical Memory*

TAU *Tuning and Analysis Utilities*

Intel TBB *Intel Threading Building Blocks*

UNIPAMPA Universidade Federal do Pampa

ViTE *Visual Trace Explorer*

Sumário

1 Introdução

O desenvolvimento de aplicações paralelas é considerado mais complexo do que o desenvolvimento de aplicações sequenciais por inúmeros motivos. Uma das principais dificuldades é desenvolver aplicações paralelas que explorem todo o paralelismo presente nos computadores atuais, os quais possuem múltiplas unidades de processamento. Nesse tipo de programação tem-se uma preocupação principalmente com o desempenho, o qual depende da maneira como é realizado o mapeamento entre os requerimentos da aplicação e os recursos computacionais disponíveis (PIOLA, 2007; SCHNORR, 2014b). Para auxiliar no desenvolvimento é realizada a análise de desempenho da aplicação paralela, com o objetivo de identificar as regiões do programa que realizam uma baixa exploração dos recursos computacionais disponíveis (SCHNORR, 2014b).

Esse trabalho propõe a análise de uma paralelização de um Algoritmo Genético (AG) aplicado ao Problema de Roteamento de Veículos (PRV), implementado por Gressler e Cera (2014). Para paralelizar o AG, Gressler e Cera (2014) utilizou técnicas de programação para memória compartilhada com a API OpenMP. Os resultados obtidos por Gressler e Cera (2014) mostraram que o desempenho da paralelização do AG ficou abaixo do ideal. Logo, esse trabalho visa identificar as possíveis causas do baixo desempenho obtido por essa paralelização.

A análise de desempenho de programas paralelos se divide em duas fases, a fase de coleta de dados e a fase da análise dos mesmos. Na fase de coleta, as informações sobre o comportamento do programação são gravadas, e na fase de análise, essas informações são analisadas com o objetivo de identificar os problemas de desempenho e suas causas (PIOLA, 2007; SCHNORR, 2014b). Existem várias técnicas tanto para a coleta, quanto para a análise de dados. Nesse trabalho vamos utilizar a técnica de rastreamento, que é uma técnica de coleta de dados guiada por eventos. Para realizar a análise dos dados coletados sobre o comportamento do AG a partir de rastreamento, iremos utilizar a técnica de análise interativa por visualização de rastros, a qual consiste em transformar os rastros coletados em representações visuais intuitivas (SCHNORR, 2014b).

Para realizar o rastreamento sobre o comportamento do AG iremos utilizar a ferramenta Score-P (VI-HPS, 2016), que é uma ferramenta altamente escalável para medição de desempenho de aplicações de alto desempenho. E para análise dos dados coletados a partir do rastreamento utilizaremos a ferramenta de visualização de rastros Vampir (GWT-TUD GmbH, 2016), que é uma ferramenta baseada em rastreamento interativo e escalável.

1.1 Objetivo

O objetivo geral desse trabalho é investigar as causas do baixo desempenho obtido por um AG paralelizado com OpenMP, realizando uma análise de desempenho a partir dos rastros de execução.

Para alcançar o objetivo geral, têm-se os seguintes objetivos específicos:

- Verificar o comportamento do AG em diferentes arquiteturas com múltiplos núcleos;
- Analisar rastros de execução paralela buscando identificar as possíveis causas do desempenho abaixo do ideal;
- Verificar o impacto das decisões de implementação, seu efeito nos rastros de execução e no desempenho do AG.

1.2 Organização deste trabalho

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 apresenta uma revisão dos conceitos que foram aplicados por Gressler e Cera (2014) no desenvolvimento e paralelização de um AG aplicado ao PRV. O Capítulo 3 apresenta os mecanismos utilizadas para realizar a coleta de dados e análise de aplicações paralelas, juntamente com ds que utilizaremos para analisar o desempenho do AG paralelo. O Capítulo 4 apresenta detalhes da arquitetura utilizada, os parâmetros do AG utilizados, juntamente com os resultados obtidos. O Capítulo 5 apresenta a conclusão do trabalho.

2 Contextualização

Esse capítulo tem por objetivo apresentar uma revisão dos principais conceitos que foram aplicados por Gressler e Cera (2014) no desenvolvimento e paralelização de um AG aplicado ao PRV. Ele está estruturado da seguinte maneira. A seção 2.1 contém uma revisão sobre Programação Paralela. A seção 2.2 detalhes do AG implementado por Gressler e Cera (2014). A seção 2.3 apresenta as considerações sobre esse capítulo.

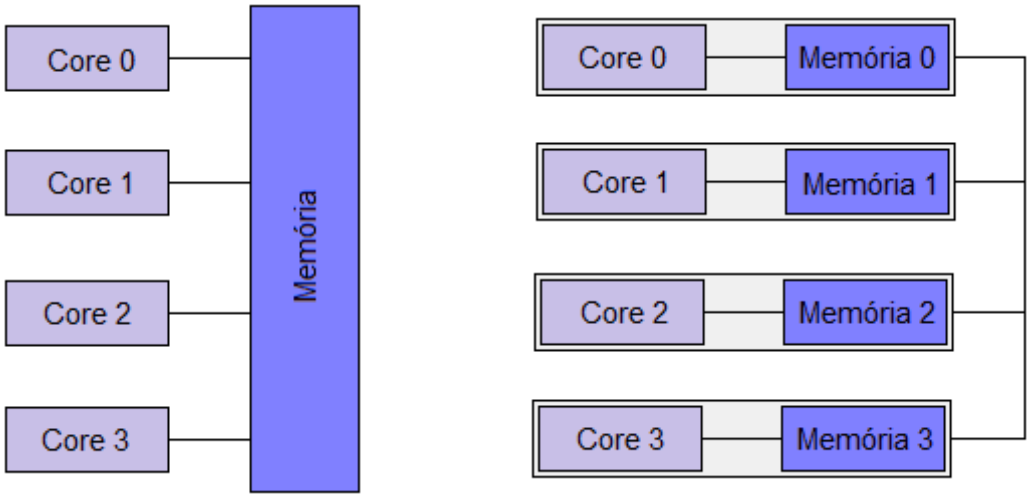
2.1 Programação Paralela

Existem muitas aplicações que necessitam de um grande poder de processamento para serem executadas em tempo hábil (SCHEPKE; LIMA, 2015). Como por exemplo a Previsão do tempo, a Procura de petróleo, Simulações físicas, etc.; as quais demorariam muitos dias ou até meses se fossem executadas sequencialmente (NAVAUX; ROSE; PILLA, 2011). Segundo Navaux, Rose e Pilla (2011) em alguns casos extremos, não seria possível executar esse tipo de aplicação por falta de memória. Para resolver esse tipo de problema é utilizada a programação paralela, a qual consiste em solucionar um problema dividindo-o em partes, de maneira que essas partes possam ser executadas em paralelo. Assim, a programação paralela busca reduzir o tempo de execução da versão paralela em relação a versão sequencial.

A programação paralela é realizada em máquinas com arquiteturas diferentes das convencionais, denominadas de arquiteturas paralelas (NAVAUX; ROSE; PILLA, 2011). Segundo Schepke e Lima (2015) arquiteturas paralelas são baseadas na utilização de múltiplas unidades de processamento, normalmente processadores, e assim esse tipo de arquitetura obtém um maior desempenho em relação a arquiteturas simples.

Segundo Pacheco (2011) uma possível forma de classificar as arquiteturas paralelas é conforme o compartilhamento de memória, podendo haver arquiteturas com memória compartilhada ou memória distribuída. A Figura 1 ilustra esses dois tipos de arquiteturas. No modelo de programação para memória compartilhada (Figura 1(a)), existe apenas um espaço de endereçamento compartilhada entre todos os processadores, o qual é usado para realizar a comunicação entre todos os processadores de forma implícita e bastante eficiente (NAVAUX; ROSE; PILLA, 2011). Já no modelo de programação com memória distribuída (Figura 1(b)) cada processador possui uma memória local e privada, a qual somente ele possui acesso (PACHECO, 2011). Como esse ambiente não utiliza variáveis compartilhadas a comunicação entre os processadores é feita de forma explícita, através de troca de mensagens (NAVAUX; ROSE; PILLA, 2011).

Figura 1 – Classificação de Arquiteturas Quanto ao Compartilhamento de Memória



(a) Arquitetura com Memória Compartilhada. (b) Arquitetura com Memória Distribuída.

Fonte: Adaptada de Pacheco (2011, p. 09)

Este trabalho será voltado a análise de desempenho de arquiteturas de memória compartilhada. Segundo Gressler e Cera (2014) este tipo de arquitetura foi amplamente popularizada após a difusão de arquiteturas *multi-core*.

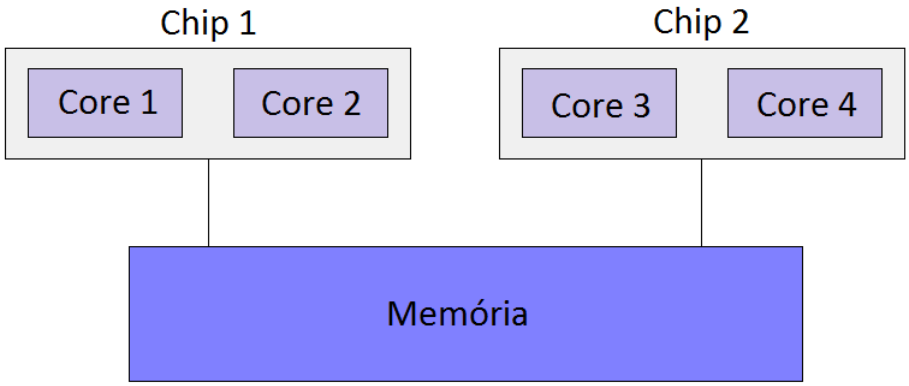
2.1.1 Arquiteturas Multi-core

As arquiteturas *Multi-core* surgiram da necessidade da indústria de produzir processadores cada vez mais rápidos (PACHECO, 2011). À medida que o tamanho dos transistores presentes no processador diminuia, era possível aumentar a densidade desses transistores, e assim, a velocidade do processador podia ser aumentada (PACHECO, 2011). Porém, ao aumentar a velocidade, consequentemente, aumenta-se o consumo de energia e a geração de calor (NAVAUX; ROSE; PILLA, 2011). Então, chegou-se a conclusão que essa abordagem não garantia ganhos reais (NAVAUX; ROSE; PILLA, 2011). Logo, a alternativa encontrada foi a de criar processadores *multi-core* (PACHECO, 2011).

Em um processador *Multi-core* as tarefas são dividas em operações concorrentes e distribuídas entre várias unidades de processamento (NAVAUX; ROSE; PILLA, 2011), conhecidas como núcleos. Esse tipo de processador possui vários núcleos em um único *chip* (PACHECO, 2011), como pode ver visualizado na Figura 2. Normalmente, os núcleos possuem *cache* de nível 1 privada, e as outras *caches* podem ou não ser compartilhadas entre todos os núcleos (PACHECO, 2011). As arquiteturas com memória compartilhada mais populares utilizam esse tipo de processador (PACHECO, 2011).

Existem várias interfaces de programação de aplicações (API) para o desenvolvimento de programas paralelos em memória compartilhada, entre elas *Intel Cilk Plus* (SURHONE; TENNOE; HENSSONOW, 2010), *Intel Math Kernel Library* (REINDERS,

Figura 2 – Arquitetura *Multi-core*



Fonte: Adaptada de Pacheco (2011, p. 34)

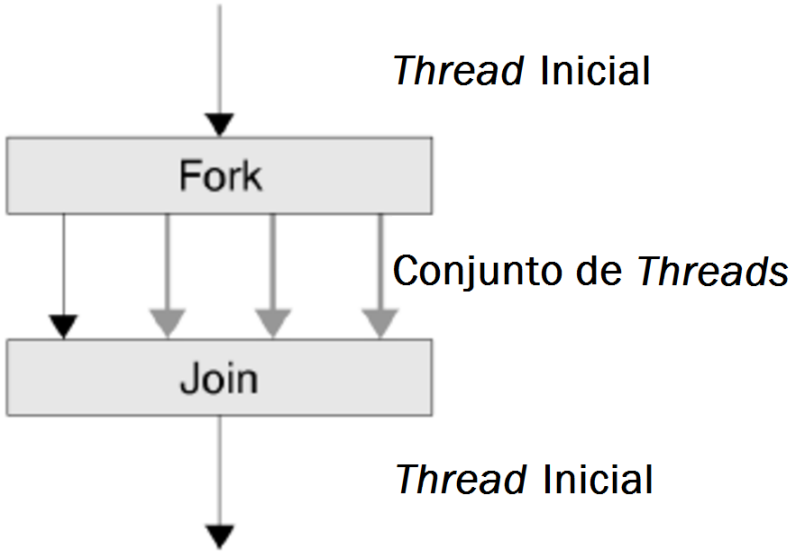
2010), *Intel Threading Building Blocks* (Intel TBB) (REINDERS, 2010), *Open Multi Processing* (OpenMP) (CHAPMAN; JOST; PAS, 2008), POSIX *Threads* (Pthreads) (BUTENHOF, 1997), etc.. A API utilizada para realizar a paralelização do Algoritmo Genético (AG) aplicado ao Problema de Roteamento de Veículos (PRV) foi a OpenMP. Segundo Gressler e Cera (2014) essa API foi escolhida por proporcionar uma interface simples e ser muito flexível para desenvolvimento de programas paralelos. A Seção 2.1.2 descreve as principais características dessa API.

2.1.2 OpenMP

A biblioteca OpenMP fornece um modelo escalável e portável para o desenvolvimento de programas com múltiplas *threads* para memória compartilhada e é disponível para as linguagens de programação C, C++ e Fortran. Ela é baseada em diretivas de compilação, que ao serem inseridas no código sequencial informam ao compilador quais blocos de código devem ser executados por *threads* em paralelo (CHAPMAN; JOST; PAS, 2008). A diretiva `#pragma omp parallel` delimita o trecho do código a ser paralelizado.

A OpenMP segue o modelo de paralelismo *Fork/Join* baseado em *threads*, onde o programa inicia com uma *thread*, a *master thread* (CHAPMAN; JOST; PAS, 2008). As regiões sequenciais do código são executadas pela *master thread* e as regiões paralelas podem ser executadas concorrentemente por um conjunto de *threads*. A *thread* inicial executa sequencialmente até encontrar uma construção paralela (`#pragma omp parallel`), logo após ela cria um conjunto de *threads* (*Fork*) e torna-se mestre das novas *threads* criadas. Esse conjunto de *threads* executa um bloco estruturado de código, o qual é chamado de região paralela. Ao término da execução da região paralela ocorre uma sincronização (*Join*), onde todas as *threads* são unidas (SCHEPKE; LIMA, 2015). A Figura 3 mostra

Figura 3 – Modelo de Execução *Fork/Join* do OpenMP.



Fonte: Adaptada de Chapman, Jost e Pas (2008, p. 24)

um exemplo de execução OpenMP *Fork/Join*, nesse exemplo existe uma região paralela com quatro *threads*.

Com OpenMP é possível alterar o número de *threads* utilizadas de duas maneiras. Uma maneira é através do uso do método de biblioteca `omp_set_num_threads()`. A outra maneira é através do uso da cláusula `num_threads()` logo após a diretiva `#pragma omp parallel`. Ambas recebem como argumento o número de *threads* que deverá ser utilizado (CHAPMAN; JOST; PAS, 2008).

A API OpenMP permite a paralelização de tarefas não iterativas através da diretiva `#pragma omp sections`. Dentro deste construtor, diferentes blocos de códigos são chamados de `sections`, e são distribuídos a diferentes *threads* (CHAPMAN; JOST; PAS, 2008).

A paralelização de laços `for` com OpenMP se dá através da diretiva `#pragma omp parallel for`. Ao inserir essa diretiva anteriormente ao laço que desejamos paralelizar, as iterações do laço serão distribuídas entre as *threads* que compõem o programa paralelo. A OpenMP fornece diferentes políticas de distribuição de iterações de laços `for` entre *threads* (cláusula `schedule(política[,chunk])`). Segundo Chapman, Jost e Pas (2008) as políticas de distribuição de iterações são definidas como:

- *Static:* A distribuição das iterações é realizada de forma igualitária entre todas as *threads*, de forma estática e em tempo de compilação;
- *Dynamic:* A distribuição das iterações é realizada de forma dinâmica, ou seja, um

bloco de iterações é atribuído a cada *thread* que tenha terminado seu bloco anterior, enquanto ainda houverem blocos a serem executados;

- *Guided :* A distribuição das iterações também é realizada de forma dinâmica, mas o bloco de iterações inicia grande e vai diminuindo até chegar ao tamanho do *chunk*;
- *Runtime:* A distribuição das iterações é realizada em tempo de execução, e dependem da implementação da biblioteca OpenMP.

Segundo Schepke e Lima (2015) como a OpenMP é uma API para memória compartilhada, a maioria das variáveis em memória são compartilhadas, porém, nem todas as variáveis podem ser compartilhadas. Como exemplo de variáveis que não podem ser compartilhadas temos as variáveis da pilha de funções e automáticas dentro de uma região paralela, as quais são privadas (SCHEPKE; LIMA, 2015). A OpenMP permite especificar e modificar o modo de acesso aos dados dentro de construções por meio de cláusulas (CHAPMAN; JOST; PAS, 2008). As cláusulas mais utilizadas são:

- *shared*: O modo de acesso aos dados é compartilhado entre todas as *threads*;
- *private*: Cria uma nova cópia local para cada *threads*.

Existem outras cláusulas disponíveis no OpenMP, as quais são cláusulas específicas relacionadas a outras regras e restrições (CHAPMAN; JOST; PAS, 2008). Uma dessas cláusulas é a cláusula `reduction(operador:lista)`, a qual permite acumular um certo valor de forma concorrente dentro de um laço (SCHEPKE; LIMA, 2015). Utilizando essa cláusula é possível especificar operações matemáticas de modo que eles possam ser realizados por *threads* em paralelo, sem a necessidade de modificar todo o código (CHAPMAN; JOST; PAS, 2008). O parâmetro `operador` especifica o operador da operação a ser realizada e o parâmetro `lista` especifica a lista de variáveis a serem acumuladas.

O pseudocódigo abaixo mostra um exemplo de cálculo do Pi paralelizado com OpenMP (CHAPMAN; JOST; PAS, 2008). Nele a paralelização do laço `for` é realizada utilizando a diretiva `#pragma omp parallel for` (linha 8). Nesse exemplo, utilizou-se a cláusula de dados `shared` para declarar explicitamente a variável `h` como uma variável compartilhada entre todas as *threads* que compõem o programa paralelo (linha 8). A variável `i` é declarada como privada de forma implícita, de acordo com as regras de compartilhamento de dados padrão do OpenMP (linha 9). E a variável `x` é compartilhada por padrão (CHAPMAN; JOST; PAS, 2008). Também utilizou-se a cláusula `reduction(+:soma)` (linha 8), a qual realiza uma operação de redução `+` para acumular os resultados na variável `soma`.

```
1: void compute(int n){
2:     int i;
3:     double h, x, soma;
4:
5:     h = 1.0/(double) n;
6:     sum = 0.0;
7:
8:     #pragma omp parallel for shared(h) reduction(+:soma)
9:         for(i=1; i<=n; i++){
10:            x = h * ((double)i - 0.5);
11:            soma += (1.0 / (1.0 + x*x));
12:        }
13:    pi = h * soma;
14:}
```

2.1.3 Desempenho

O principal objetivo ao desenvolver uma aplicação paralela é o reduzir seu tempo de execução em relação a sua versão sequencial, e assim conseguir um maior desempenho. Se conseguirmos dividir a carga de trabalho igualmente entre p núcleos de um processador, de forma que um processo ou *thread* execute por núcleo sem acrescentar nenhuma carga adicional a esses núcleos, então nossa aplicação paralela será executada p vezes mais rápido que sua versão sequencial (PACHECO, 2011). Quando isso acontece, dizemos que o programa paralelo conseguiu um *Speedup* linear ou ideal. Onde valor do *Speedup* é igual ao número de núcleos utilizados (p) (PACHECO, 2011).

Na prática, é pouco provável obter um *Speedup* linear, pois o uso de vários processos/*threads* quase sempre resultam em alguma sobrecarga (PACHECO, 2011). Um exemplo de sobrecarga é a existência de seções críticas em programas de memória compartilhada, que exigem mecanismos de exclusão mútua, os quais forçam a serialização da execução da seção crítica. O aumento do número de processos/*threads* consequentemente aumenta a sobrecarga gerada e consequentemente diminui o valor do*Speedup* (PACHECO, 2011).

Segundo Wilkinson e Allen (2005), o valor do *Speedup* obtido por uma paralelização é calculado a a partir da seguinte fórmula:

$$Speedup = \frac{Tempo\ de\ execução\ Sequencial}{Tempo\ de\ execução\ usando\ p\ processadores}$$

Como o $Speedup = p$ é improvável, a medida que p aumenta, espera-se que o valor do *Speedup* fique ainda mais longe do ideal. Para saber se o valor do *Speedup* obtido

está ficando cada vez mais longe do ideal conforme p aumenta, é utilizando o cálculo da Eficiência (PACHECO, 2011). Segundo Wilkinson e Allen (2005), a Eficiência de uma paralelização é dada pela seguinte fórmula:

$$Eficiência = \frac{Tempo\ de\ execução\ Sequencial}{Tempo\ de\ execução\ usando\ p\ processadores \times p}$$

$$Eficiência = \frac{Speedup}{p}$$

Para avaliar o desempenho da paralelização de uma aplicação, também devemos levar em consideração a lei de Amdahl, que ficou conhecida na década de 1960 a partir de uma observação de Gene Amdahl (PACHECO, 2011; WILKINSON; ALLEN, 2005). Essa lei diz que, a menos que praticamente todo programa sequencial seja paralelizado, o aumento do desempenho vai ser limitado, independente do número de núcleos disponíveis no processador (PACHECO, 2011). Se uma fração r de nosso programa sequencial não é paralelizado, então a lei de Amdahl diz que não podemos obter um *Speedup* melhor que $1/r$. Por exemplo, se r for um valor pequeno como 1/100, e tivermos um sistema com milhares de núcleos, não poderemos possivelmente obter um *Speedup* maior do que 100 (PACHECO, 2011).

Além de o tempo de execução do programa paralelo, o *Speedup* e a Eficiência dependerem do número de núcleos e o número de processos/*threads* utilizadas, eles também dependem do tamanho do problema. Porém, a lei de Amdahl não leva em consideração o tamanho do problema. Segundo a lei de Gustafson (PACHECO, 2011; WILKINSON; ALLEN, 2005), para muitos problemas, à medida que aumentamos o tamanho do problema, a parte do programa que não foi paralelizada diminui de tamanho. Logo, todos esses fatores devem ser levados em consideração na hora de avaliar o desempenho de uma paralelização.

2.2 Características do Algoritmo Genético aplicado ao Problema do Roteamento de Veículos

Essa Seção tem por objetivo descrever as características do AG aplicado ao PRV. A Seção 2.2.1 descreve o PRV e a Seção 2.2.2 explica o que é o AG, seu funcionamento, como o algoritmo foi implementado e paralelizado.

2.2.1 Problema do Roteamento de Veículos

O PRV é um problema de otimização combinatória (GOMÉZ; GALAFASSI, 2011) que possui um papel importante na área de gerenciamento da distribuição e logística

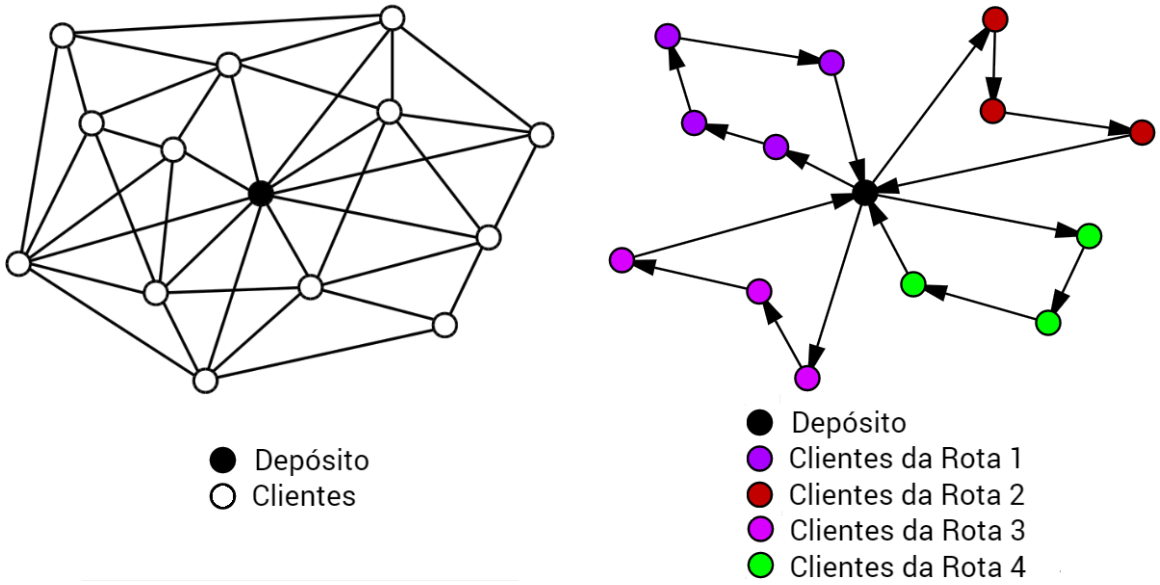
(ARENALES; ARMENTANO; MORABITO, 2007). É um problema pertencente à classe dos problemas NP-Difíceis, os quais são difíceis de solucionar utilizando métodos exatos por se tratarem de problemas com um alto custo computacional (GOMÉZ; GALAFASSI, 2011).

No PRV tem-se um depósito de abastecimento, existe uma demanda de um determinado produto para cada cidade a ser visitada e o veículo possui uma capacidade limitada (REGO; ALIDAEE, 2006). O veículo deve sair do depósito, visitar cada uma das cidades e retornar novamente ao depósito. Sendo que cada cidade deve ser visitada apenas uma vez e por apenas um veículo, e que a demanda total atribuida a cada rota não exceda a capacidade do veículo (ARENALES; ARMENTANO; MORABITO, 2007). O objetivo é encontrar um conjunto de rotas capaz de satisfazer a demanda de todos as cidades, e cujo somatório seja mínimo.

Segundo Rego e Alidaee (2006) o PRV é geralmente representado por um grafo $G = (V, E)$ não direcionado onde:

- $V = \{v_0, \ldots, v_n\}$ é o conjunto de vértices, sendo que o vértice v_0 representa o depósito e os demais vértices representam as cidades $(V - \{v_0\})$;
- $E = \{(v_i, v_j) : v_i, v_j \in V, i < j\}$ é o conjunto de arestas que ligam uma cidade a outra;
- n é o número total de cidades;
- c_{ij} representa o custo de viajar da cidade v_i até a cidade v_j, onde $c_{ij} = cji$;
- q_i representa a demanda da cidade i, $i \in 1, \ldots, n$;
- k é o número total de veículos;
- A capacidade Q_k do veículo não deve ser ultrapassada: $\sum_{i=1}^{n} q_i y_i^k \leq Q_k$, $k = \{1, \ldots, m\}$;
- Cada cidade i é visitada por exatamente um veículo e m veículos visitam o depósito: $\sum_{k=1}^{m} y_i^k = \begin{cases} m, & \text{se} \quad i = 0, \\ 1, & \text{se} \quad i = 1, \ldots, n \end{cases}$
- O mesmo veículo entrou em uma cidade é o mesmo que partiu dela: $\sum_{i=0}^{n} x_{ij}^k = \sum_{i=0}^{n} x_{ji}^k = y_i^k, \begin{array}{c} j = 0, \ldots, n \\ k = 1, \ldots, m \end{array}$.
- $S = \{R_1, ..., R_m\}$ é o conjunto solução com m rotas, sendo uma rota para cada veículo;

Figura 4 – Exemplo de PRV.



(a) Um Exemplo de PRV. (b) Um Exemplo de Rotas no PRV.

Fonte: Adaptado de Gressler e Cera (2014, p. 28).

- $R_k = (v_0, v_{k_1}, v_{k_2}, \ldots, v_0)$ é o conjunto ordenado representando vértices consecutivos na rota do veículo k;
- A solução S define o custo total da viajem: $C(S) = \sum_{i \leq k \leq m} \sum_{(i,j) \in R_k} c_{ij}$;

A Figura 4 apresenta dois grafos, sendo que o primeiro grafo apresenta um exemplo de PRV (Figura 4(a)) e o segundo apresenta uma possível solução para esse PRV (Figura 4(b)). Na Figura 4(a) o depósito é representado por um círculo na cor preta, todos os clientes são representados por um círculo na cor branca e o veículo deve sair do depósito para atender as demandas de cada um dos clientes. Na Figura 4(b), para atender a demanda de todos os clientes presentes nesse exemplo foram necessárias 4 rotas: uma para atender a demanda dos clientes representados em Azul, outra para os clientes representados em Roxo, outra para os clientes em Vermelho e uma última para os clientes em Verde.

Segundo Goméz e Galafassi (2011) muitos métodos foram propostos para resolução do PRV, os quais podem ser classificados como Métodos Exatos, Heurísticas ou Metaheurísticas. Entre estes os que mais vem sendo utilizados para propor soluções ao PRV são as Metaheurísticas. A grande vantagem dessa técnica sobre as demais é o fato dela conseguir encontrar um ótimo local, através de movimentos que causam uma alteração da função objeto e assim explorar profundamente as regiões mais promissoras do espaço de solução. (GOMÉZ; GALAFASSI, 2011).

Segundo Goméz e Galafassi (2011) as meta-heurísticas mais utilizadas para solucionar o PRV são: Algoritmos Genéticos (AGs), Busca Tabu, Colônia de Formigas, Redes

Neurais e *Simulated Annealing*. Segundo Gressler e Cera (2014) a meta-heurística escolhida para a resolução do PRV foi o AG, por esse conseguir obter bons resultados quando aplicado a problemas que não possuem uma técnica especializada para resolvê-los.

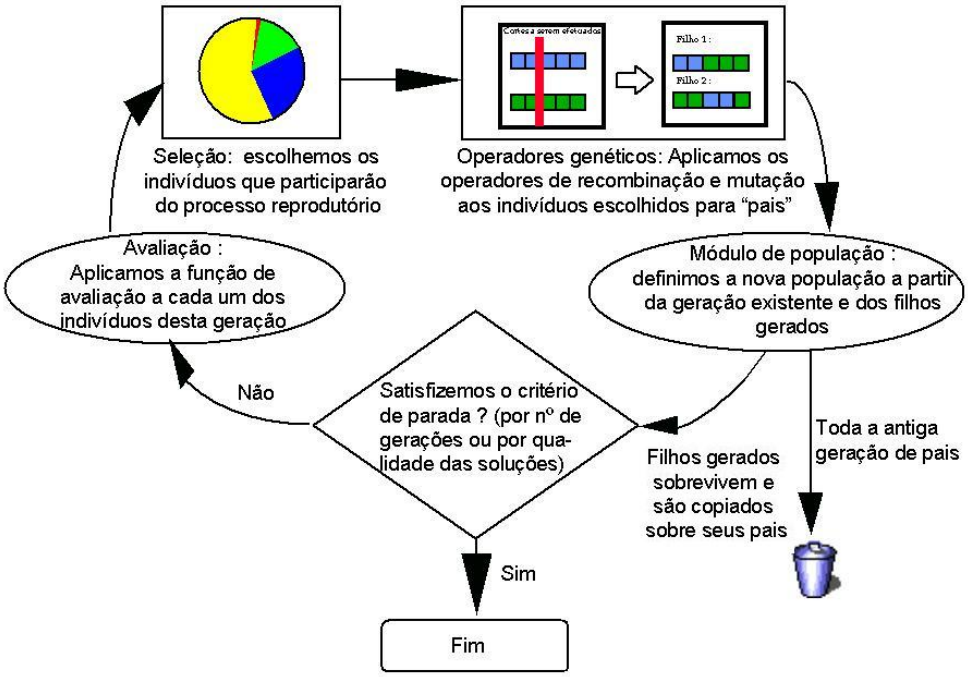
2.2.2 Algoritmo Genético

2.2.2.1 Visão Geral

Os AGs são algoritmos evolucionários que podem ser definidos como uma técnica de busca baseada na teoria da Evolução Natural de Charles Darwin (LINDEN, 2008a). Os sistemas desenvolvidos a partir dessa técnica são muito utilizados para procurar soluções de problemas complexos ou com espaço de busca muito grande (VELLASCO; ANDRADE; LIMA, 2014).

O AG funciona da seguinte forma, cada possível solução de um problema é codificado em uma estrutura de dados chamada cromossomo ou indivíduo. Esses indivíduos compõem a população inicial. Cada um dos indivíduos da população é aplicado ao processo de evolução, o qual consiste na avaliação dos indivíduos, seleção de quais indivíduos participarão do processo reprodutório, e aplicação dos operadores genéticos de recombinação e mutação aos indivíduos selecionados para gerar a nova população. A cada ciclo da evolução uma nova população é gerada, essa nova população substitui a antiga, que

Figura 5 – Esquema Simples de um AG.



Fonte: Linden (2008b).

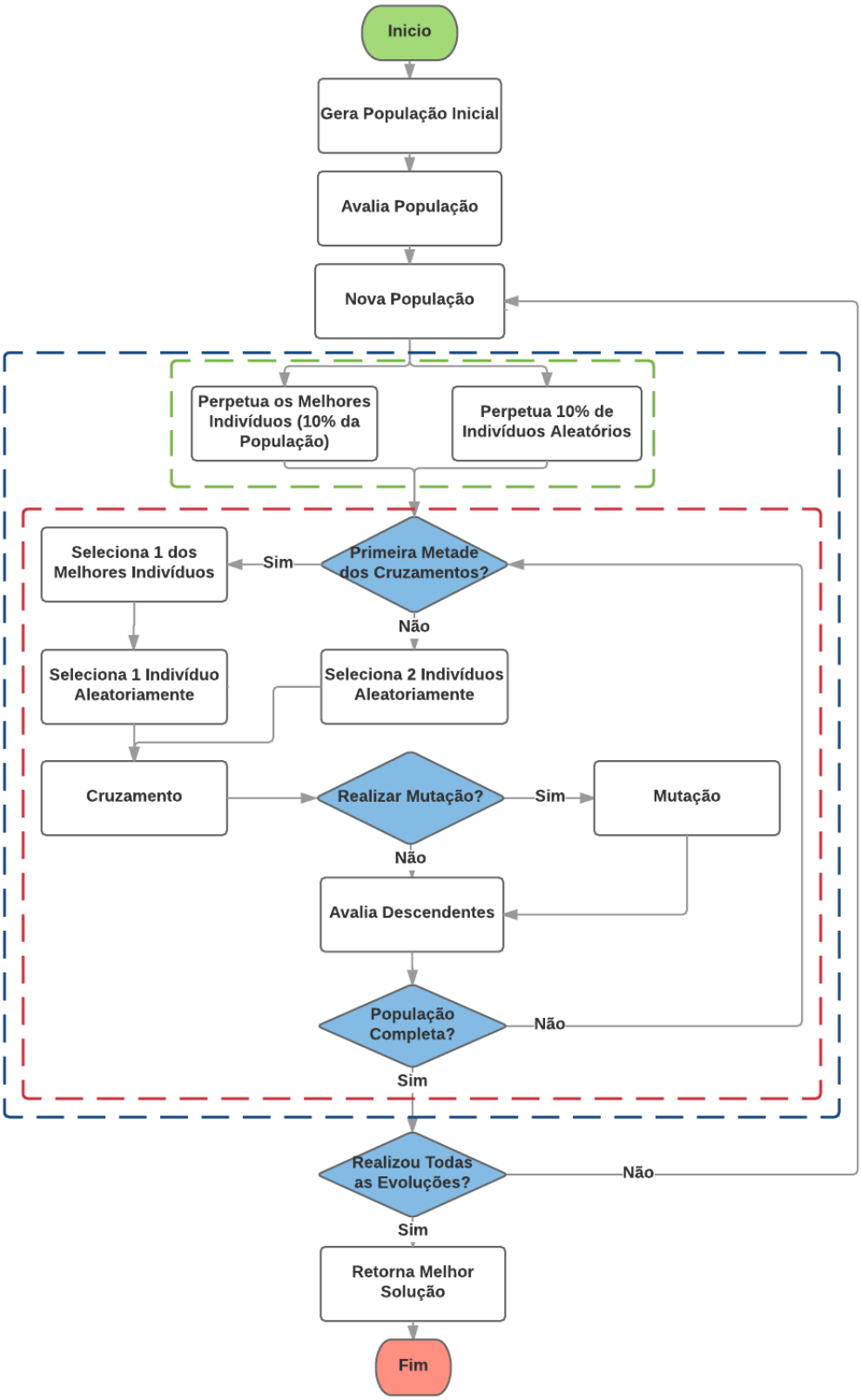
é descartada. Após realizar vários ciclos de evolução a população final deverá conter os indivíduos mais aptos (LINDEN, 2008a). A Figura 5 ilustra o funcionamento do AG.

2.2.2.2 Implementação Sequencial do Algoritmo Genético

A Figura 6 apresenta o fluxograma do Algoritmo Genético Sequencial implementado por Gressler e Cera (2014), onde os principais procedimentos são:

- **Gera População Inicial:** É responsável por gerar a primeira população ou o conjunto inicial de indivíduos do AG. Para cada indivíduo, ela sorteia a primeira cidade a ser visitada, e a partir dessa cidade, ela busca sempre a cidade mais próxima, até que todas as cidades tenham sido visitadas;
- **Avalia População Inicial:** É responsável por avaliar a aptidão dos indivíduos gerados calculando a distância percorrida para visitar as cidades e retornar ao depósito, até que todas as rotas tenham sido atendidas e não existam mais cidades a serem visitadas. As menores distâncias indicam as melhores soluções. Utiliza a função `Busca distância`, a qual recebe dois pontos como parâmetro, consulta a matriz distâncias e retorna a distância que está armazenada na linha e coluna correspondentes;
- **Nova Geração:** Recebe uma população como entrada e constrói a nova geração utilizando os operadores genéticos. Essa função é composta dos seguintes procedimentos:
  - **Perpetua 10% dos Melhores Indivíduos:** Copia 10% dos indivíduos mais aptos para a nova população;
  - **Perpetua 10% de Indivíduos Aleatórios:** Copia 10% de indivíduos aleatórios para a nova população;
  - **Seleciona 2 Indivíduos Aleatoriamente:** É responsável por selecionar 2 indivíduos aleatoriamente para realizarem o cruzamento;
  - **Seleciona 1 dos Melhores Indivíduos:** É responsável por selecionar 1 dos melhores indivíduos para o cruzamento;
  - **Seleciona 1 Indivíduo Aleatoriamente:** É responsável por selecionar 1 indivíduo aleatoriamente para realizar o cruzamento;
  - **Cruzamento:** É responsável por criar novos indivíduos para compor a nova população. Esse indivíduos herdam características de cada um dos seus genitores. Foram implementadas por Gressler e Cera (2014) cinco técnicas de cruzamento: três técnicas básicas e duas técnicas híbridas. As técnicas híbridas são combinações das técnicas básicas. As técnicas de cruzamento implementadas por Gressler e Cera (2014) são:

Figura 6 – Fluxograma do Algoritmo Genético Sequencial



Fonte: Adaptado de Gressler e Cera (2014, p. 67)

1. **Cruzamento Uniforme:** Nesse tipo de cruzamento é realizado um sorteio gene a gene de qual dos dois genitores o filho herdará as características. No final do processo são formados dois novos cromossomos completos;
2. **Cruzamento de 1 Ponto:** Nesse tipo de cruzamento um ponto é escolhido aleatoriamente para separar em partes menores os cromossos dos genitores. Após essa separação os cromossos são recombinados intercalando as partes de um genitor com as de outro, formando novos cromossomos completos;
3. **Cruzamento de 2 Pontos:** Nesse tipo de cruzamento dois ponto são escolhidos aleatoriamente para separar em partes menores os cromossos dos genitores. Após essa separação os cromossos são recombinados intercalando as partes de um genitor com as de outro, formando novos cromossomos completos;
4. **Híbrida 1:** No momento do cruzamento, a técnica a ser utilizada é selecionada de forma randômica entre as técnicas básicas descritas anteriormente. Dessa maneira, é possível encontrar melhores soluções devido a diversidade genética obtida através do uso aleatório das técnicas básicas de cruzamento;
5. **Híbrida 2:** Essa técnica analisa a cada evolução o custo da melhor solução. Se não houver uma melhoria em N/5 rodadas, troca-se a técnica de cruzamento utilizando uma fila circular, que contém as técnicas básicas. Sendo que N é o número de cidades na entrada. Dessa maneira, é possível encontrar melhores soluções alternando entre as técnicas de cruzamento sempre que houver melhora na qualidade da solução encontrada.

– **Mutação:** É responsável por alterar cromossos de um indivíduo. Apenas em uma certa porcentagem de indivíduos possui a probabilidade de ocorrer uma mutação. Foram implementadas por Gressler e Cera (2014) cinco técnicas de mutação:

1. **Mutação por Troca de Genes:** Nessa técnica genes distintos são selecionados aleatoriamente e trocados de posição;
2. **Mutação por Troca de Bloco de Genes:** A diferença dessa técnica para a anterior é que nela blocos de genes distintos são selecionados aleatoriamente e trocados de posição;
3. **Mutação por Inserção de Genes:** Nessa técnica um bloco de genes é selecionando aleatoriamente, removido do cromossomo e logo após inserido em outra posição no mesmo cromossomo;
4. **Mutação por Inversão de Genes:** Nessa técnica um bloco de gene é selecionado aleatoriamente e invertido. Por exemplo, se o bloco escolhido

fosse ABCD, após realizar a mutação por inversão, o mesmo seria DCBA.

5. **Mutação Randômica:** Essa técnica sorteia entre a melhor técnica de mutação utilizar entra as outras quatro técnicas.

– **Avalia Descendentes:** Funciona da mesma maneira que a Avaliação da População Inicial. A diferença é que essa função consiste em avaliar a aptidão dos indivíduos gerados enquanto é realizada a evolução. Após realizar a avaliação os indivíduos gerados são inseridos na nova população.

- **Retorna Melhor Solução:** o indivíduo mais apto é apresentado como melhor solução encontrada para o problema.

Os AGs retornam a melhor solução encontrada porém não garantem que essa seja a melhor solução possível. Segundo Gressler e Cera (2014) alguns ajustes de parâmetros, como aumentar o tamanho da população ou o número de gerações, melhora a qualidade das soluções do PRV. Porém, esse ajuste de parâmetros pode elevar o tempo de computação do AG. Logo, a alternativa escolhida por Gressler e Cera (2014) para melhorar o desempenho do AG aplicado ao PRV foi a de utilizar a Programação Paralela.

2.2.3 Algoritmo Genético Paralelizado com OpenMP

Para definir o que podia ser paralelizado no AG, Gressler e Cera (2014) analizou o algoritmo sequencial em conjunto com seu perfil da execução (*profiling*). Para gerar o perfil da execução Gressler e Cera (2014) utilizou a ferramenta `gprof`, que é uma ferramenta livre e incorporada ao *GNU Compiler Collection* (GCC).

Gressler e Cera (2014) identificou que a função `Cruzamento` recebeu 40.000.000 chamadas, as quais representaram 37,42% do tempo total de execução. Essa função não possui dependência de dados, logo pode ser paralelizada. Porém o grande número de chamadas faz com que essa função tenha um grade impacto no tempo total de execução, logo Gressler e Cera (2014) concluiu que ao invés de paralelizar o código da função é mais viável que ela receba chamadas paralelas.

A função `Avaliação` recebeu 80.000.000 chamadas, as quais representam 25,37% do tempo total de execução. Essa função faz chamadas sucessivas à função `Busca distância` e não é paralelizável pois possui dependência de dados. Logo Gressler e Cera (2014) concluiu que essa função também é uma candidata a receber chamadas paralelas.

A função `Busca distância` recebeu 4.349.590.210 chamadas, que representam 26,09% do tempo total de execução. Essa função não possui dependência de dados, porém realiza um acesso à memória, logo Gressler e Cera (2014) concluiu que não é viável que ela receba esforços de paralelização.

Gressler e Cera (2014) identificou que a função `Nova Geração` recebeu 100.000 chamadas, as quais representaram apenas 4,38% do tempo total de execução. Como essa função é chamada apenas uma vez a cada geração, sua chamada não pode ser paralelizada. No entanto, suas instruções foram identificadas por Gressler e Cera (2014) como fortes candidatas à paralelização. Podemos visualizar na Figura 6 que as funções que mais impactam no tempo de execução recebem chamadas na função `Nova Geração`, representada pela área envolvida por um retângulo tracejado na cor azul. A área envolvida por um retângulo tracejado na cor verde representa o trecho de código dedicado a perpetuação de indivíduos. A área envolvida por um retângulo tracejado na cor vermelho um laço de repetição que controla as chamadas às funções `Cruzamento` e `Avaliação`. Gressler e Cera (2014) conclui que ao paralelizar as instruções da função `Nova geração`, estaremos realizando chamadas diretas à funções `Cruzamento` e `Avaliação` e indiretas à função `Busca distância`.

Após Gressler e Cera (2014) definir as instruções da função `Nova geração` como alvo da paralelização, foi necessário identificar as regiões paralelas e qual a melhor diretiva de paralelização a ser aplicada para realizar a implementação da versão paralela com OpenMP.

2.2.3.1 Definição das Regiões Paralelas

Gressler e Cera (2014) indentificou duas regiões a serem paralelizadas na função `Nova Geração` a partir do perfil de execução gerado pela ferramenta `gprof`. Para delimitar os trechos de código a serem paralelizados em ambas regiões, Gressler e Cera (2014) utilizou a diretiva `#pragma omp parallel` da OpenMP. Essa diretiva informa ao compilador quais blocos de código devem ser executados por *threads* em paralelo (CHAPMAN; JOST; PAS, 2008). O tipo de paralelização a ser aplicada a cada região depende das características do bloco de instruções a ser executado.

Na Figura 6, a função `Nova geração` inicia procurando 10% dos melhores indivíduos e copiando-os para a nova população. Em seguida, copia outros 10% de indivíduos aleatórios. Essas tarefas apesar de estarem em uma sequência cronológica, podem ser realizadas ao mesmo tempo ou na ordem inversa, pelo fato de não possuírem dependência de dados. Logo, Gressler e Cera (2014) utilizou a diretiva `#pragma omp sections` do OpenMP, a qual é utilizada para paralelizar blocos que não possuem tarefas iterativas. Utilizando esta diretiva, cada bloco fica dentro de uma `section` e cada `section` é executada por uma *thread* diferente. Neste caso, como são apenas dois blocos de código, as tarefas de cada bloco serão sempre executadas por duas *threads*. Nesse trabalho pretende-se avaliar o impacto do uso das `sections` no desempenho quando utilizam-se mais que 2 *threads*.

Após realizar a perpetuação dos indivíduos, a função `Nova geração` inicia o laço

que controla os cruzamentos e realiza chamadas às funções `Cruzamento` e `Avaliação`. A execução deste laço se repete até que sejam criados todos os indivíduos necessários para completar a nova geração. As operações executadas nesse laço não possuem dependência de dados. É realizada uma chamada direta à função `Cruzamento`, duas chamadas diretas à função `Avaliação`, N chamadas indiretas à função `Busca distância` e algumas operações de controle. Logo, foi utilizada por Gressler e Cera (2014) a diretiva `#pragma omp for` do OpenMP, a qual é utilizada para a paralelização de laços. Ao utilizar essa diretiva, as iterações do laço são divididas por entre as *threads* e executadas em paralelo. O número de *threads* a ser utilizado foi definido por Gressler e Cera (2014) utilizando a a rotina `omp_set_num_threads(int num_threads)`, onde `num_threads` é um número inteiro que define o número de *threads* a ser utilizado.

2.2.3.2 Desempenho do Algoritmo Genético Paralelo

Para a validação do AG implementado, Gressler e Cera (2014) utilizou parte das instâncias do *Benchmark* de Christofides et al. (1979), o qual reúne 14 instâncias do PRV com número de cidades variando entre 50 e 199. Nesse *Benchmark* metade das instâncias possuem restrição quanto a capacidade do veículo e a outra metade, além de possuir essa restrição, também possui restrições quanto a distância percorrida por rota. Gressler e Cera (2014) utilizou três instâncias com restrição da capacidade de transporte do veículo, as quais estão disponíveis para *download* na *internet*[1]. As instâncias utilizadas por Gressler e Cera (2014) foram as instâncias c50, c100 e c120, as quais consistem em percursos com 50, 100 e 120 cidades, respectivamente. Cada instância possui sua própria capacidade de transporte do veículo, além disso, específica a coordenada do depósito, o número de cidades, a coordenada e a demanda de cada uma das cidades da instância (GRESSLER; CERA, 2014). Gressler e Cera (2014) testou o comportamento do AG utilizando instâncias com diferentes cargas computacionais, levando em consideração o tempo de computação que o AG implementado consome para processá-las.

Os testes de Gressler e Cera (2014) foram realizados em um microcomputador com um multiprocessador Intel® Core™ 2 Quad Q8300, com 2.50GHz de frequência e 4 núcleos físicos. Em função da configuração da arquitetura utilizada, Gressler e Cera (2014) executou cada instância do AG utilizando 2, 3 e 4 *threads*, além da versão sequencial. Gressler e Cera (2014) variou as políticas de distribuição de iterações entre as *threads* utilizando a cláusula `schedule(política,[chunk])` do OpenMP, onde `chunk` é o tamanho do bloco de iterações. As políticas de distribuição de iterações utilizadas por Gressler e Cera (2014) foram a *Static*, *Dynamic* e *Guided* e para cada política foram testados 3 tamanhos de `chunk`, resultando em uma granularidade fina, média ou grossa. No `chunk` de tamanho 1 as iterações são atribuídas às *threads* uma a uma, representando

[1] Disponível em <http://mistic.heig-vd.ch/taillard/problemes.dir/vrp.dir/vrp.html> (último acesso em junho de 2016).

Tabela 1 – Configuração das versões paralelas do AG que levaram o melhor *speedups*, a partir dos resultados de Gressler e Cera (2014) e (ROSA; CERA, 2015a).

Fonte	Instância	*Threads*	Tempo (s)	*Speedup*	Eficiência
Gressler e Cera (2014)	c50	4 *threads*	4,36	1,73	0,43
	c100	4 *threads*	228,71	2,26	0,57
	c120	4 *threads*	626,09	2,48	0,62
Rosa e Cera (2015a)	c50	8 *threads*	2,60	2,04	0,26
	c100	16 *threads*	121,00	2,64	0,17
	c120	16 *threads*	350,00	2,76	0,17
	c150	32 *threads*	1242,00	3,10	0,10

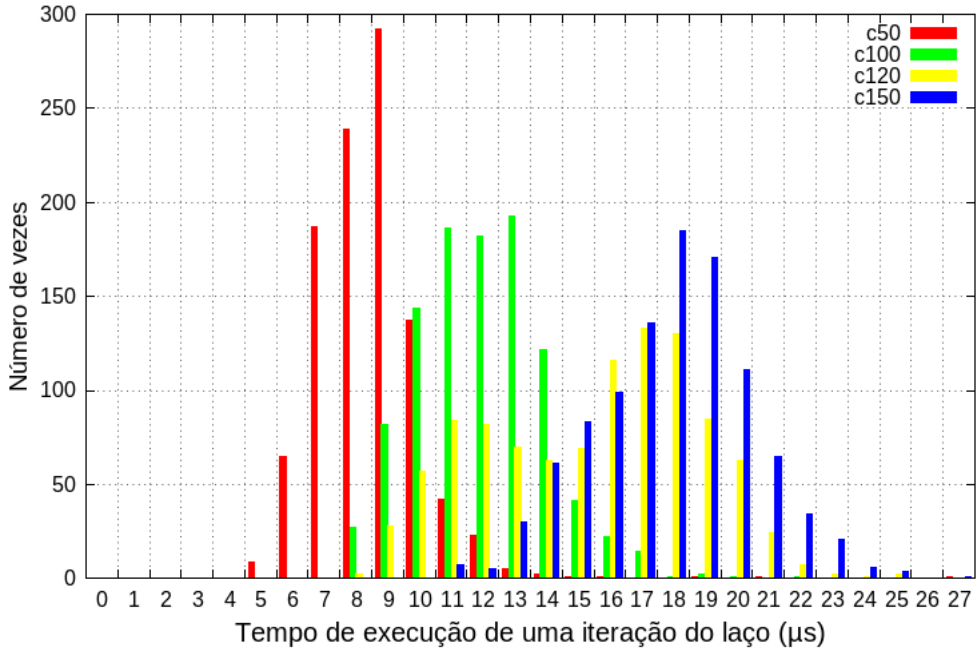
Fonte: Andrade e Cera (2016a, p. 70)

uma granularidade fina. O *chunk* 10% distribui por entre as *threads* blocos com 10% do total de iterações, representando uma granularidade média. E o *chunk* que representa a granularidade grossa é definido pelo total de iterações dividido pelo número de *threads* utilizadas.

A Tabela 1 apresenta as configurações do AG que levaram ao melhor desempenho a partir dos resultados obtidos por Gressler e Cera (2014), sendo que para todas as instâncias utilizadas os menores tempos de execução foram obtidos com 4 *threads* e consequentemente os maiores *Speedups*, na faixa de 1,73 à 2,48. Isso acontece porque o processador utilizado possui 4 núcleos físicos e o uso de uma *thread* por núcleo representou o melhor aproveitamento do potencial da arquitetura. Pode-se observar que valor do *Speedup* aumenta conforme executa-se instâncias com maiores cargas computacionais, porém estão sempre abaixo do ideal.

A política de distribuição de iterações que levou ao melhor desempenho foi a *static*, pois o AG possui carga regular, onde todas as iterações do laço demandam aproximadamente o mesmo tempo de processamento. Dessa forma é mais eficiente distribuir as iterações estaticamente e em tempo de compilação, o que não acarreta em custo adicional para distribuição da carga. A Figura 7 apresenta o gráfico com o tempo de execução em μs das iterações por instância do PRV pela quantidade de iterações que tem o mesmo tempo. A média do tempo de execução das iterações da instância c50 é de 9 μs, da c100 é de 12 μs, da c120 é 15 μs e da instância c150 é de 18 μ. Observando esse gráfico podemos ver que a maioria das iterações de cada instância está próxima à média, sendo que a instância c120 é que a possui maior variação no tempo de execução (de alguns μs). Isto ocorre devido a probabilidade de ocorrer ou não a mutação nos cromossomos (20%). A eficiência obtida por Gressler e Cera (2014) está na faixa de 0,43 à 0,62, onde a eficiência também é maior para as instâncias com maior carga computacional. Entretanto, a eficiência é baixa, sendo que usou um pouco mais de 60% do potencial da arquitetura. Isso demonstra que a aplicação possui limitações de desempenho e que uma avaliação mais

Figura 7 – Tempo de Execução do Laço por Instância do PRV



detalhada se faz necessário.

O trabalho realizado por Gressler e Cera (2014) foi expandido por Rosa e Cera (2015a), executando em uma arquitetura com um maior número de núcleos, buscando verificar a escalabilidade do AG. As instâncias do PRV utilizadas por Rosa e Cera (2015a) foram as instâncias c50, c100, c120 e c150, com percursos de 50, 100, 120 e 150 cidades, respectivamente. Os testes de Rosa e Cera (2015a) foram realizados em uma *Workstation* Dell Precision T7600 do Laboratório de Estudos Avançados em Computação (LEA) Universidade Federal do Pampa (UNIPAMPA). Ela possui 2 processadores Intel® Xeon® E5-2650 com 2.80 GHz de frequência. Cada processador possui 8 núcleos físicos com suporte a técnologia *Hyper-Threading*. Em função da configuração da arquitetura utilizada, Rosa e Cera (2015a) executaram cada instância do AG utilizando 2, 4, 8, 16, 32 e 64 *threads*, além da versão sequencial. A política de distribuição de iterações foi a *static* (ROSA; CERA, 2015b), devido a regularidade do tempo de execução das iterações do AG, como pode ser observada na Figura 7.

A Tabela 1 também apresenta as configurações do AG que levaram ao melhor desempenho a partir dos resultados obtidos por Rosa e Cera (2015a). Para as instâncias c50, c100, c120 e c150, os menores tempos de execução foram obtidos com 8, 16, 16 e 32 *threads*, respectivamente. Sendo que para a instância c50, o melhor desempenho foi atingido apenas quando suas *threads* executam sobre os 8 núcleos físicos de um dos processadores disponíveis, devido a sua baixa carga computacional, onde os ganhos obtidos acabam sendo afetados pela sobrecarga (*overhead*) da paralelização quando utilizam-se

mais de 8 *threads*. Para as intâncias c100 e c120 o melhor desempenho é obtido quando utilizam todos os núcleos disponíveis nos dois processadores. E para a instância c150 o uso de *Hyper-Threading* mostrou-se mais eficiente. Como essa instância possui uma carga computacional maior, o uso de *threads* suplementares favoreceu a distribuição da carga computacional.

Rosa e Cera (2015a) confirmaram que o *Speedup* aumenta conforme executava-se as instâncias com maiores cargas computacionais. Adicionalmente identificaram que o aumento do número de *threads*, associado ao uso de uma máquina *multicore* com um número maior de núcleos físicos, conseguiu melhorar o desempenho obtido por Gressler e Cera (2014), porém esse desempenho ainda permanece abaixo do ideal. A eficiência obtida por Rosa e Cera (2015a) está na faixa de 0,10 à 0,26. Logo, o aumento do número de *threads* diminui a eficiência do AG paralelo, confirmando que existe um gargalo na implementação do AG afetando o desempenho de sua paralelização, que impede uma melhora na eficiência. O foco deste trabalho é identificar tal gargalo e para isso serão analisados rastros de execução da aplicação.

2.3 Considerações sobre a Contextualização

Nesse capítulo foi apresentado o PRV, que é um problema de otimização combinatória que possui um papel importante na área de gerenciamento da distribuição e logística. As técnicas mais utilizadas para resolver esse problemas são as Meta-heurísticas, dentre elas os AG, que é a técnica a ser utilizada neste trabalho Gressler e Cera (2014). Foi apresentando o AG implementado por Gressler e Cera (2014), onde a partir da população inicial aplicam-se os operadores genéticos de cruzamento e mutação, e após realizar vários ciclos de evolução a população final deverá conter os indivíduos mais aptos.

Neste capítulo também foram apresentados os conceitos de programação paralela com memória compartilhada e distribuída. O foco deste trabalho é programação paralela com memória compartilhada aplicada em arquiteturas *multi-core*. A API investigada é o OpenMP, a qual apresenta diretivas de compilação que quando inseridas no código sequencial informam ao compilador quais blocos de códigos serão executados por *threads* em paralelo.

Vimos que Gressler e Cera (2014) identificou duas regiões a serem paralelizadas no código do AG. A primeira região possui dois blocos de código que podem ser executados em paralelo, e que não possuem tarefas iterativas, logo foi paralelizada utilizando a diretiva `#pragma omp sections` do OpenMP. A segunda região é controlada por um laço `for`, logo essa região foi paralelizada utilizando a diretiva `#pragma omp for` do OpenMP, onde as iterações do laço serão distribuídas entre as *threads*.

Apresentou-se um breve resumo dos principais resultados obtidos em trabalhos

anteriores. Sendo que os os melhores *Speedups* obtidos por Gressler e Cera (2014) para as instâncias c50, c100 e c120, utilizando 4 *threads*, representam *Speedups* na faixa de 1,73 à 2,48. E os *Speedups* obtidos por Rosa e Cera (2015a) para as instâncias c50, c100, c120 e c150, utilizando 8, 16, 16 e 32 *threads* respectivamente, representam *Speedups* na faixa de 2,04 à 3,10. Onde todos os *Speedups* foram abaixo do ideal.

O próximo capítulo apresenta as principais técnicas utilizadas para realizar a análise de desempenho de programas paralelos. Como este trabalho será voltado a uma análise mais aprofundada do desempenho da paralelização do AG aplicado ao PRV desenvolvida por Gressler e Cera (2014), utilizando rastros de execução, também serão apresentadas as ferramentas que iremos utilizar para realizar a coleta e analisar os rastros de execução.

3 Mecanismos para Análise de Desempenho

Existem diversas razões pelas quais o desenvolvimento de aplicações paralelas é considerado mais complexo quando comparado ao desenvolvimento de aplicações sequenciais. Na programação paralela tem-se uma preocupação com as diversas linhas de controle executando de forma simultânea e concorrente, com as interações, e principalmente com o desempenho da aplicação (PIOLA, 2007). Segundo Schnorr (2014b) um dos principais fatores que dificultam o desenvolvimento desse tipo de aplicação é a escalabilidade dos supercomputadores paralelos atuais, os quais possuem múltiplas unidades de processamento. A grande dificuldade está em desenvolver uma aplicação que explore todo o paralelismo presente nessas máquinas (SCHNORR, 2014b). Logo, o desenvolvimento de uma aplicação paralela com um bom desempenho depende da maneira como é realizado o mapeamento entre os requerimentos da aplicação e os recursos disponíveis (SCHNORR, 2014b).

Segundo Schnorr (2014b) a análise de desempenho é uma etapa muito importante no desenvolvimento de aplicações paralelas, ela tem por objetivo identificar as regiões do programa que realizam uma baixa exploração dos recursos computacionais. Nessa etapa o desenvolvedor realiza várias execuções experimentais da aplicação, onde coleta informações a respeito de sua execução, e realiza a análise desses dados (SCHNORR, 2014b). E assim, consegue determinar o quão eficiente é a aplicação paralela e quais ajustes são necessários para melhorar seu desempenho (PIOLA, 2007).

A análise de desempenho é composta por duas fases, a fase de coleta de dados e fase de análise desses dados (SCHNORR, 2014b). A fase de coleta de dados consiste em obter informações sobre o desempenho do programa (PIOLA, 2007), o comportamento da aplicação e do sistema (SCHNORR, 2014b). Já a fase de análise dos dados consiste em identificar os problemas de desempenho durante a execução da aplicação e quais as suas causas (SCHNORR, 2014b). Essas duas fases podem ser executadas de forma simultânea (Análise *online*) ou de maneira separada (Análise *offline*) (SCHNORR, 2014b). Existem várias técnicas tanto para a coleta de dados, quanto para a análise dos mesmos. A técnica de coleta de dados a ser utilizada depende exclusivamente do tipo de análise que se deseja realizar (SCHNORR, 2014b).

Esse capítulo tem por objetivo apresentar as principais técnicas utilizadas para realizar a coleta de dados e análise de aplicações paralelas. Esse está estruturado da seguinte maneira. A seção 3.1 apresenta as principais técnicas utilizadas para realizar a coleta dos dados da aplicação. A seção 3.2 apresenta as técnicas que já foram utilizadas por Gressler e Cera (2014) para coletar dados sobre o comportamento do AG paralelo, e

quais técnicas de coleta nós utilizaremos. A seção 3.3 apresenta a técnica de análise de desempenho que será utilizada, juntamente com as ferramentas que serão utilizadas para realizar essa análise (subseção 3.3.1). A seção 3.4 apresenta as considerações sobre esse capítulo.

3.1 Técnicas Para Coleta de Dados

Como mencionado anteriormente, a fase de coleta de dados tem por objetivo registrar informações sobre o comportamento da aplicação e do sistema (SCHNORR, 2014b). Nessa fase é possível identificar quais os componentes da aplicação que limitam o seu desempenho, é possível medir o uso de recursos utilizados, o tempo gasto em cada chamada às rotinas, o número de execuções de cada bloco do programa, verificar a comunicação e sincronização realizada, e também identificar quais partes da aplicação gastam mais tempo durante a execução (PIOLA, 2007). Essas informações são geralmente registradas em arquivos e podem ser coletadas durante ou após a execução da aplicação, em alguns casos podem também ser exibidas ao usuário em tempo real (PIOLA, 2007).

Existem várias técnicas para realizar a coleta de dados. As técnicas de coleta podem ser classificadas de acordo com a maneira em que o registro da informação é lançado, podendo ser pelo tempo ou por eventos (SCHNORR, 2014b). As técnicas dirigidas pelo tempo coletam as informações sobre a aplicação de maneira periódica, entre intervalos regulares de tempo que são especificados pelo analista (SCHNORR, 2014b). Já as técnicas dirigidas por eventos coletam as informações conforme a ocorrência de eventos inseridos no código da aplicação pelo analista ou por alguma biblioteca de observação (SCHNORR, 2014b). Segundo Schnorr (2014b) a técnica guiada pelo tempo que se destaca é a amostragem, e as técnicas guiadas por eventos podem ser a contagem, a cronometragem e o rastreamento. Cada uma dessas técnicas é descrita a seguir.

3.1.1 Amostragem

A amostragem é uma técnica de coleta de dados guiada pelo tempo. Essa técnica consiste em examinar de forma periódica o estado de um programa ou o valor de uma variável necessária para a análise de desempenho, ou seja, de tempos em tempos ela realiza a coleta dos dados necessários para realizar a análise (SCHNORR, 2014b). A medição do valor ocorre em intervalos regulares de tempo, os quais são definidos pelo analista, e assim estabelecendo a frequência da amostragem (SCHNORR, 2014b). No momento da medição o sistema de coleta dispara uma ordem de observação do comportamento da aplicação, após o intervalo de tempo definido o sistema se reconfigura pra que a medição ocorra novamente (SCHNORR, 2014b).

Essa técnica é muito usada para gerar um perfil de execução da aplicações (SCH-

NORR, 2014b). Verificando por exemplo, qual função do programa está sendo executada, gravando essa informação no disco ou na memória, incrementando um contador associado ao nome da função. Ao terminar a execução, está disponível o perfil de execução da aplicação, o qual possui uma lista das funções mais executadas (SCHNORR, 2014b). Esse perfil só é capaz de informar quantas vezes uma função foi executada segundo a amostragem (SCHNORR, 2014b).

A qualidade da amostragem depende do intervalo entre as medições. Quanto menor for o intervalo entre as medições, maior será a qualidade da amostra. E quanto maior for esse intervalo, menor será a confiabilidade do perfil de execução obtido. Um intervalo de tempo menor entre as medições permite que comportamentos que duram menos tempo sejam capturados por amostragem, o que pode não acontecer se o intervalo entre as medições for muito grande (SCHNORR, 2014b). Por outro lado, se a frequência de amostragem for muito alta, a intrusão causada pela coleta será excessiva Schnorr (2014b).

Para realizar a amostragem são utilizadas ferramentas como `prof` e `gprof`. Essas ferramentas são muito utilizadas para depurar o desempenho de programas sequencias, porém, esse tipo de ferramenta pode falhar ao encontrar problemas comuns em programas paralelos (STEIN, 2001). Como o perfil de execução retornado pela amostragem só é capaz de informar quantas vezes uma função foi executada segundo essa amostragem (SCHNORR, 2014b), esse perfil não é suficiente para mostrar gargalos ou para avaliar o tempo gasto comunicação ou o tempo ocioso (STEIN, 2001).

3.1.2 Contagem

A técnica de contagem é uma técnica dirigida por eventos, com a qual é possível medir a quantidade de vezes que um evento ocorre (SCHNORR, 2014b). Essa técnica pode ser utilizadas para registrar por exemplo, quantas vezes uma função foi executada, o número de acessos a uma variável compartilhada (SCHNORR, 2014b), o número total de mensagens, o volume total das mensagens ou o número de mensagens enviadas entre cada par de processos (PIOLA, 2007). E dessa forma, é possível descobrir qual parte do programa é mais utilizada e focar na otimização da mesma (SCHNORR, 2014b).

Para medir a quantidade de vezes que um evento ocorre a técnica de contagem utiliza locais de armazenamento chamados de contadores, os quais são incrementados a cada vez que um evento específico ocorre (PIOLA, 2007). Segundo Schnorr (2014b) a utilização desses contadores na aplicação paralela apresenta uma baixa intrusão, o que é uma vantagem. Logo, essa técnica pode ser utilização em aplicações com um longo tempo de duração e nos eventos de provavelmente sejam muito frequentes (SCHNORR, 2014b).

Para implementar a técnica de contagem em programas paralelos pode-se utilizar

programas de computadores ou equipamentos computacionais. A vantagem de utilizar programas para implementar a técnica de contagem ao invés de equipamentos computacionais está na flexibilidade, na capacidade de adaptação e portabilidade (SCHNORR, 2014b). Porém a desvantagem de utilizar programas para implementar contadores está no efeito de sonda (o tempo gasto para registrar o comportamento da aplicação), o qual pode ser mais significativo quando comparada a implementação diretamente no equipamento computacional (SCHNORR, 2014b).

Existem contadores embutidos na maioria dos processadores modernos, os quais são conhecidos como contadores de *hardware* (BROWNE et al., 2000). Utilizando esse tipo de contador é possível obter informações importantes sobre o desempenho de partes críticas da aplicação parelela (BROWNE et al., 2000). Porém o acesso a esses contadores é muitas vezes mal documentado, instável ou indisponível. Um exemplo de interface é a *Precision Approach Path Indicator* (PAPI), com o objetivo de tornar mais fácil o acesso a esses contadores (BROWNE et al., 2000). A PAPI é uma interface utilizada para o monitorar o desempenho de aplicações paralelas, obtendo dados a respeito do consumo de energia desse tipo de aplicação(BROWNE et al., 2000).

3.1.3 Cronometragem

A cronometragem é uma técnica guiada por eventos. Nessa técnica a medição do tempo de execução de uma região do código da aplicação paralela é realiza inserindo instruções adicionais, as quais são responsáveis por cronometrar o tempo de execução (SCHNORR, 2014b). Assim como a amostragem essa técnica é utilizado para gerar um perfil da execução do programa paralelo, porém com algumas diferenças. Uma diferença entre essas técnicas é que a cronometragem é guiada pelas instruções adicionadas ao programa a ser analisado, logo, é guiada por eventos e não pelo tempo (SCHNORR, 2014b). Outra diferença é que a cronometragem mede exatamente quanto tempo foi gasto para executar uma função, enquanto que a amostragem mede o número de vezes que uma função é chamada na execução da aplicação (SCHNORR, 2014b).

Para realizar a cronometragem de uma aplicação paralela é necessário modificar seu código. Quanto maior for o número de modificações realizadas pelo desenvolvedor para cronometrar o tempo em regiões e funções da aplicação, maior é a intrusão causada por essa técnica. Logo, na cronometragem, o nível de intrusão está relacionado diretamente com a quantidade de instruções adicionadas no código para cronometrar o tempo (SCHNORR, 2014b). Ao Selecionar um número reduzido de funções a serem cronometradas é mais fácil controlar o nível de intrusão (SCHNORR, 2014b). Porém, mesmo que hajam poucas funções a serem cronometradas, e essas funções sejam executadas frequentemente, poderá ocorrer uma alteração excessiva do comportamento natural da aplicação paralela, e assim, o perfil de execução gerado será de baixa qualidade (SCHNORR, 2014b).

Existe um problema ao realizar a cronometragem em uma aplicação paralela. Pode se tornar difícil identificar quais regiões do código da aplicação podem ser cronometradas para obter informações importantes sobre o comportamento dessa aplicação, conforme a complexidade da mesma (SCHNORR, 2014b).

3.1.4 Rastreamento

Segundo (PIOLA, 2007) a técnica de rastreamento é a maneira mais detalhada para coletar informações sobre aplicações paralelas, e também a mais geral. Essa é uma técnica de coleta interna, com a qual é possível coletar informações importantes sobre o comportamento da aplicação (SCHNORR, 2014b). É utilizada para medir tempos de comunicação, registrar eventos de emissão e recepção, exibir gargalos, e registrar em que partes da aplicação os processos gastam mais tempo executando (PIOLA, 2007).

As informações coletadas a respeito da aplicação são gravadas em arquivos, esses arquivos contém os registros de eventos que representam as ocorrências mais significativas na execução da aplicação, como envio de mensagens e chamadas a procedimentos (PIOLA, 2007). Cada registro contém o identificador do evento, a data e o local onde o evento ocorreu, pode conter por exemplo o número da linha no código fonte, entre outras informações (PIOLA, 2007).

É utilizada para reconstruir o comportamento original da aplicação a partir das informações coletadas pelo rastreamento (SCHNORR, 2014b). Quanto maior for a quantidade de evento coletados pelo rastreamento, mais preciso ele será e assim, mais fiel será a reconstrução do comportamento original da aplicação a partir desses rastros (SCHNORR, 2014b). Essa é técnica de coleta de dados que apresenta a maior quantidade de dados em relação às outras técnicas (PIOLA, 2007). Essa é sua principal desvantagem, pois um volume de dados muito grande pode tornar difícil de interpretar (PIOLA, 2007).

3.2 Técnicas Utilizadas com o Algoritmo Genético

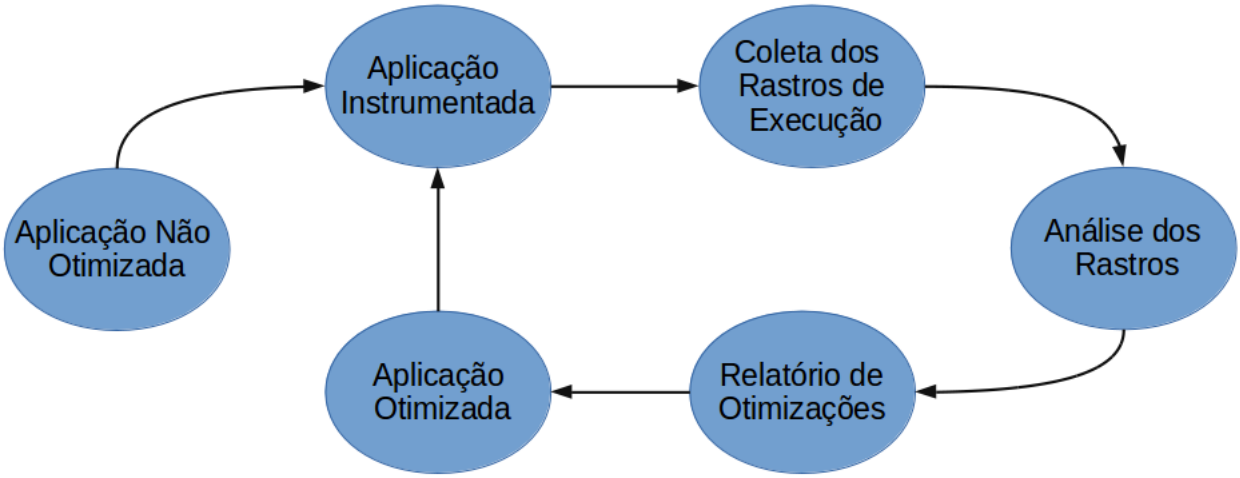
Para analizar o AG sequencial com objetivo de determinar quais regiões poderiam ser paralelizadas, Gressler e Cera (2014) o utilizou a técnica de amostragem para obter o perfil da execução (*profiling*) da aplicação. Para realizar a amostragem foi utilizada a ferramenta `gprof`, que é uma ferramenta livre e já vem incorporada no GCC. Para utilizar o `gprof`, basta acrescentar a opção `pg` à linha de compilação e assim, sempre que o programa gerado após a compilação for executado é criado um arquivo contendo o perfil da execução. A instância do PRV utilizada por Gressler e Cera (2014) na análise foi a c100, e a partir do o perfil da execução do AG sequencial, com o qual foi possível obter informações como o número de chamadas a uma determinada função e o tempo que leva

para executá-la. Iremos repetir essa análise para verificar se as informações obtidas se mantêm independente da arquitetura utilizada.

Para coletar o tempo de execução do AG, Gressler e Cera (2014) utilizou a técnica de cronometragem. Para cada uma das instâncias testadas Gressler e Cera (2014) cronometrou o tempo da versão paralela do AG utilizando 2, 3 e 4 *threads*, e também de sua versão sequencial. A partir dos valores dos *Speedup* Gressler e Cera (2014) concluiu que o desempenho da paralelização do AG ficou abaixo do ideal. Utilizaremos essa mesma técnica para coletar o tempo de execução da versão sequencial e paralelo do AG, e a partir destes calcularemos o *Speedup* para verificar o desempenho na arquitetura que iremos utilizar.

O objetivo deste trabalho é descobrir os motivos que levaram ao desempenho longe do ideal da paralelização do AG implementado por Gressler e Cera (2014) utilizando a técnica de rastreamento, pois essa é a maneira mais detalhada para coletar informações sobre o comportamento de aplicações paralelas (SCHNORR, 2014b). A Figura 8 ilustra as etapas necessárias para otimizar uma aplicação a partir do uso da técnica de rastreamento. Inicialmente se instrumenta uma aplicação utilizando uma ferramenta para a coleta de rastros de execução, a aplicação instrumentada é então executada para coletar os rastros. Os rastros coletados são analisados utilizando a técnica de Análise Interativa por Visualização (SCHNORR, 2014b), e a partir dessa análise é possível gerar um relatório contendo as modificações necessárias para otimizar essa aplicação. O processo de otimização pode ser executado quantas vezes forem necessárias, para que a aplicação já

Figura 8 – Etapas da Otimização de uma Aplicação Utilizando a Técnica de Rastreamento de Execuções.



Fonte: Adaptado de VI-HPS (2016).

otimizada continue a ser melhorada.

3.3 Análise Interativa por Visualização de Rastros

Existem diversas técnicas para realizar a análise de desempenho de aplicações paralelas. Segundo (SCHNORR, 2014b) um dos objetivos da análise de desempenho de uma aplicação paralela é o de melhorar o desempenho dessa aplicação, o qual pode ser medido através do tempo de execução da aplicação, da aceleração e da eficiência. O outro objetivo é o de aumentar a eficiência da utilização dos recursos computacionais. Logo, as técnicas de análise de desempenho devem conseguir informações sobre a execução da aplicação, com as quais seja possível melhorar os índices de desempenho, e identificar se há uma má utilização dos recursos computacionais disponíveis e qual a razão (SCHNORR, 2014b). Segundo (SCHNORR, 2014b) a grande parte dessas técnicas é inspirada diretamente nas formas de coleta de dados comportamentais. É possível por exemplo, que o analista realize a análise através de índices estatísticos calculados a partir de dados obtidos por contadores (SCHNORR, 2014b). Esse trabalho tem por objetivo coletar rastros de execução da paralelização de um AG aplicado PRV, logo, iremos utilizar a técnica de análise interativa por visualização de rastros para analisá-los.

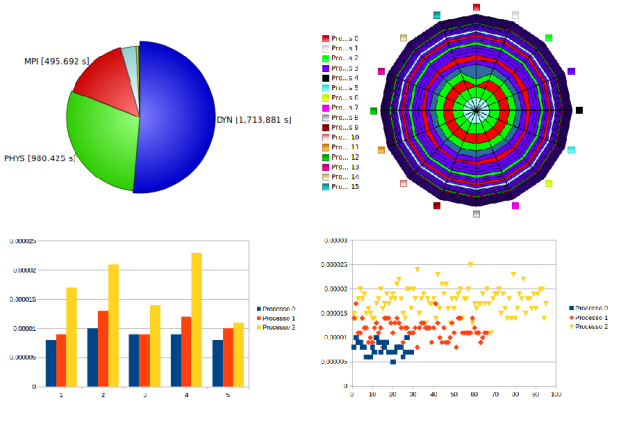
Segundo Schnorr (2014b) a técnica de análise interativa por visualização de rastros é uma das mais antigas técnicas de análise de aplicações paralelas e distribuídas. Essa técnica consiste em transformar os rastros coletados da aplicação paralela em representações visuais intuitivas, essas representações visuais colocam em evidência padrões que podem ser facilmente detectados pelo analista ao analisar a representação (SCHNORR, 2014b).

Segundo Schnorr (2014b) as técnicas de visualização de rastros podem ser divididas em três grupos de acordo com as características da representação dos dados (Figura 9).

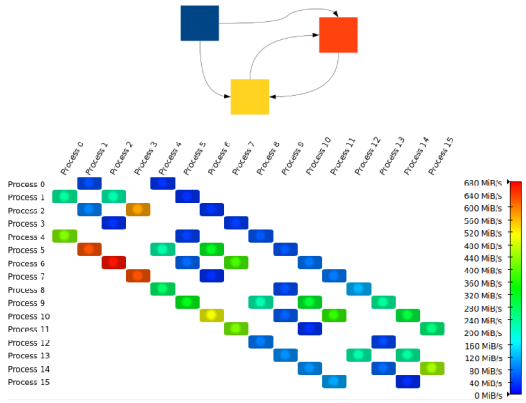
- **Visualização Estatística**: Essa técnica de visualização é a maneira mais comum de representar dados (Figura 9(a)). Ela agrupa todas as representações tradicionais, como gráficos de estatísticos de dispersão, com duas ou mais variáveis correlacionadas; gráficos de pizza, gráficos de funções e de processos. Outras formas de representações também podem ser utilizadas, como os diagramas de *kiviat* e representações estatísticas em três dimensões.

- **Visualização Estrutural**: Essa técnica de visualização ilustra a estrutura da aplicação paralela ou do sistema computacional de execução (Figura 9(b)). As técnicas de visualização baseadas na topologia de rede são classificadas com técnicas de visualização estruturais, entre elas estão, a representações com três dimensões de ambientes em grade com a interconexão da rede, a representações de grafo de cha-

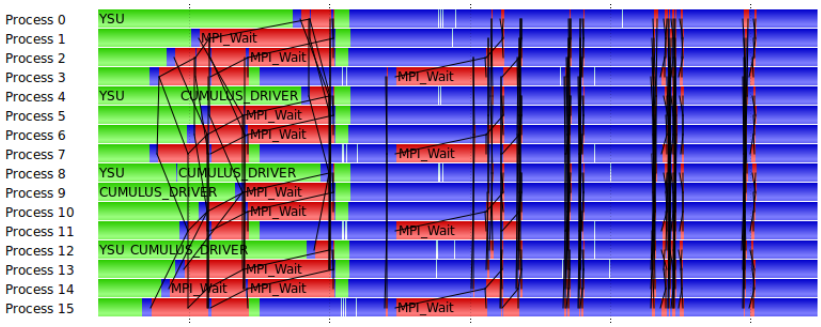
Figura 9 – Exemplos de Visualização de Rastros de Execução.



(a) Visualização Estatística.



(b) Visualização Estrutural.



(c) Visualização Comportamental.

Fonte: Adaptado de GWT-TUD GmbH (2016).

madas e matrizes de comunicação. O diferencial da visualização estrutural é que ela é independente de uma linha de tempo, onde elas consideram os valores em um dado instante de tempo ou dados agregados temporalmente.

- **Visualização Comportamental**: Nessa técnica de visualização os dados são desenhados ao longo de uma linha temporal. O exemplo mais conhecido e intuitivo dessa representação é o gráfico de espaço-tempo, como pode ser observado na Figura 9(c). No gráfico de espaço-tempo o eixo vertical contêm todas as entidades observadas (processos, fluxos de execução, etc.), podendo estar organizadas hierarquicamente. Horizontalmente é desenhado o comportamento de cada uma dessas entidades, onde retângulos representam os estados da aplicação e flechas representam as possíveis interações entre essas entidades.

3.3.1 Ferramentas Para Análise Interativa por Visualização de Rastros

Existem várias ferramentas utilizadas tanto para coletar os rastros de execução de programas paralelos, quanto para analizar esses rastros. A Tabela 2 apresenta as principais

Tabela 2 – Ferramentas para coleta de rastros de execução.

Ferramenta	**API**	**Recompilar**	**Formato do rastro**	**Preço**
Akypuera	*Message Passing Interface* (MPI)	não	*Paje File Format* (Pajé)	Gratuíta
EZTrace	MPI e OpenMP	não	Pajé e *Open Trace Format* (OTF)	Gratuíta
Score-P	MPI e OpenMP	sim	*Open Trace Format* Versão 2 (OTF2)	Gratuíta
TAU	MPI e OpenMP	sim	slog-2	Gratuíta

Fonte: Schnorr (2014a), Trahay et al. (2011), VI-HPS (2016), Shende e Malony (2006).

ferramentas para a coleta de informações a respeito da execução de aplicações paralelas: Akypuera (SCHNORR, 2014a), *Easy To use Trace Generator* (EZTrace) (TRAHAY et al., 2011), Score-P (VI-HPS, 2016) e *Tuning and Analysis Utilities* (TAU) (SHENDE; MALONY, 2006). Como podemos observar a ferramenta Akypuera é a única que não permite rastrear as execuções de aplicações paralelizadas com OpenMP. A ferramenta EZTrace (TRAHAY et al., 2011) é a única que não possui a necessidade de recompilar o código fonte da aplicação a ser estudada. Todas as ferramentas são gratuítas e não há necessidade de modificar o código fonte da aplicação para utilizá-las.

Para realizar esse trabalho vamos utilizar a ferramenta Score-P. A ferramenta Score-P possui uma grande vantagem em relação a outras ferramentas da área, ela oferece uma certa padronização. Dessa maneira permite-se que os dados coletados através do Score-P possam ser analisados por várias ferramentas, como por exemplo, Periscope (BENEDICT; PETKOV; GERNDT, 2010), Scalasca (GEIMER et al., 2010), Vampir (GWT-TUD GmbH, 2016), *Visual Trace Explorer* (ViTE), entre outras.

A Tabela 3 apresenta as principais ferramentas para a visualização de rastros de execução de aplicações paralelas: Pajé (PAJÉ, 2013), *Paje Next Generation* (PajeNG) (SCHNORR, 2012), Vampir (GWT-TUD GmbH, 2016), ViTE (COULOMB et al., 2010) e Viva (SCHNORR, 2014c).

Tabela 3 – Ferramentas para visualização de rastros de execução.

Ferramenta	**Formato do rastro**	**Preço**
Pajé	Pajé	Gratuíta
PajeNG	Pajé	Gratuíta
Vampir	OTF e OTF2	R$ 3607,82 à R$ 151832.1000
ViTE	Pajé, OTF e TAU	Gratuíta
Viva	Pajé	Gratuíta

Fonte: Pajé (2013), Schnorr (2012), GWT-TUD GmbH (2016), Coulomb et al. (2010), Schnorr (2014c).

Nossa primeira tentativa em visualizar os rastros gerados foi utilizando a ferramenta ViTE. Porém, o formato de rastro visualizado por essa ferramenta não é compatível com o gerado pela Score-P. Dessa forma, realizamos uma conversão do rastro para o formato Pajé utilizando a ferramenta Akypuera. Porém, a visualização com o ViTE não era compreendível, dificultando a análise. Outra alternativa para a visualização dos rastros era a de utilizar o pacote ggplot2 (WICKHAM, 2013), que é um sistema de plotagem de gráficos estatísticos para linguagem R (MATLOFF, 2011), baseado na gramática dos gráficos. Porém, a curva de aprendizagem da linguagem R excede o tempo limite para a finalização deste trabalho.

A alternativa escolhida foi a de utilizar a ferramenta Vampir (GWT-TUD GmbH, 2016), a qual possui uma interface amigável e intuitiva. A principal vantagem em utilizar o Vampir é que ela permite visualizar rastros no formato OTF2, compatível com o formato gerado pelo Score-P. Dessa forma não é necessário realizar conversão no formato dos rastros.

3.3.1.1 Score-P

Score-p é uma ferramenta para rastrear a execução de aplicações de alto desempenho (VI-HPS, 2016) escritas nas linguagens de programação C, C++ e Fortran (KNUPFER et al., 2012). Essa ferramenta tem por objetivo simplificar a análise de aplicações paralelas e assim, permitir que os desenvolvedores desse tipo de aplicação descubram onde estão os problemas de desempenho e qual o motivo deles (VI-HPS, 2016).

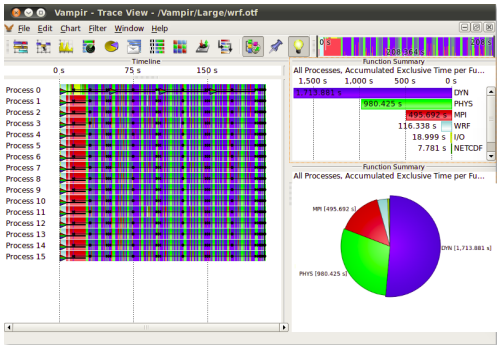
O Score-P suporta diferentes paradigmas de programação. Suporta paradigmas com múltiplos processos (MPI e *Symmetric Hierarchical Memory* (SHMEM)), com múltiplas *threads* (OpenMP e Pthreads), paradigmas baseados em aceleração (*Compute Unified Device Architecture* (CUDA) e *Open Computing Language* (OpenCL)), além de combinações entre eles (VI-HPS, 2016). Como o AG foi paralelizada utilizando a API OpenMP, nesse caso serão coletados os eventos referentes aos construtores paralelos do OpenMP.

É uma ferramenta fácil de usar para criação de perfis de execução através de técnicas de amostragem, para rastreamento de eventos e análise *on-line* (KNUPFER et al., 2012). No Score-P os dados de amostragem são gravados no formato CUBE4 e os dados de rastreamento são gravados no formato recém desenvolvido OTF2 (VI-HPS, 2016), o qual é baseado nos formatos OTF e EPILOG, formatos nativos das ferramentas Vampir e Scalasca, respectivamente (KNUPFER et al., 2012). Como esse trabalho tem por objetivo coletar rastros de execução da paralelização de um AG, logo, os dados por nós coletados serão salvos no formato OTF2.

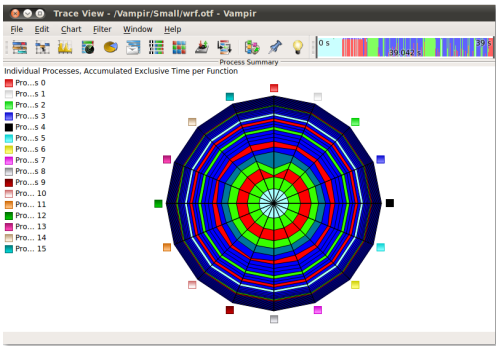
Score-P está em sua versão 3.0, que está disponíveis para *download* na *internet*[1].

[1] Disponível em <http://www.vi-hps.org/upload/packages/scorep/scorep-3.0.tar.gz>

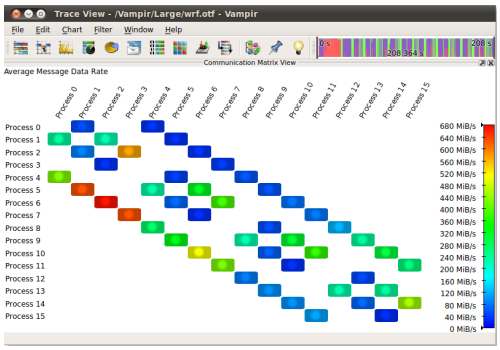
Figura 10 – Visualização com Vampir.



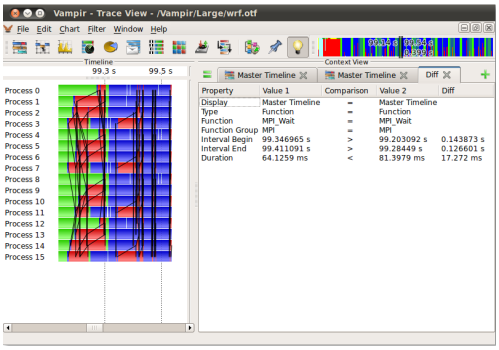
(a) Visualização Comportamental com Gráfico de Espaço/Tempo e Estatística com Gráfico de Pizza.



(b) Visualização comportamental com gráfico *kiviat*.



(c) Visualização estrutural com matriz de comunicação.



(d) Visualização da Comparação de Rastros de duas Aplicações.

Fonte: GWT-TUD GmbH (2016).

A versão do Score-P utilizada neste trabalho foi a 2.0.2, a qual não está mais disponível. Após realizar o *download* e a instalação, basta acrescentar a opção `scorep` à linha de compilação e habilitar o rastreamento a partir do `prompt` utilizando o comando `export SCOREP_ENABLE_TRACING=true`. Após a execução da aplicação é criado um um diretório contendo os eventos rastreados e o arquivo OTF2 com os rastros de execução.

3.3.1.2 Vampir

Vampir é uma ferramenta para a visualização de rastros de execução portátil que permite uma analise detalhada do desempenho de aplicaçoes paralelas (GWT-TUD GmbH, 2016) e é compartível com o formato de rastros gerado com Score-P. A ferramenta Vampir está em sua versão 9.1, a qual é paga. Logo, utilizamos sua versão demonstração 9, porém, esta possui limitações quanto ao número de eventos que podem ser visualizados. Para realizar o *download* da versão de demonstração basta preencher o formulário disponível no site oficial da ferramenta[2].

Essa ferramenta possui várias técnicas de visualização de rastros disponíveis, como é possível observar na Figura 10. É possível visualizar os rastros a partir da visualiza-

[2] Disponível em <https://www.vampir.eu/downloads/demo>

ção comportamental utilizando gráficos de espaço-tempo e de visualização estáticas Figura 10(a), com gráficos *kiviat* Figura 10(b) e com a matriz de comunicação Figura 10(c). Também é possível realizar a comparação entre dois rastros de execução diferentes Figura 10(d).

3.4 Considerações Sobre os Mecanismos para Análise de Desempenho

Nesse capítulo foram apresentados conceitos sobre a etapa de análise de desempenho de programas paralelos, a qual tem por objetivo identificar as regiões do programa que realizam uma baixa exploração dos recursos computacionais disponíveis. Vimos que a etapa de análise de desempenho se divide em duas fases, a fase de coleta dos dados sobre o comportamento da aplicação paralela e a fase de análise desse dados.

Nesse capítulo também foram apresentadas diferentes técnicas que são utilizadas para realizar a coleta de dados, dentre elas o rastreamento, que é a técnica a ser utilizada neste trabalho. No rastreamento as informações a respeito da aplicação são coletadas e gravadas em arquivos contendo os registros de eventos que representam as ocorrências mais significativas na execução da aplicação. A técnica utilizada para analizar os dados coletados a partir de rastreamento é a técnica de análise interativa por visualização de rastros, a qual consiste em transformar os rastros coletados da aplicação paralela em representações visuais intuitivas.

Vimos que existem diferentes ferramentas tanto para coletar os dados a respeito da aplicação paralela, quanto para analizá-los. A ferramenta que será utilizada na fase de coleta de dados é a Score-P, que é uma ferramenta fácil de usar para criação de perfis de execução, para rastreamento de eventos e análise *on-line*. Para fase de análise dos rastros coletados utilizaremos a ferramenta Vampir, por ser uma ferramenta compatível com o formato de rastro gerado pelo Score-P.

O próximo capítulo apresenta os resultados obtidos pela análise do *profilling*, do tempo de execução e *Speedup* da execução do AG em uma arquitetura diferente das utilizadas anteriormente por Gressler e Cera (2014) e Rosa e Cera (2015a). São apresentados detalhes da arquitetura utilizada e as configurações utilizadas no AG. Também são apresentados os resultados referentes aos rastros de execução coletados com Score-P.

4 Análise dos Resultados

Nesse capítulo iremos apresentar os resultados obtidos, ele está organizado da seguinte maneria. A seção 4.1 apresenta as características da arquitetura utilizada para realizar a execução do AG e os parâmetros utilizados. A seção 4.2 apresenta os resultados obtidos pela execução do AG com a ferramenta `gprof` na arquitetura utilizada. A seção 4.3 apresenta a análise do desempenho do AG paralelo na arquitetura utilizada. As seções 4.4 e 4.5 apresentam a análise do uso de `sections` e `parallel for` na paralelização do AG. A seção 4.6 apresenta as considerações sobre esse capítulo.

4.1 Ambiente de Execução

A coleta dos dados foi realizada em uma *Workstation* da UNIPAMPA - Campus Alegrete. Ela possui um processador Intel® Xeon® E5-2603 v3, com frequência de 1.9 GHz e 6 núcleos físicos, sem suporte a tecnologia *Hyper-Threading*. Possui 3 níveis de *cache*, com *cache* L1 de dados e instruções ambas com 32 KB, *cache* L2 com 256 KB e *cache* L3 com 15 MB, e memória RAM DDR 4 de 16 GB. O sistema operacional utilizado é o Ubuntu 16.04 LTS compilador GCC em sua versão 5.31.

4.1.1 Parâmetros do Algoritmo Genético para as instâncias alvo

A instâncias do PRV utilizadas neste trabalho fazem parte do do *Benchmark* de Christofides et al. (1979), o qual reúne 14 instâncias do PRV com número de cidades variando entre 50 e 199. Nesse *Benchmark* metade das instâncias possuem restrição quanto a capacidade do veículo e a outra metade, além de possuir essa restrição quanto a capacidade do veículo, também possui restrições quanto a distância percorrida por rota. Gressler e Cera (2014). Serão utilizadas quatro instâncias com restrição da capacidade de transporte do veículo, as quais estão disponíveis para *download* na *internet*[1]. As instâncias a serem utilizadas são c50, c100, c120 e c150, que consistem em percursos com respectivamente 50, 100, 120 e 150 cidades (ROSA; CERA, 2015a).

Baseando-se nos resultados de desempenho obtidos por Gressler e Cera (2014) os parâmetros definidos para o AG que obtiveram o maior desempenho foram:

- **Tamanho da População:** N x 10, onde N é o número total de cidades;
- **Número de Evoluções:** N x N x 10 evoluções;

[1] Disponível em <http://mistic.heig-vd.ch/taillard/problemes.dir/vrp.dir/vrp.html> (último acesso em junho de 2016).

- **Técnica de Cruzamento:** Cruzamento de 1 ponto. Nesse tipo de cruzamento escolhe-se um ponto em dois cromossomos e intercala-se as seções de gene resultantes gerando dois novos indivíduos;
- **Probabilidade de Mutação:** ocorre mutação em 20% das vezes;
- **Taxa de Mutação:** variável entre 4 a 10% dos genes dos cromossomos;
- **Técnica de Mutação:** Randômica. Essa técnica sorteia quais das técnicas de mutação serão utilizadas entre as técnicas de troca (troca um gene de lugar), troca bloco (troca um bloco de genes), inversão (seleciona e inverte uma seção de genes) e inserção (remove uma seção de genes e a insere em outro lugar do mesmo cromossomo).

4.1.2 Parâmetros para o OpenMP

A arquitetura utilizada possui um processador com 6 cores físicos, logo, iremos executar com 2, 3, 4, 5, 6, 7 e 8 *threads*, além da versão sequencial para testar o potencial da arquitetura.

Baseando-se nas conclusões expostas na subseção 2.2.3.2, definimos para o OpenMP a política de distribuição de iterações como *Static*. Pois essa política levou ao maior desempenho devido ao AG possuir uma carga regular, onde todas as iterações do laço demandam aproximadamente o mesmo tempo de processamento. Dessa forma é mais eficiente distribuir as iterações do laço de forma igualitária entre todas as *threads*, em tempo de compilação sem acarretar em custo adicional para a distribuição da carga. Logo, políticas dinâmica (ou em tempo de execução) como *Dynamic* e *Guided* não fornecem ganhos.

4.2 Análise da determinação das regiões paralelas

Para determinar quais regiões podiam ser paralelizadas Gressler e Cera (2014) executaram o AG sequencial com a instância c100 utilizando a ferramenta `gprof`. A Tabela 4 apresenta o resultado da execução do AG sequencial realizada por Gressler e Cera (2014). Repetimos para verificar o comportamento da aplicação na arquitetura utilizada.

A Figura 11 apresenta execução do AG sequencial com o `gprof` que realizamos na forma de um grafo de chamadas. Esse grafo foi gerado utilizando um *script* em Python disponível para *download* na *internet*[2]. Esse *script* transforma o perfil de execução gerado pelo `gprof` em um *script* na linguagem `dot` nativo da ferramenta *Graph Visualization Soft-*

[2] Disponível em <https://github.com/jrfonseca/gprof2dot> (último acesso em novembro de 2016).

Tabela 4 – Impacto das funções no tempo total de execução para a instância c100, obtido pelo *gprof*.

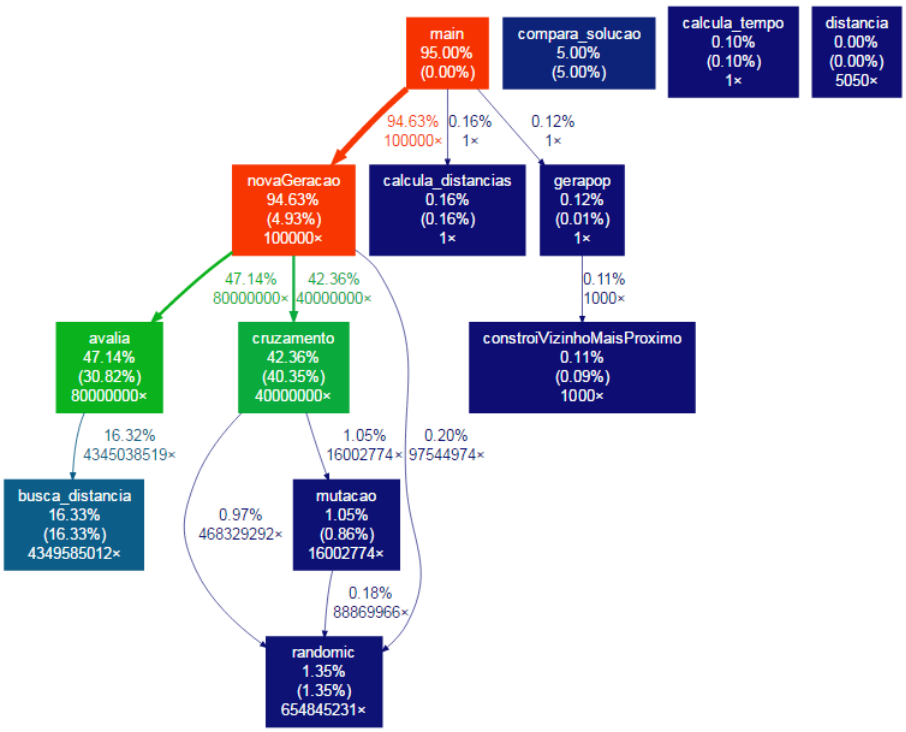
Função	Número de Chamadas	Soma das Chamadas (s)	% do Tempo Total
Cruzamento	40.000.000	179,99 s	37,42%
Busca distância	4.349.590.210	125,49 s	26,09%
Avalia	80.000.000	122,05 s	25,37%
Nova Geração	100.000	23,72 s	4,93%
Compara Solução	-	14,44 s	3,00%
4,76 Número randômico	769.795.125	7,53 s	1,58%
Calcula 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%

Fonte: Gressler (2013)

ware (Graphviz) (LABS, 2014). O Graphviz é uma ferramenta para criação e visualização de grafos a partir de *script* escrito na linguagem `dot`.

Observando a Figura 11 vemos que as funções que mais impactam no tempo total da execução são: `Cruzamento` representando 40,35% do tempo total da execução com 40.000.000 chamadas, `Avalia` representando 30,82% do total com 80.000.000 cha-

Figura 11 – Grafo de Chamadas das Funções do AG Sequencial para a Instância c100.



madas e `Busca Distância` representando 16,33% do total com 4.349.585.012 chamadas. Comparando nossos resultados com os obtidos por Gressler e Cera (2014), vemos que o comportamento da aplicação se mantêm apesar da arquitetura utilizada, onde as funções que mais impactam no tempo total da aplicação permanecem a `Cruzamento`, `Busca Distância` e `Avalia`. Essas funções possuem um grande número de chamadas, logo uma execução dessas funções representa uma porcentagem muito pequena do tempo total de execução.

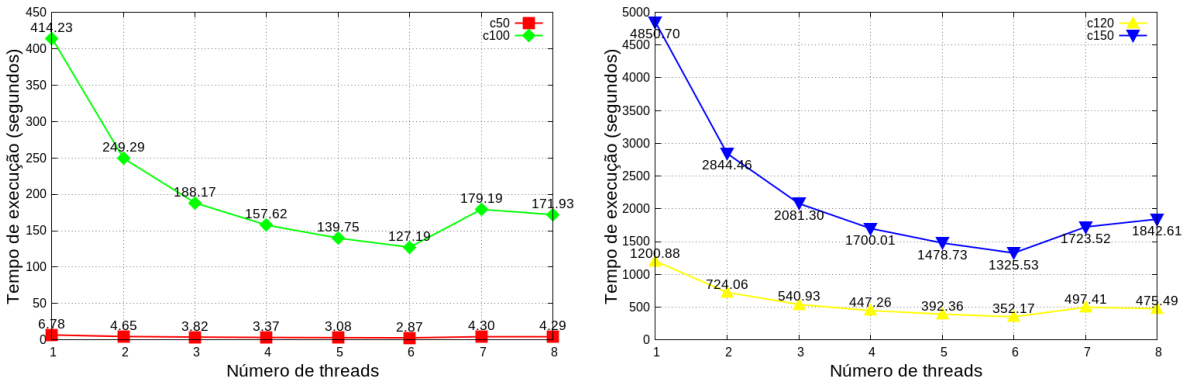
Assim como o perfil da execução obtido por Gressler e Cera (2014) (Tabela 4) identificamos que a função `Nova Geração` recebeu 100.000 chamadas, as quais representaram apenas 4,93% do tempo total de execução. Essa função é chamada apenas uma vez a cada geração, então sua chamada não foi paralelizada. Porém, ela realiza chamadas diretas às funções `Avalia` e `Cruzamento` e indiretas à função `Busca Distância`, as quais mais impactam no tempo total de execução. Justificando a escolha de Gressler e Cera (2014) em paralelizar as instruções da função `Nova Geração`.

4.3 Análise de desempenho

Para analisar o desempenho do AG paralelo, executamos com 2, 3, 4, 5, 6, 7 e 8 *threads*, além da versão sequencial, para testar o potencial da arquitetura. Foram realizadas no mínimo 30 execuções para cada configuração. Dessas 30 execuções foram retiradas as 10 mais rápidas e as 10 mais demoradas. Os resultados discutidos nesta seção representam a média dos tempos de execução de amostras das 10 execuções restantes, sendo que seu desvio padrão ficou sempre abaixo de 0,5%. Para nos auxiliar na análise dos resultados, foi calculado o *Speedup* da média dos tempos de execução, que é a medida de quão rápido um programa paralelo é em relação a sua versão sequencial (PACHECO, 2011).

A Figura 12 apresenta os gráficos do tempo de execução (em segundos) para as instâncias alvo conforme o aumento do número de *threads*, onde os menores tempos de execução foram de 2,87, 127,19, 352,17 e 1325,53 segundos para as instâncias c50, c100, c120 e c150, respectivamente. Sendo que para todas as instâncias, os menores tempos de execução foram obtidos com 6 *threads* e consequentemente os maiores *speedups* (Figura 13(a)), na faixa de 2,37 à 3,66. Logo, para todas as instâncias o uso de 6 *threads* levou a geração de tarefas com granularidade mais adequada e consequentemente o melhor desempenho, confirmando que o melhor desempenho é atingido quando utilizam-se todos os núcleos do processador. Observando a Figura 13(a) podemos perceber que o *speedup* aumenta conforme executa-se instâncias com maiores cargas computacionais. Porém, todos os *speedups* ficaram abaixo do ideal, sendo que os *speedup* mais próximos do ideal foram obtidos com 2 *threads*.

Figura 12 – Tempo de Execução (em Segundos) do AG para as Instâncias Alvo do PRV Conforme o Aumento do Número de *Threads* Utilizadas.



(a) Tempo de Execução (em Segundos) das Instâncias c50 e c100 Conforme o Aumento do Número de *Threads*.

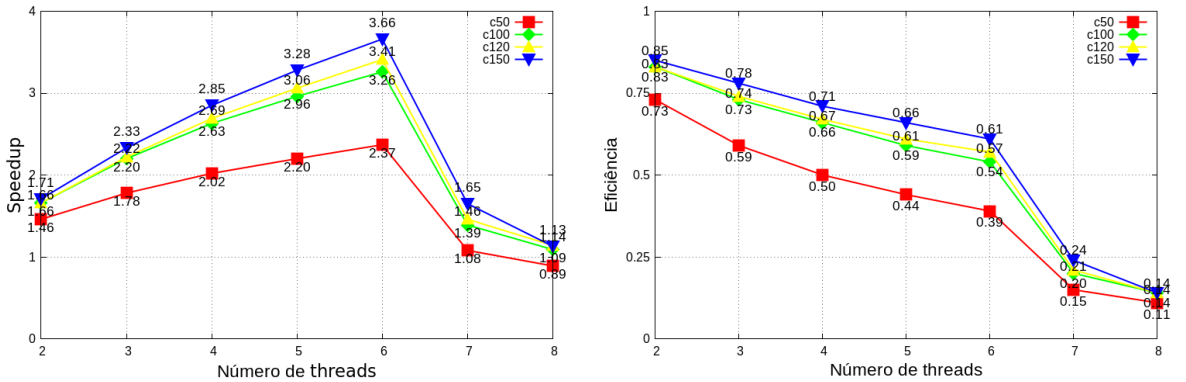
(b) Tempo de Execução (em Segundos) das Instâncias c120 e c150 Conforme o Aumento do Número de *Threads*.

Fonte: Adaptada de Andrade e Cera (2016a, p. 072)

A Figura 13(b) apresenta o gráfico da eficiência obtida pela paralelização do AG. Observando esse gráfico podemos ver que a eficiência obtida está na faixa de 0,39 à 0,61. Pode-se perceber que a eficiência da paralelização diminui conforme aumenta-se o número de *threads* utilizadas. Estima-se que isso ocorre devido ao *overhead* causado na criação de um número maior de *threads* e por uma distribuição de carga ineficiente em função da característica da implementação do AG.

A Figura 14 apresenta os melhores *Speedups* obtidos por Gressler e Cera (2014), Rosa e Cera (2015a) e os nossos resultados para as instâncias alvo (ANDRADE; CERA, 2016a). Os melhores *Speedups* obtidos por Gressler e Cera (2014) para as instâncias c50,

Figura 13 – *Speedup* e Eficiência Obtidos pelas Instâncias Alvo Conforme o Aumento do Número de *Threads* utilizadas.

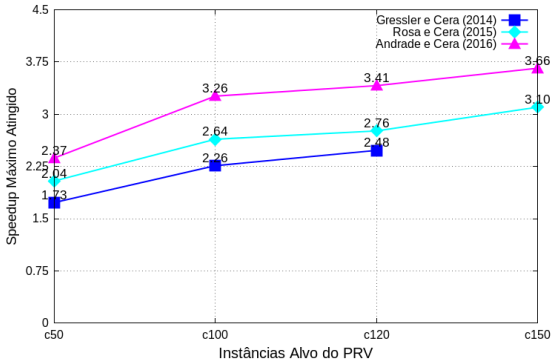


(a) Speedup.

(b) Eficiência.

Fonte: Adaptada de Andrade e Cera (2016a, p. 72)

Figura 14 – *Speedups* Máximos Obtidos por Gressler e Cera (2014), Rosa e Cera (2015b), Andrade e Cera (2016a)



Fonte: Gressler e Cera (2014), Rosa e Cera (2015b), Andrade e Cera (2016a).

c100 e c120, foram utilizando 4 *threads*, esses *Speedups* estão na faixa de 1,73 à 2,48. Os melhores *Speedups* obtidos por Rosa e Cera (2015a) para as instâncias c50, c100, c120 e c150, foram utilizando 8, 16, 16 e 32 *threads* respectivamente, representam *Speedups* na faixa de 2,04 à 3,10. Os melhores *Speedups* por nós obtidos para as instâncias c50, c100, c120 e c150, foram utilizando 6 *threads*, esses *Speedups* estão na faixa de 2,37 à 3,66. Considerando que arquitetura utilizada por Gressler e Cera (2014) possui um processador com 4 cores, a utilizada por Rosa e Cera (2015a) possui 2 processadores cada um com 8 núcleos e suporte a tecnologia *Hyper-Threading* e a arquitetura que utilizamos possui um processador com 6 núcleos. Pode-se observar que houve um aumento no desempenho quando comparado aos resultados obtidos por Gressler e Cera (2014) e Rosa e Cera (2015a). Nesta circustância atribui-se estre ganho às características da arquitetura utilizada, uma vez que os demais parâmetros mantiveram-se os mesmos.

Com base nesse análise podemos observar que o problema do baixo desempenho do AG paralelo persiste apesar da arquitetura. Logo, iremos realizar o rastreamento das execuções do AG paralelo, com o objetivo de identificar a existência de possíveis gargalos afetando no desempenho da aplicação.

4.4 Análise do uso de Sections

A região paralelizada com `sections` possui dois blocos de código que não possuem tarefas iterativas e podem ser executados em paralelo. Logo, cada bloco fica dentro de um `section` e cada `section` é executada por uma *thread* diferente. Neste caso, como são 2 blocos de código, as tarefas de cada bloco serão sempre executadas por 2 *threads*. Logo, nosso objetivo é avaliar o impacto do uso das *sections* utilizando mais de 2 núcleos.

Ao término da execução da região paralela ocorre uma sincronização implícita (*Join*), a partir da qual o programa segue sua execução apenas com a *thread* mestre. O tempo de espera em sincronização representa o tempo que uma *thread* que já terminou de executar espera até que todas as outras *threads* terminem suas tarefas para assim realizarem a sincronização (CHAPMAN; JOST; PAS, 2008). A Tabela 5 apresenta o tempo médio de espera em sincronização por *thread* e a porcentagem que esse tempo representa no tempo total de execução do AG paralelo para as instâncias c50 e c100, com e sem o uso das `sections`. A ferramenta Vampir gera em suas estatística o tempo acumulado que um construtor OpenMP demora para executar uma região do código, ou seja a soma de todas as reuniões executadas por esse construtor. Logo, o tempo médio de sincornização é calculado como:

$$\frac{Tempo\ de\ SincronizaçãoAcumulado}{Número\ de\ Threads}$$

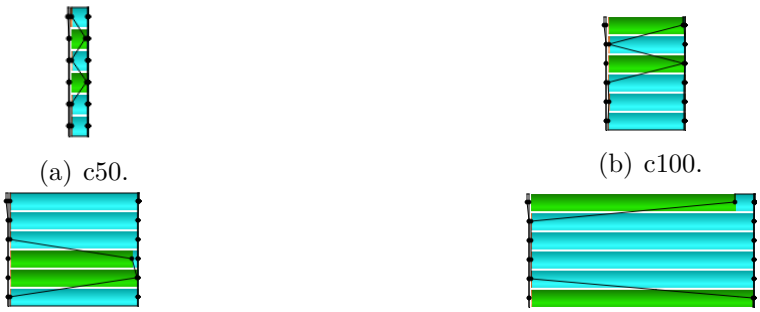
Através dos resultados obtidos percebe-se que o tempo de espera em sincronização aumenta conforme o aumento do número de *threads* utilizadas devido ao maior número de *threads* envolvidas em sincronização. Para a instância c50, de 2 para 6 *threads* houve um aumento de 2,75 vezes utilizando `sections` e de 1,33 vezes quando não se utiliza `sections`. Para a instância c100, o aumento no número de *threads* levou a um aumento de 2,97 vezes utilizando `sections` e de 1,90 vezes caso contrário. O tempo de espera por sincronização quando se utiliza `sections` é maior do que quando não se utiliza, para ambas as instâncias essa diferença passou do dobro do tempo de espera. Sendo que o maior impacto da utilização de `sections` é com 6 *threads*. Isso ocorre pois as `sections` são executadas por apenas 2 *threads*. Logo, quando se utiliza mais de 2 *threads*, apenas 2 *threads* estarão executando, enquanto que as demais não estarão realizando nenhum trabalho útil, apenas estarão aguardando o término da execução das `sections` para realizar a sincronização. Devido a limitações no número de eventos que podem ser visualizados pelo Vampir, a Tabela 5 apresenta somente as estatísticas das instâncias c50 e

Tabela 5 – Tempo de espera em sincronização (em segundos) com e sem o uso de `sections` e sua porcentagem no tempo total de execução da aplicação.

Threads	**c50**				**c100**			
	Com Sections		**Sem Sections**		**Com Sections**		**Sem Sections**	
	Tempo (s)	**% do total**	**Tempo (s)**	**% do total**	**Tempo (s)**	**% do total**	**Tempo (s)**	**% do total**
2	0,08	1,78%	0,06	1,31%	3,03	1,20%	1,84	0,71%
3	0,15	3,72%	0,06	1,50%	6,50	3,42%	1,98	1,02%
4	0,18	5,21%	0,06	1,88%	7,05	4,42%	2,00	1,23%
5	0,20	6,26%	0,10	3,25%	7,90	5,50%	1,92	1,34%
6	0,22	7,33%	0,08	4,10%	9,00	6,75%	3,49	2,62%

Fonte: Andrade e Cera (2016b).

Figura 15 – Execução da Região Paralelizada Utilizando `sections` com 6 *threads*.



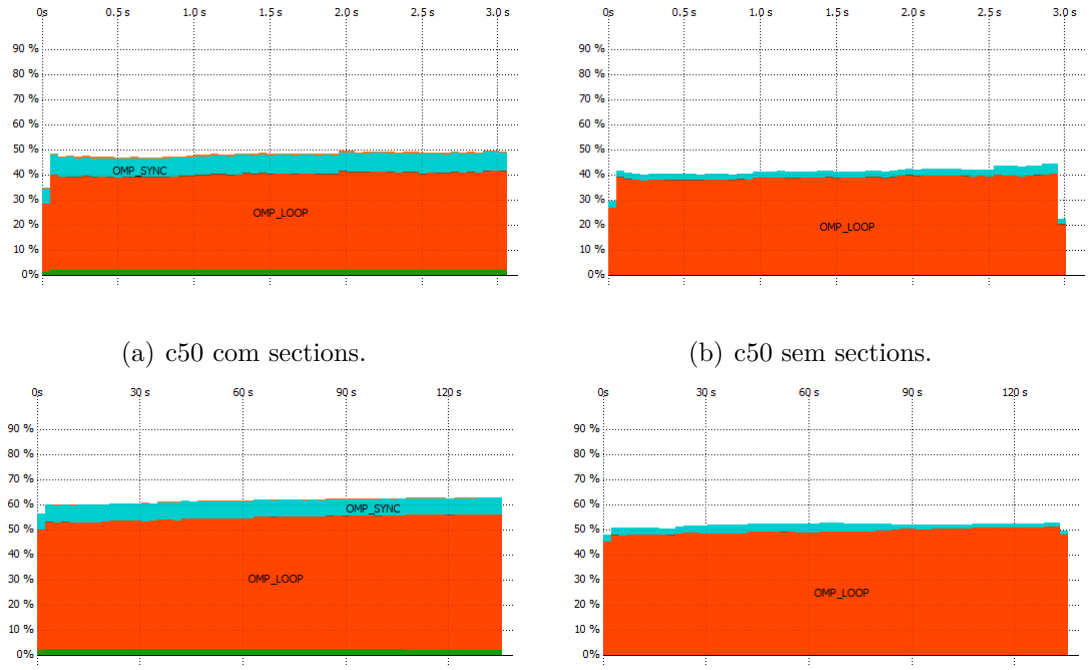
(a) c50. (b) c100.

(c) c120. (d) c150.

c100. Porém identificou-se que todas as instâncias apresentam o mesmo comportamento, como pode observado na Figura 15, onde a execução da `section` está representada em verde e a sincronização em azul. Através dessa figura podemos observar que as *sections* podem ser executadas por *threads* diferentes e uma *section* pode terminar sua execução antes da outra, como é o caso da Figura 15(d).

As Figuras 16 e 17 apresentam gráficos gerados pelo Vampir que ilustram o aumento do tempo de sincronização com 6 *threads* para cada uma das instâncias alvo. Onde a sincronização está representada em azul, a execução do `parallel for` em laranja, a execução das `sections` em verde e o espaço em branco representa o que é executado sequencialmente. Nesse gráfico o eixo x representa o intervalo do tempo de execução e

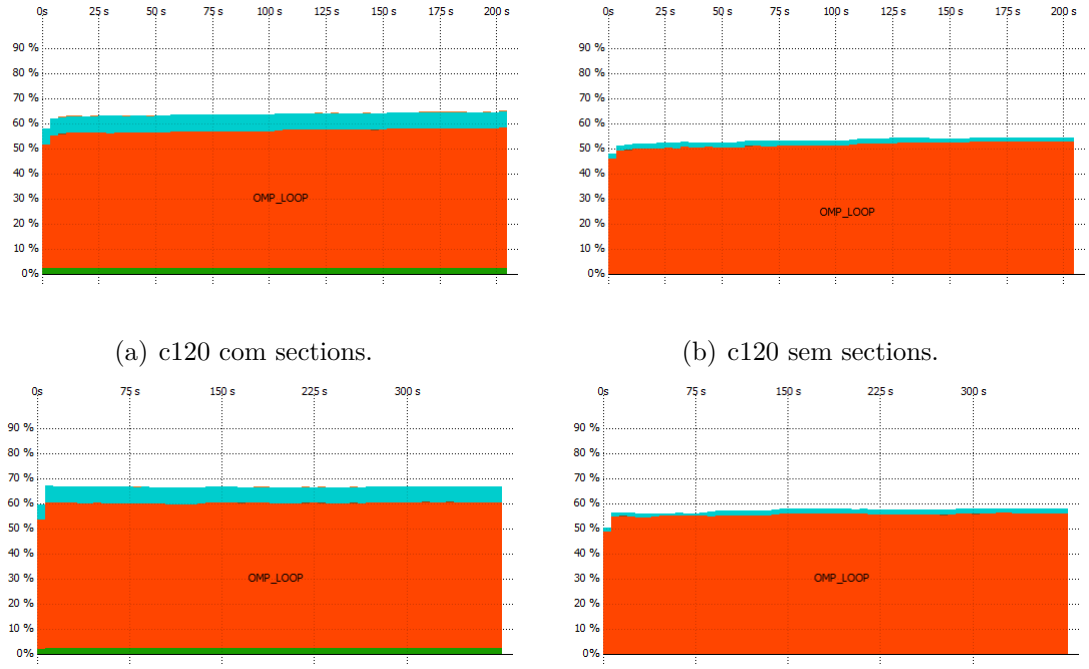
Figura 16 – Gráficos do Comportamento do AG Com e Sem o Uso de `sections` para as Instâncias c50 e c100 Utilizando 6 *threads*.



(a) c50 com sections. (b) c50 sem sections.

(c) c100 com sections. (d) c100 sem sections.

Figura 17 – Gráficos do Comportamento do AG Com e Sem o Uso de `sections` Para as Instâncias c120 e c150 Utilizando 6 *threads*.



(a) c120 com sections.

(b) c120 sem sections.

(c) c150 com sections.

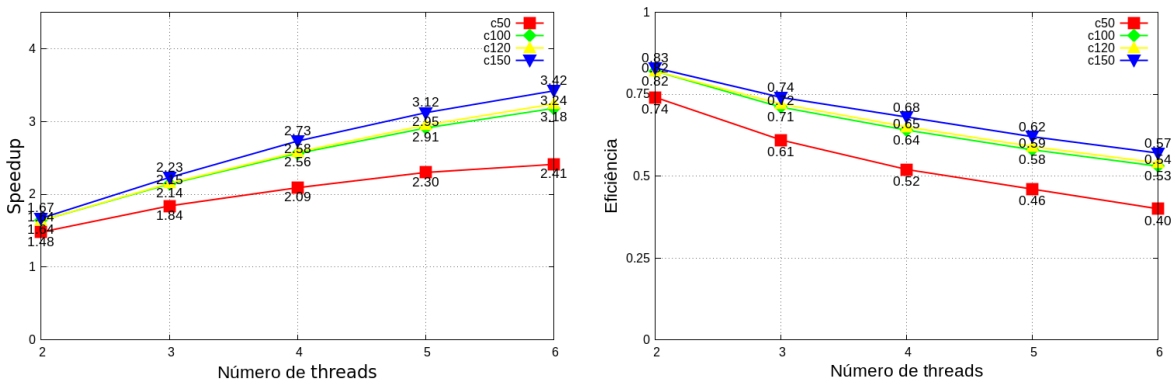
(d) c150 sem sections.

o eixo y representa a porcentagem que o tempo de execução de uma tarefa leva para executar nesse intervalo.

A Figura 16 apresenta os gráficos para as instâncias c50 e c100, onde é apresentado o comportamento de todo o tempo de execução. A Figura 17 apresenta os gráficos para as instâncias c120 e c150, onde o intervalo máximo de visualização para a instância c120 é de aproximadamente 200 segundos e para a instância c150 é de aproximadamente 370. Porém, essa limitação do intervalo de tempo não compromete os resultados. Observando as Figuras 16 e 17 podemos perceber que o tempo de sincronização utilizando *section* é no mínimo 2 vezes maior do que quando não se utiliza. Sendo que a instância c50 é a que sofre o maior impacto com tempo de sincronização. O tempo de sincronização diminui sem o uso de `sections`, é 2,75 vezes menor para a instância c50, 2,58 vezes menor para c100, 3,74 vezes menor para c120 e 7,00 vezes menor para a instância c150. Embora o tempo de sincronização seja menor é preciso verificar o impacto no desempenho.

A Figura 18 apresenta os gráficos do *speedup* (Figura 18(a)) e Eficiência (Figura 18(b)) obtidos ao executar sem o uso de `sections`. Observando a Figura 18(a) vemos que o maior desempenho é obtido com 6 *threads*, com *speedups* na faixa de 2,41 à 3,42. Assim como com o uso de `sections`, a eficiência do AG diminui com o aumento do número de *threads*. Ou seja, a existência *threads* ociosas, sem executar nenhum trabalho útil, diminui a eficiência da aplicação. Esse desperdício dos recursos ocasiona em uma sobrecarga de sincronização, como é mostrado nas Figuras 16 e 17.

Figura 18 – *Speedup* e Eficiência Obtidas Pela Execução do AG Paralelo Sem o Uso de `sections`.



(a) `Speedup`. (b) Eficiência.

A Figura 19 apresenta um gráfico comparando o melhor desempenho obtido pelo AG com e sem o uso de `sections`. Observando esse gráfico podemos perceber que a instância c50 é a única no qual o desempenho melhorou sem o uso de `sections`. Pois essa instância apresenta um tempo de execução sequencial pequeno, de 6,78 segundos, logo sofre um maior impacto com a sobrecarga de sincronização. Para as outras instâncias o uso de `sections` aumenta o desempenho do AG. Embora o uso de `sections` aumente o tempo de sincronização é melhor manter o uso das mesmas. Pois é preferível uma sobrecarga em tempo de sincronização do que a redução do desempenho da aplicação.

Figura 19 – *Speedups* Máximos Atingidos Com e Sem o Uso de `sections`.

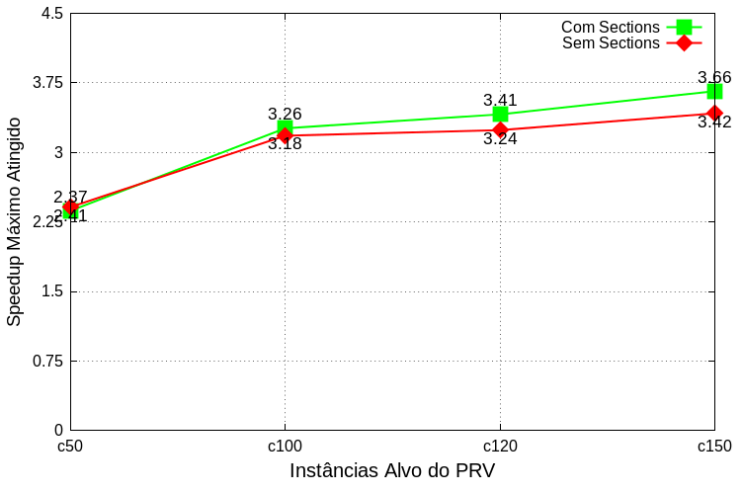
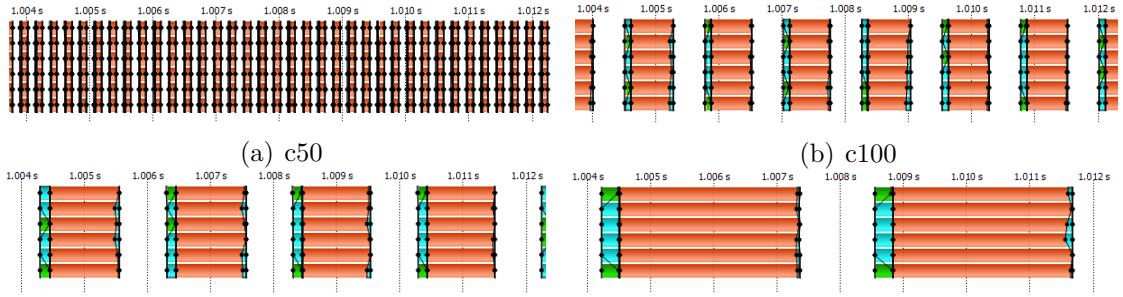


Figura 20 – Visualização do Rastro da Execução do AG Paralelo Gerada pelo Vampir



(a) c50 (b) c100

(c) c120 (d) c150

4.5 Análise do Parallel For

A Figura 20 ilustra o comportamento do AG paralelo quando executa um `parallel for` com 6 *threads* para as instâncias c50, c100, c120 e c150. Observando essa figura podemos ver que logo após a execução das `sections` ocorre um *Fork*, criando um conjunto de *threads* para a execução do `for` e logo após essa execução ocorre uma sincronização das *threads* (*Join*). Não há problemas na paralelização com o `parallel for`, sendo que todas as 6 *threads* o executam. Porém, identificamos a existência de uma região que não foi parelelizada entre o *Join* do *parallel for* e o *Fork* da `section`. Essa região é representada pela parte em branco na Figura 20.

O pseudocódigo abaixo ilustra o AG implementado por Gressler e Cera (2014). O

Algorithm 1 Algoritmo Genético

```
1: Início
2: Gera população inicial
3: Avalia todos os indivíduos
4: while ( !Terminou ) do
5:     Ordena a população
6:     Inicialização a estrutura da Nova População
7:     Copia indivíduos perpetuados para Nova População
8:     for (1/2 da taxa de cruzamento) do
9:         Seleciona 2 indivíduos
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:    end for
16:    Libera Memória
17:    if (Número de evoluções foi atingido) then
18:        Terminou ← Verdade
19:    end if
20: end while
21: Retorna Melhor Solução Encontrada
22: Fim
```

laço `while` é responsável por controlar o número de evoluções e é onde se concentram as regiões paralelas. Nesse pseudocódigo identificamos que entre um *Join* e um *Fork* ocorre a ordenação da população (linha 5), a inicialização da nova geração (linha 6) e a liberação da memória da estrutura que armazena a população (linha 16). Executamos novamente o AG sequencial utilizando o `gprof` para identificar o impacto dessas operações no tempo total de execução. A Tabela 6 apresenta o perfil de execução gerado para cada uma das instâncias alvo.

As funções `Inicializa Nova População` e `Libera Memória` possuem um impacto muito pequeno no tempo total de execução do AG sequencial, não chegando nem a 1% desse tempo. Logo, a paralelização dessas duas funções não fornecem ganhos de desempenho devido ao custo adicional gerado na criação e gerenciamento das *threads* (*overhead*).

A execução da função `Libera Memória` é muito importante para a execução correta da aplicação. Uma vez que a alocação realiza na memória não seja liberada, mesmo que o programa termine sua execução essa memória continuará alocada, ocasionando no chamado vazamento de memória. Em alguns casos o vazamento de memória faz com que o programa consuma toda a memória disponível e interrompa sua execução ocasionando uma falha (GRIFFITHS, 2013).

O algoritmo de ordenação escolhido por Gressler e Cera (2014) foi o QuickSort (CORMEN et al., 2009). Ele foi executado utilizando a função `qsort()`, incorporada na biblioteca `stdlib.h` da linguagem C. Essa função recebe como parâmetro a função `Compara Solução`, que é responsável por comparar dois elementos da estrutura a ser

Tabela 6 – Perfil de execução do AG sequencial para cada uma das instâncias do PRV, gerado a partir do `gprof`.

Instância	Função	Nº de chamadas	Soma das chamadas (s)	% do Tempo total
c50	**Ordena População**	12500	-	-
	Compara Solução	-	0,48 s	7,28%
	Inicializa Nova População	12500	0,01 s	0,15%
	Libera Memória	12500	0,04 s	0,61%
c100	**Ordena População**	100000	-	-
	Compara Solução	-	19,65 s	4,76%
	Inicializa Nova População	100000	0,49 s	0,12%
	Libera Memória	100000	1,58 s	0,38%
c120	**Ordena População**	172800	-	-
	Compara Solução	-	51,13 s	4,12%
	Inicializa Nova População	172800	1,02 s	0,08%
	Libera Memória	172800	3,55 s	0,29%
c150	**Ordena População**	337500	-	-
	Compara Solução	-	161,45 s	3,23%
	Inicializa Nova População	337500	3,10 s	0,06%
	Libera Memória	337500	9,89 s	0,20%

ordenada com o objetivo de verificar qual é maior e menor. As chamadas à função `qsort` não aparecem no perfil de execução gerado, por essa ser uma função pronta dentro da biblioteca. Porém, a função `Compara Solução` deve ser implementada, logo, aparece no perfil de execução gerado. A função `qsort()` é uma função de ordenação rápida, com desempenho $O(nlog(n))$, onde n é igual à 500 para a instância c50, 1000 para a instância c100, 1200 para a c120 e 1500 para à c150. Existem paralelizações dessa função conhecida como `qsort paralelo`, porém para ordenar poucos elementos a paralelização pode não fornecer ganhos de desempenho, levando em conta o *overhead* que a paralelização pode ocasionar para conjuntos pequenos de elementos como os necessários.

4.6 Considerações sobre a Análise dos Resultados

Nesse capítulo foi apresentado os resultados referentes à análise do desempenho do AG em diferentes arquiteturas. Onde nossas execuções obtiveram o melhor desempenho com um menor esforço computacional, com um *speedup* na faixa de 2,37 à 3,66. Porém, concluímos que o desempenho da aplicação permanece abaixo do ideal apesar da arquitetura utilizada.

Vimos que o desempenho da aplicação é maior quando se utiliza `sections` aumenta do que quando não se utiliza. Embora o `sections` aumente o tempo de sincronização entre as *threads*, seu uso aumenta ligeiramente o desempenho da aplicação.

Concluímos que, para esse modelo de implementação do AG e para o conjunto de instâncias do PRV utilizadas, a paralelização está de acordo. Sendo que não foram identificadas possíveis melhorias para o AG. Entretanto, existem modelos de implementação mais propensos a serem eficiente após a paralelização, como o modelo baseado em ilhas (WHITLEY; RANA; HECKENDORN, 1999). Na paralelização do AG baseado em ilhas cada ilha consiste em uma *thread* ou processo. Cada ilha executa um AG e evoluir própria subpopulação. As ilhas podem trabalhar em conjunto, trocando periodicamente uma porção de suas populações em processo chamado de migração (WHITLEY; RANA; HECKENDORN, 1999).

O próximo capítulo apresenta a conclusão deste trabalho.

5 Conclusão

Nesse trabalho foi apresentado um AG aplicado ao PRV implementado e paralelizado por Gressler e Cera (2014). Os resultados obtidos por Gressler e Cera (2014) apontam que o desempenho dessa paralelização ficou abaixo do ideal. Logo, o objetivo desse trabalho é investigar os problemas de desempenho presentes e suas causas, realizando o rastreamento das execuções do AG paralelo.

Em nossas execuções utilizamos 4 instâncias do PRV: c50, c100, c120 e c150, com 50, 100, 120 e 150 cidades respectivamente. Para testar o comportamento do AG quando aplicado a instâncias com diferentes cargas computacionais. Primeiro verificamos o comportamento da aplicação em diferentes arquiteturas. Através dos resultados obtidos percebemos que houve um aumento do desempenho de até 1,4 vezes na arquitetura utilizada, porém ainda continuando abaixo do ideal.

Realizamos o rastreamento das execuções do AG e a partir da análise desses rastros verificamos o tempo de espera em sincronização é maior quando se utiliza `sections` do quando não se utiliza. A eficiência obtida com e sem o uso de `sections` é aproximadamente a mesma. Porém, o uso de `sections` aumenta o desempenho da aplicação em até 7%.

Identificamos que entre o *Join* do `parallel for` e o *Fork* das `sections` existe uma região que não foi paralelizada. Essa região contém 3 funções, onde cada uma representa no máximo 7% do tempo total da execução da aplicação. Logo, a paralelização dessas funções não fornecem ganhos de desempenho devido ao custo adicional gerado na criação e gerenciamento das *threads*.

Logo, concluímos que para o modelo implementado do AG e para o conjunto de instâncias do PRV utilizadas, a paralelização está de acordo. E dessa forma, não foram identificadas possíveis melhorias para o AG. Uma possível solução para o aumento do desempenho da aplicação é a implementação de um modelo de AG baseado em ilhas, o qual é mais propenso em ser eficiente quando paralelizado.

Referências

ANDRADE, G. L.; CERA, M. C. Avaliando a paralelização de um algoritmo genético com openmp. In: *Anais do WSCAD-WIC 2015*. Aracaju, SE: SBC, 2016. p. 68–73. Citado 4 vezes nas páginas 8, 31, 51 e 52.

ANDRADE, G. L.; CERA, M. C. Avaliando o uso de sections openmp na paralelização de um algoritmo genético. In: *Anais do Salão Internacional de Ensino, Pesquisa e Extensão*. Uruguaiana, RS: SBC, 2016. No prelo. Citado na página 53.

ARENALES, M.; ARMENTANO, V.; MORABITO, R. *Pesquisa operacional: para cursos de engenharia*. Rio de Janeiro, RJ: Elsevier, 2007. Citado na página 22.

BENEDICT, S.; PETKOV, V.; GERNDT, M. Periscope: An online-based distributed performance analysis tool. In: *Tools for High Performance Computing 2009*. Berlin: Springer, 2010. p. 1–16. Citado na página 43.

BROWNE, S. et al. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: *Supercomputing, ACM/IEEE 2000 Conference*. [S.l.]: IEEE, 2000. p. 42–42. Citado na página 38.

BUTENHOF, D. R. *Programming with Posix threads*. 1st. ed. Boston, MA, USA: Addison-Wesley, 1997. Citado na página 17.

CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP: portable shared memory parallel programming*. Cambridge, MA, USA: MIT Press, 2008. Citado 5 vezes nas páginas 17, 18, 19, 29 e 53.

CHRISTOFIDES, N. et al. Combinatorial optimization. John Wiley & Sons, 1979. Citado 2 vezes nas páginas 30 e 47.

CORMEN, T. H. et al. *Introduction to Algorithms*. 3. ed. Cambridge, Massachusetts, EUA: MIT Press, 2009. ISBN 9780262033848. Citado na página 58.

COULOMB, K. et al. *ViTE - Visual Trace Explorer*. [S.l.], 2010. Citado na página 43.

GEIMER, M. et al. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, Wiley, v. 22, n. 6, p. 702–719, apr 2010. Citado na página 43.

GOMÉZ, A. T.; GALAFASSI, C. Uma abordagem aplicada ao problema de roteamento de veículos utilizando a busca tabu. In: *Anais do XLIII Simpósio Brasileiro de Pesquisa Operacional*. [S.l.]: SBC, 2011. p. 1654–1665. Citado 3 vezes nas páginas 21, 22 e 23.

GRESSLER, H. d. O. *Aumentando a Eficiência de um Algoritmo Genético através da Paralelização com OpenMP*. Monografia (Trabalho de Conclusão de Curso) — Graduação em Ciência da Computaçã, Universidade Federal do Pampa, Alegrete, RS, 2013. Citado na página 49.

GRESSLER, H. d. O.; CERA, M. C. O impacto da paralelização com openmp no desempenho e na qualidade das soluções de um algoritmo genético. *Revista Brasileira de Computação Aplicada*, SBC, Porto Alegre, RS, p. 35–47, 2014. Citado 31 vezes nas páginas 8, 9, 13, 14, 15, 16, 17, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 39, 40, 46, 47, 48, 50, 51, 52, 57, 58 e 60.

GRIFFITHS, D. *Use a Cabeça! C:*. Rio de Janeiro, RJ: Alta Books Editora, 2013. (Use a Cabeça!). ISBN 9788576087946. Citado na página 58.

GWT-TUD GmbH. *Vampir 9.1*. 2016. <https://www.vampir.eu/>. Acessado em 03/11/2016. Citado 5 vezes nas páginas 13, 42, 43, 44 e 45.

KNUPFER, A. et al. Score-p – a joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In: *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing*. Berlin Heidelberg: Springer, 2012. p. 79–91. Citado na página 44.

LABS, A. *Graphviz - Graph Visualization Software*. 2014. <http://www.graphviz.org/>. Acessado em 30/10/2016. Citado na página 49.

LINDEN, R. *Algoritmos genéticos: uma importante ferramenta da inteligência computacional*. 2a. ed. Rio de Janeiro, RJ: Brasport, 2008. Citado 2 vezes nas páginas 24 e 25.

LINDEN, R. *Algoritmos Genéticos*. 2008. <http://www.algoritmosgeneticos.com.br/Figura04.01.jpg>. Acessado em 20/05/2016. Citado na página 24.

MATLOFF, N. *The Art of R Programming: A Tour of Statistical Software Design*. San Francisco: No Starch Press, 2011. (No Starch Press Series). ISBN 9781593273842. Citado na página 44.

NAVAUX, P. O. A.; ROSE, C. A. F. D.; PILLA, L. L. Fundamentos das arquiteturas para processamento paralelo distribuído. In: *Anais da ERAD/RS 2011*. Porto Alegre, RS: SBC, 2011. p. 22–59. Citado 2 vezes nas páginas 15 e 16.

PACHECO, P. *An Introduction to Parallel Programming*. Burslington, MA, USA: Elsevier Science, 2011. Citado 6 vezes nas páginas 15, 16, 17, 20, 21 e 50.

PAJÉ. *Pajé Visualization Tool analysis of execution traces*. 2013. <http://paje.sourceforge.net/index.html>. Acessado em 20/05/2016. Citado na página 43.

PIOLA, T. d. F. *Tracing de aplicações paralelas com informações de alto nível de abstração*. Tese (Doutorado em Física Aplicada) — Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 2007. Disponível em: <http://www.teses.usp.br/teses/disponiveis/76/76132/tde-25102007-114809/>. Acesso em: 18 de maio de 2016. Citado 5 vezes nas páginas 13, 35, 36, 37 e 39.

REGO, C.; ALIDAEE, B. *Metaheuristic Optimization via Memory and Evolution: Tabu Search and Scatter Search*. USA: Springer, 2006. (Operations Research/Computer Science Interfaces Series). Citado na página 22.

REINDERS, J. *Intel Threading Building Blocks, Outfitting C++ for Multi-core Processor Parallelism*. USA: O'reilly, 2010. Citado na página 17.

ROSA, M. F. G. D.; CERA, M. C. Análise da escabilidade de um algoritmo genético paralelizado usando openmp. In: *Anais do WSCAD-WIC 2015*. Florianópolis, SC: SBC, 2015. p. 51–56. Citado 9 vezes nas páginas 9, 31, 32, 33, 34, 46, 47, 51 e 52.

ROSA, M. F. G. D.; CERA, M. C. Estudo preliminar sobre a escalabilidade de um algoritmo genético paralelizado com openmp. In: *Anais da ERAD/RS 2015*. Gramado, RS: SBC, 2015. p. 217–220. Citado 3 vezes nas páginas 8, 32 e 52.

SCHEPKE, C.; LIMA, J. V. F. Programação paralela em memória compartilhada e distribuída. In: *Anais da ERAD/RS 2015*. Gramado, RS: SBC, 2015. p. 45–70. Citado 3 vezes nas páginas 15, 17 e 19.

SCHNORR, L. M. *PajeNG - Trace Visualization Tool*. 2012. <https://github.com/schnorr/pajeng>. Acessado em 18/05/2016. Citado na página 43.

SCHNORR, L. M. *Akypuera*. 2014. <http://github.com/schnorr/akypuera>. Acessado em 18/05/2016. Citado na página 43.

SCHNORR, L. M. Análise de desempenho de programas paralelos. In: *Anais da ERAD/RS 2014*. Alegrete, RS: SBC, 2014. p. 57–81. Citado 8 vezes nas páginas 13, 35, 36, 37, 38, 39, 40 e 41.

SCHNORR, L. M. *Viva visualization toll*. 2014. <https://github.com/schnorr/viva>. Acessado em 03/11/2016. Citado na página 43.

SHENDE, S. S.; MALONY, A. D. The tau parallel performance system. In: *The International Journal of High Performance Computing Applications*. Thousand Oaks, CA, USA: Sage, 2006. v. 20, n. 2, p. 287–311. Citado na página 43.

STEIN, B. d. O. Depuração de programas paralelos. In: *Anais da ERAD/RS 2001*. Gramado: SBC, 2001. p. 151–175. Citado na página 37.

SURHONE, L.; TENNOE, M.; HENSSONOW, S. *Intel Cilk Plus*. [S.l.]: VDM Publishing, 2010. Citado na página 16.

TRAHAY, F. et al. Eztrace: a generic framework for performance analysis. In: IEEE. *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*. Newport Beach, CA, USA, 2011. p. 618–619. Citado na página 43.

VELLASCO, P.; ANDRADE, S.; LIMA, L. *Modelagem de Estruturas de Aço e Mistas*. Rio de Janeiro, RJ: Elsevier Brasil, 2014. Citado na página 24.

VI-HPS. *Score-P User Manual 2.0.2*. [S.l.], 2016. Disponível em: <https://silc.zih.tu-dresden.de/scorep-current/pdf/scorep.pdf>. Citado 4 vezes nas páginas 13, 40, 43 e 44.

WHITLEY, D.; RANA, S.; HECKENDORN, R. B. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, UNIVERSITY COMPUTING CENTRE ZAGREB, v. 7, p. 33–48, 1999. Citado na página 59.

WICKHAM, H. *ggplot2*. 2013. <http://ggplot2.org/>. Acessado em 30/11/2016. Citado na página 44.

WILKINSON, B.; ALLEN, M. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. 2a. ed. [S.l.]: Pearson, 2005. Citado 2 vezes nas páginas 20 e 21.