Francisco E. M. Pereira

Implementação de um Algoritmo Paralelo e Escalável do *Ant Colony System* aplicado ao Problema do Roteamento de Veículos

Pau dos Ferros

2015

Francisco E. M. Pereira

Implementação de um Algoritmo Paralelo e Escalável do *Ant Colony System* aplicado ao Problema do Roteamento de Veículos

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte, em cumprimento às exigências legais como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte – IFRN

Técnico Integrado em Informática

Orientador: Prof. Me. Demétrios A. M. Coutinho

Pau dos Ferros

2015

Francisco E. M. Pereira

Implementação de um Algoritmo Paralelo e Escalável do *Ant Colony System* aplicado ao Problema do Roteamento de Veículos

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte, em cumprimento às exigências legais como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Trabalho aprovado. Pau dos Ferros, 20 de outubro de 2015:

Prof. Me. Demétrios A. M. Coutinho
Orientador
Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte - IFRN

Prof. Me. Lucas Vieira de Souza
Examinador interno
Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte - IFRN

Prof. Esp. Jeferson Queiroga Pereira
Examinador interno
Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte - IFRN

Prof. Dr. Carlos Carreto
Examinador externo
Instituto Politécnico da Guarda - IPG - Portugal

Pau dos Ferros
Out/2015

Agradecimentos

Agradeço primeiramente a Deus pela vida, pela força e pelo cuidado.

Agradeço ao meu orientador, Prof. Me. Demétrios A. M. Coutinho, pela oportunidade, paciência, conhecimento transmitido, empenho em minha orientação e pelo incentivo constante durante o curso.

Agradeço a minha família por ter me proporcionado essa oportunidade e pelo suporte incondicional ao longo da graduação.

Por último, agradeço aos meus amigos, pela companhia e incentivo em todos os momentos.

Resumo

O presente trabalho apresenta a implementação de um algoritmo paralelo e escalável do *Ant Colony System* (ACS) na arquitetura de processadores *multicore* para resolver o Problema do Roteamento de Veículos. Apresenta-se um esquema geral explicando como foi paralelizado cada passo da meta-heurística ACS abordando os pontos mais importantes da implementação paralela. Os experimentos foram realizados em uma máquina de memória compartilhada com 4 núcleos. Foram realizadas análises de desempenho através da comparação entre a versão paralela do algoritmo e a sua respectiva versão sequencial. Para a análise de escalabilidade foram realizados experimentos com diferentes tamanhos de problemas e quantidades de formigas. As conclusões deste trabalho caracterizam o ACS como um bom algoritmo para ser paralelizado, se mostrando bastante promissor para problemas de larga escala.

Palavras-chaves: *Ant Colony System*. Problema do Roteamento de Veículos. Paralelismo. OpenMP.

Lista de ilustrações

Figura 1 – Saída de um mínimo local para um ótimo global 19
Figura 2 – Experimento demonstrando a capacidade de colônias de formigas encontrarem o menor caminho. 21
Figura 3 – A arquitetura de Von Neumann 26
Figura 4 – Exemplo de acesso a uma palavra que não se encontra no cache, provocando um *cache miss*. 27
Figura 5 – Exemplo de um arquitetura *multicore*. 29
Figura 6 – Diferença entre *threads* e processo 31
Figura 7 – Uso de uma barreira. 32
Figura 8 – Exclusão mútua usando regiões críticas. 34
Figura 9 – *Speedup* de acordo com a Lei de Amdahl 37
Figura 10 – Eficiência de um programa paralelo escalável por Gustafson 39
Figura 11 – Estrutura de dados utilizada no algoritmo 40
Figura 12 – Fluxograma do algoritmo ACS Multicore 41
Figura 13 – Modelo de Execução *Fork-Join* 42
Figura 14 – Pseudocódigo do algoritmo ACS paralelo 43
Figura 15 – Escalabilidade de Amdahl para 250 formigas 48
Figura 16 – Escalabilidade de Amdahl para 500 formigas 48
Figura 17 – Escalabilidade de Amdahl para 1000 formigas 49
Figura 18 – Escalabilidade de Gustafson para 250 formigas 50
Figura 19 – Escalabilidade de Gustafson para 500 formigas 51
Figura 20 – Escalabilidade de Gustafson para 1000 formigas 51

Lista de tabelas

Tabela 1 – Instâncias dos problemas usados neste estudo. 46
Tabela 2 – Parâmetros utilizados. 46
Tabela 3 – Valores médios das soluções obtidas através da meta-heurística ACS . . 47

Sumário

1 INTRODUÇÃO . . . 13
1.1 Organização do Texto . . . 14

2 FUNDAMENTAÇÃO TEÓRICA . . . 16
2.1 Problema do Roteamento de Veículos . . . 16
2.1.1 Definição Formal do Problema . . . 17
2.2 Heurística e Meta-heurística . . . 18
2.2.1 A Meta-heurística Otimização por Colônia de Formigas . . . 20
2.2.1.1 Inspiração Biológica . . . 20
2.2.1.2 Ant System - AS . . . 21
2.2.1.3 Ant Colony System - ACS . . . 23
2.2.1.4 Otimização por Colônia de Formigas . . . 24
2.3 Arquitetura Paralela . . . 25
2.3.1 Arquitetura de Von Neumann . . . 25
2.3.2 Memória Cache . . . 26
2.3.3 Multicore . . . 28
2.4 Programação Paralela . . . 29
2.4.1 MPI . . . 29
2.4.2 OpenMP . . . 30
2.5 Paralelismo Aplicado . . . 31
2.5.0.1 Threads e Processos . . . 31
2.5.0.2 Barreira . . . 32
2.5.0.3 Região Crítica . . . 33
2.5.0.4 Overhead . . . 34
2.5.1 Métricas de Escalabilidade Paralela . . . 35
2.5.1.1 Speedup e Eficiência . . . 35
2.5.2 Lei de Amdahl . . . 36
2.5.3 Escalabilidade . . . 38
2.5.3.1 Escalabilidade de Amdahl . . . 38
2.5.3.2 Escalabilidade de Gustafson . . . 38

3 ALGORITMO ANT COLONY SYSTEM PARALELO . . . 40
3.1 Esquema Geral de Paralelização . . . 40
3.2 Implementação do Algoritmo Ant Colony System Multicore . . . 42

4 EXPERIMENTOS E RESULTADOS . . . 46

4.1 Análise dos Valores Obtidos **46**
4.2 Análise de Escalabilidade **47**
4.2.1 Escalabilidade de Amdahl 47
4.2.2 Escalabilidade de Gustafson 49

5 CONCLUSÃO **53**

REFERÊNCIAS **54**

1 Introdução

Nos últimos vinte anos, a comunidade científica tem buscado por novas técnicas de otimização que possibilitem substituir e melhorar os tradicionais métodos exatos, cuja baixa eficiência muitas vezes os torna inúteis para resolver problemas de grande escala, encontrados na vida real, em tempo hábil (ALBA, 2005). Neste contexto, os métodos meta-heurísticos inspirados na natureza surgiram como ferramentas flexíveis e robustas para resolver problemas de otimização NP-difícil (BLUM; ROLI, 2003), o quais, são capazes de encontrar um boa solução, eventualmente a melhor, consumindo tempo e esforço computacionais relativamente pequenos quando comparados aos gastos por métodos exatos.

A classe de algoritmos Otimização por Colônia de Formigas (ACO, do inglês *Ant Colony Optimization*), introduzido por Dorigo (1992), é uma meta-heurística com base populacional inspirada no comportamento coletivo das formigas, que vem sendo usada para resolver diversos problemas de otimização e, em particular, os que podem ser reduzidos a encontrar bons caminhos através de grafos. De acordo com Bonabeau, Dorigo e Theraulaz (1999), os algoritmos ACO são os exemplos mais eficientes de inteligência coletiva, onde têm sido aplicados, com sucesso, para diversos problemas combinatórios, tal como, o Problema do Caixeiro Viajante (TSP, do inglês *Traveling Salesman Problem*), o Problema de Atribuição Quadrático (QAP, do inglês *Quadratic Assignment Problem*), o Problema do Roteamento de Veículos (VRP do inglês *Vehicle Routing Problem*), entre outros.

Na última década, implementações paralelas de meta-heurísticas de base populacional tem-se tornando populares a fim de melhorar a sua eficiência (PEDEMONTE et al., 2011). Implementações paralelas de meta-heurísticas permitem encontrar soluções de qualidade em tempo computacionalmente viável, ao distribuir uma população entre vários elementos de processamento. Além de minimizar o tempo de processamento, com meta-heurísticas paralelas é possível introduzir novos padrões de exploração, a fim de melhorar a qualidade da solução.

Programação paralela envolve tanto questões de hardware como software, de forma, que o modelo de programação paralela está intimamente relacionada ao tipo de hardware que vai executar o programa paralelo. Hoje em dia, existem dois tipos principais de computadores paralelos, multiprocessados ou *multicores*[1] e sistemas distribuídos, e programas que executam sobre eles, utilizam, respectivamente, programação de memória compartilhada e programação distribuída (ALBA, 2005).

Diante da popularização de hardwares paralelos, processadores *multicores* são

[1] Processadores formados por diversos núcleos de processamento.

facilmente encontrados tanto em computadores pessoais como em servidores (PACHECO, 2011). Para tirar o total proveito dessa popularização de arquiteturas paralelas, diversos estudos vem sendo realizados para saber o quão satisfatório um programa paralelo pode ser executado em uma arquitetura paralela. Neste sentido, diversos autores tem estudado formas de implementações paralelas da meta-heurística ACO. Stützle (1998) resolveu o Problema do Caixeiro Viajante, onde ele introduziu a execução de várias colônias de formigas, em que as colônias são distribuídas entre os processadores, a fim de aumentar a velocidade dos cálculos e melhorar a qualidade da solução, através da cooperação entre colônias. Lucka e Piecka (2009) estudou a implementação paralela do ACO para resolver o Problema do Roteamento de Veículos usando *threads* Posix, onde múltiplas colônias se comunicam de forma assíncrona na busca de soluções. Mohamed, Elbialy Abouelfarag e Aly (2015) propõem a análise e implementação paralela do ACO aplicado ao Problema do Caixeiro Viajante, usando a especificação OpenMP (*Open Multi-Processing*), tendo como foco a análise do desempenho do algoritmo em relação a distribuição de carga entre os processadores.

Embora a meta-heurística ACO encontre soluções de boa qualidade, esta é ainda uma área a se explorar, uma vez que, segundo (PEDEMONTE et al., 2011), em 2010, apenas treze publicações de implementações paralelas da meta-heurística ACO foram realizadas. Neste contexto, este trabalho visa apresentar uma implementação paralela da meta-heurística ACO, aplicada ao Problema do Roteamento de Veículos, em um ambiente *multicore*. Como também, analisar o comportamento da escalabilidade paralela do algoritmo ao modificar o tamanho do problema e o número de formigas.

Objetivos

O objetivo geral desse trabalho consiste em analisar a escalabilidade da implementação paralela da meta-heurística Otimização por Colônia de Formigas aplicado ao Problema do Roteamento de Veículos Capacitado. Os objetivos específicos são:

- Implementar uma versão sequencial da meta-heurística ACO;
- Implementar uma versão paralela da meta-heurística ACO;
- Analisar a escalabilidade e o desempenho computacional da versão paralela em relação a serial da meta-heurística ACO;

1.1 Organização do Texto

O restante deste trabalho é organizado da seguinte maneira: no Capítulo 2 é introduzido a noção sobre o Problema do Roteamento de Veículos e também é realizado

uma descrição da meta-heurística ACO, assim como uma breve contextualização sobre os conceitos básicos para desenvolver programas paralelos, e formas de medir a sua escalabilidade; o Capítulo 3 apresenta em detalhes a implementação paralela da meta-heurística ACO; e no Capítulo 4 é realizada uma análise de escalabilidade dos resultados obtidos através da implementação paralela do algoritmo. E finalmente, o Capítulo 5 aborda as considerações finais e propostas de trabalhos futuros.

2 Fundamentação Teórica

O objetivo deste capítulo é apresentar a formulação do Problema de Roteamento de Veículos, a meta-heurística otimização por colônia de formigas, a arquitetura *multicore*, os conceitos básicos para criar programas paralelos e as métricas básicas usadas para analisar a escalabilidade paralela.

2.1 Problema do Roteamento de Veículos

O Problema de Roteamento de Veículos (VRP, do inglês *Vehicle Routing Problem*), introduzido por (DANTZIG; RAMSER, 1959), é um problema presente na maioria das empresas de transporte, logística e distribuição. Seu objetivo é determinar, dentre todas as possíveis rotas alternativas, aquela que representa o menor custo, ou seja, a solução ótima (GOLDBARG; LUNA, 2000). Existem algumas restrições que são inerentes ao problema, são elas: cada cliente é visitado uma única vez por um único veículo; cada rota é iniciada num depósito e finalizada no mesmo depósito; todas as demandas de todos os consumidores devem ser satisfeitas. Há restrições que podem ser acrescentadas de forma que possa modificar o problema, deixando-o mais complexo. Como por exemplo:

- Restrição no número de pontos de entrega em cada rota;
- Restrição de tempo ou distância de uma rota;
- Restrição de janela de tempo: cada ponto deve ser visitado em um período de tempo específico;
- Restrição de precedência entre cidade: o ponto de entrega i deve ser visitado antes do ponto j;
- Tipo de frota: veículos disponíveis para execução das rotas são homogêneos ou heterogêneos;
- Quantidade de veículos contidos na frota: quantidade limitada ou ilimitada;
- Capacidade dos veículos: veículos com capacidade limitada ou ilimitada;
- Quantidade de depósitos: único depósito ou vários depósitos.

Devido a sua importância tanto prática como teórica, encontram-se na literatura diversas variantes do VRP. Essas variantes são compostas por restrições que são incorporados ao VRP simples, de modo a modificar a complexidade e deixá-lo mais próximo a problemas reais e específicos. Alguns deles:

Problema do Caixeiro Viajante
O Problema do Caixeiro Viajante (TSP, do inglês *Traveling Salesman Problem*) é, sem dúvida, um dos problemas de otimização mais conhecido. Ele consiste em encontrar um trajeto que visite N cidades diferentes, sem repetição, retornando à origem e utilizando a menor rota possível. Pode ser considerado um Problema de Roteamento de Veículos quando esse possui apenas um único veículo com capacidade igual ou superior à demanda total de todas as várias cidades visitadas.

Problema de Roteamento de Veículos Capacitado
Um dos problemas mais gerais de roteamento é o CVRP, do inglês *Capacitated Vehicle Routing Problem*, onde uma rota de veículos homogênea serve um conjunto de clientes com uma demanda previamente conhecida, a partir de um único depósito. No entanto, a principal restrição deste problema é que a capacidade do veículo nunca pode ser excedida ao realizar um percurso. O objetivo deste problema consiste em minimizar os custos totais, quer em termos do número de veículos da frota, como o tempo ou a distância de viagem das rotas, tendo em conta a capacidade de cada veículo (BRESLIN; KEANE, 1997).

Problema de Roteamento de Veículos com Janela de Tempo
O Problema de Roteamento de Veículos com Janela de Tempo (VRPTW, do inglês *Vehicle Routing Problem With Time Windows*) é uma variante bastante abordada na literatura e, também, uma das extensões mais importantes do CVRP (GAMBARDELLA et al., 1999). Esta variante introduz uma restrição de tempo, que determina um intervalo de tempo para começar e terminar o atendimento do consumidor e ainda um intervalo para saída e retorno ao depósito. Existem diversos exemplos deste problema na vida real, como as entregas de correio postal e ônibus escolares.

Problema do Roteamento de Veículos com *Backhauls*
O Problema do Roteamento de Veículos com *Backhauls* (VRPB, do inglês *Vehicle Routing Problem with Backhauls*) é uma extensão do problema capacitado, que além da entrega, é realizado a coleta de mercadorias durante o trajeto. Contudo, as entregas devem ser efetuadas antes da coleta e, em cada ponto do trajeto, existe somente demanda de entrega ou coleta (TOTH; VIGO, 1997).

Neste trabalho será abordado o problema de roteamento de veículos capacitado, devido a isso, na próxima seção é realizado uma definição formal deste problema.

2.1.1 Definição Formal do Problema

O CVRP pode ser definido como um grafo (direcionado ou não) $G = (V, A, C)$, onde $V = v_0, v_1, \ldots, v_n$ é o conjunto dos vértices, $A = (a_{ij} = v_i, v_j) : v_i, v_j \in V$ é o conjunto de

arestas que ligam os vértices e $C = c_{ij}$, é uma matriz que denota o custo (distância) entre os vértices v_i e v_j. Por convenção o vértice v_0 representa o depósito, sendo este a base de uma frota composta por k veículos homogêneos de capacidade Q, enquanto os vértices remanescentes correspondem aos consumidores. Cada consumidor v_i tem a demanda não negativa d_i, sendo $d_0 = 0$. Cada aresta a_{ij} está associada a um valor não negativo c_{ij} que representa a distância entre os consumidores. No geral, o Problema de Roteamento de Veículos Capacitado consiste em determinar um conjunto de rotas que deverão ser percorridas pelos veículos, com o objetivo de minimizar a distância total, respeitando as seguintes restrições:

- Cada rota começa e termina no depósito v_0;
- Todo e qualquer consumidor de V é visitado somente uma vez por somente um veículo;
- A carga transportada não deve exceder a capacidade Q do veículo;
- Todas as demandas de todos os consumidores devem ser satisfeitas.

2.2 Heurística e Meta-heurística

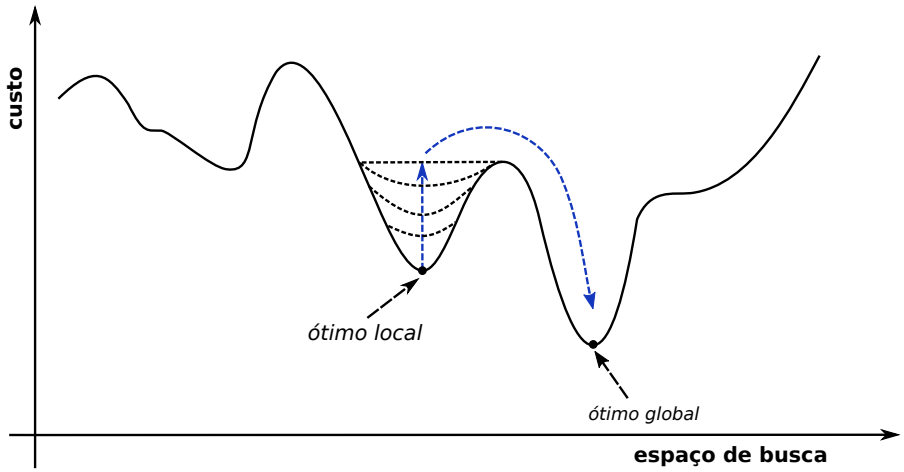
O nome heurística é derivado da palavra grega *heuristiké*, que significa "arte de encontrar, descobrir". Em ciência da computação, heurísticas são métodos aproximativos que baseado na experiência ou julgamento do problema abordado visam encontrar uma boa solução em um tempo computacionalmente viável, no entanto, não garante produzir uma solução ótima (FOULDS, 2012). Os métodos aproximativos, são usualmente divididos entre métodos construtivos e métodos de busca local (BLUM; ROLI, 2003). Algoritmos construtivos geram soluções a partir de uma solução inicialmente vazia, que incrementalmente é adicionado componentes à ela até que esteja completa. Eles são tipicamente mais rápidos, mas muitas vezes retornam soluções de qualidade inferior, se comparados aos algoritmos de busca local. Algoritmos de busca local iniciam a partir de uma solução completa que iterativamente testa a substituição da solução atual por uma solução melhor.

Já o termo **meta-heurística** deriva da composição da palavra "heurística" e o prefixo "meta", que exprime a ideia de "nível superior, maior generalidade", indicando um nível superior de descoberta. Em linhas gerais, meta-heurísticas são procedimentos que resolvem de forma genérica problemas de otimização, explorando eficientemente o espaço de busca[1], a fim de encontrar boas soluções. Normalmente as soluções encontradas tem melhor qualidade que as obtidas através de heurísticas (CARDOSO, 2009). A figura 1 demonstra a ação de escapar de um ótimo local, que representa uma solução parcial, para

[1] Espaço de busca é o conjunto de todas as soluções factíveis do problema.

um local de melhor qualidade, que neste caso é o ótimo global, que representa a melhor solução do problema.

Figura 1 – Saída de um mínimo local para um ótimo global



Fonte: Autoria própria.

Dentre os algoritmos classificados como meta-heurística podemos citar: Otimização por Colônia de Formigas (ACO, do inglês *Ant Colony Optimization*), Algoritmo Genético (GA, do inglês *Genetic Algorithm*), Busca Tabu (TS, do inglês *Tabu Search*), Recozimento Simulado (SA, do inglês *Simulated Annealing*), Pesquisa em Vizinhança Variável (VNS, do inglês *Variable Neighboorhod Search*), Busca Local Iterativa (ILS, do inglês *Iterated Local Search*), entre outros.

Para que uma meta-heurística explore um espaço de busca de forma inteligente, obtenha soluções de ótima qualidade e consiga mover-se para áreas não exploradas quando necessário, os conceitos de diversificação e intensificação (do inglês *exploration* e *exploitation*) devem ser atingidos. Diversificação significa encontrar novas regiões no espaço de busca que ainda não tenham sido investigados, enquanto, intensificação consiste em concentrar as buscas em regiões promissoras em torno de boas soluções.

Um dos maiores desafios de uma meta-heurística é estabelecer o equilíbrio entre diversificação e intensificação. Algoritmos que focam na intensificação têm uma velocidade de convergência muito rápida, mas correm o risco de não encontrar a solução ótima e talvez ficarem presos em um ótimo local. Já algoritmos que realizam uma diversificação excessiva podem encontrar a solução ótima, mas levaram um tempo computacional muito alto. Então o balanceamento adequado da meta-heurística é fundamental para que, de um lado possa ser identificado rapidamente regiões no espaço de busca com soluções de boa qualidade, e de outro lado para não perder muito tempo em regiões do espaço de busca

que já foram explorada e não forneçam boas soluções.

2.2.1 A Meta-heurística Otimização por Colônia de Formigas

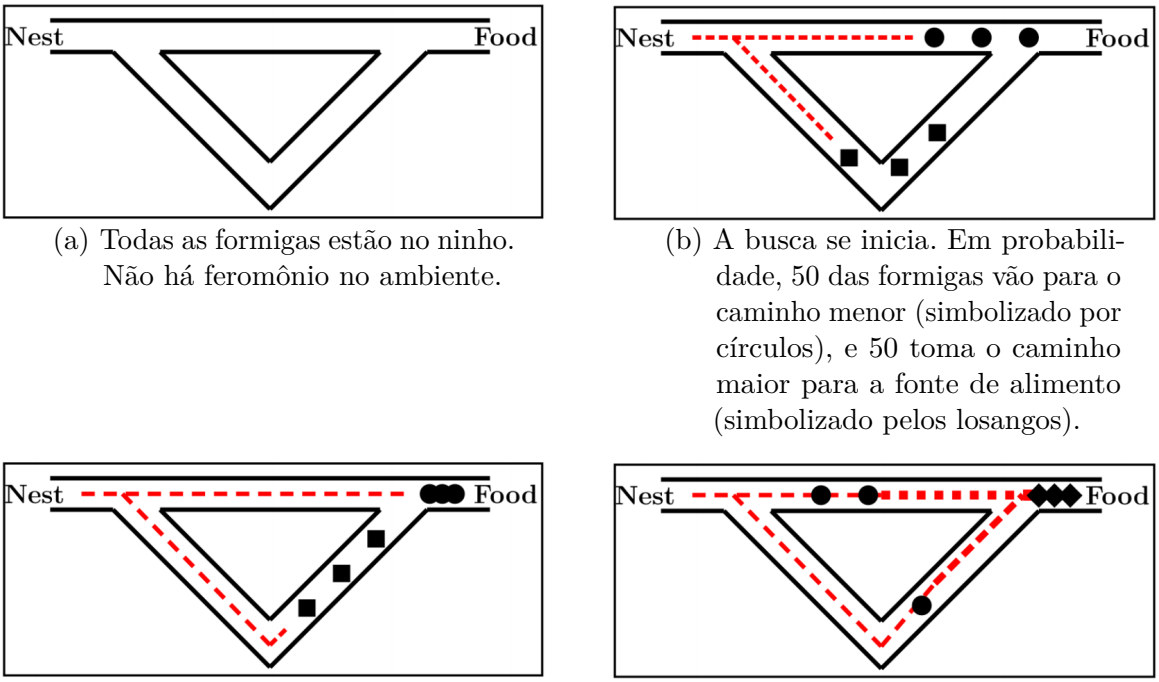
Esta seção tem como propósito apresentar a meta-heurística Otimização por Colônia de Formigas (ACO, do inglês *Ant Colony Optimization*), contextualizando a sua inspiração, como algumas variantes.

2.2.1.1 Inspiração Biológica

O algoritmo de formigas foi inspirado pela observação de colônias de formigas reais. Formigas são insetos sociais, isto é, insetos que vivem em colônias e tem como objetivo a sobrevivência da colônia ao invés de ser focado na sobrevivência individual (DORIGO; CARO; GAMBARDELLA, 1999). Um importante e interessante comportamento das formigas é capacidade delas encontrarem o menor caminho entre uma fonte de comida e o seu ninho.

Para analisar mais detalhadamente como as formigas conseguem encontrar o menor caminho entre o ninho e a fonte de comida, Goss et al. (1989) realizou um experimento denominado *double brige*, onde uma colônia de formigas da espécie *Linepithema humile* e uma fonte de comida foram separados por duas pontes, conforme à figura 2. Esta figura mostra como elas encontram o caminho mais curto entre a fonte de comida e o ninho. Enquanto caminham da fonte de comida para a o ninho e vice-versa, as formigas depositam sobre o caminho percorrido uma substância química, chamada de feromônio, formando assim uma trilha de feromônios. Quando outras formigas detectam a existência desse feromônio, elas tendem escolher a direção com o maior nível dessa substância. Através do processo de depósito do feromônio a trilha de menor distância tende a ficar mais forte, pois mais formigas irão passar por ela. Quanto maior a quantidade de feromônio no caminho, maior será a probabilidade das formigas optarem por essa direção. O nível de feromônio é estabelecido como base no comprimento do caminho e na qualidade de fonte de alimento, ou seja, o nível de feromônio vai aumentar de acordo com o número de formigas que seguirem pelo mesmo caminho. Em questão de tempo, todas as formigas tendem a optar pelo menor caminho. O ato de todas as formigas convergirem sempre pelo mesmo caminho é chamado de estagnação.

Figura 2 – Experimento demonstrando a capacidade de colônias de formigas encontrarem o menor caminho.



(a) Todas as formigas estão no ninho. Não há feromônio no ambiente.

(b) A busca se inicia. Em probabilidade, 50 das formigas vão para o caminho menor (simbolizado por círculos), e 50 toma o caminho maior para a fonte de alimento (simbolizado pelos losangos).

(c) As formigas que tomaram o menor caminho chegaram mais cedo ao ninho. Portanto, quando retornarem, a probabilidade de tomar o menor caminho novamente é alta.

(d) A trilha de feromônio no menor caminho recebe, em probabilidade, um reforço mais forte, e a probabilidade de tomar esse caminho aumenta.

Fonte: Adaptado de (BLUM, 2005)

2.2.1.2 Ant System - AS

O *Ant System* (AS) foi o primeiro algoritmo ACO proposto por Dorigo, Maniezzo e Colorni (1996) e aplicado ao Problema do Caixeiro Viajante (TSP). Embora tenha alcançado bons resultados iniciais, o seu desempenho não foi o suficiente para competir com os melhores algoritmos propostos para resolver o TSP (DORIGO; STüTZLE, 2004).

O AS tem duas fases principais, a construção de uma solução e atualização das trilhas de feromônios. Durante a fase de construção, m formigas criam uma solução de acordo com uma regra probabilística chamada de *random proportional rule*, que é a função da distância entre as cidades (i, j) e a quantidade de feromônio sobre a aresta (i, j). Para não permitir passos inválidos todas as formigas possuem uma memória M que armazena quais cidades já foram visitadas durante a construção de uma rota. Assim, enquanto ela constrói uma solução, ela é forçada a não visitar uma mesma cidade duas vezes até que a solução esteja completa.

Quando a formiga completa uma solução ela deposita uma certa quantidade de

feromônio em cada aresta(i, j) que ela visitou.

A probabilidade de transição (*random proportional rule*) de uma cidade i para um cidade j é definida pela fórmula:

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta}, \quad \text{se } j \in N_i^k, \tag{2.1}$$

onde α e β são parâmetros que determinam a influência relativa entre o feromônio e a distância entre as cidades, τ_{ij} é a quantidade de feromônio na aresta (i, j), N_i^k é conjunto de todas as cidades vizinhas que ainda não foram visitadas pela formiga k e η_{ij} é um valor heurístico calculado por $\frac{1}{distância_{ij}}$.

Por esta regra, a probabilidade de escolher uma aresta (i, j) aumenta proporcionalmente de acordo com o valor associado à trilha de feromônio τ_{ij} e à informação heurística η_{ij}, ou seja, será escolhido a cidade com maior probabilidade. Os parâmetros α e β possuem uma notável influência sobre o comportamento do algoritmo. Se α tem um valor muito alto, significa que o feromônio é muito importante e, deste modo, as formigas tendem a escolher arestas que outras formigas escolheram no passado, o que tem um efeito de intensificação. Se $\alpha = 0$, o valor da trilha de feromônios não é levado em conta, e o algoritmo terá um comportamento semelhante a um algoritmo guloso. Mas valores muito altos podem provocar a estagnação precoce do algoritmo, que é quando todas as formigas tendem a seguir pelo mesmo caminho (DORIGO; MANIEZZO; COLORNI, 1996). Já um valor muito baixo de β provocaria a estagnação precoce do algoritmo, e um valor alto demais o aproxima de uma construção gulosa.

Ao terminar a fase de construção, dá-se inicio a atualização das trilhas de feromônios, que envolve tanto o incremento do feromônio como a sua evaporação, que é realizado por todas as formigas. A atualização do feromônio é definida pela seguinte fórmula:

$$\tau_{ij} \leftarrow \underbrace{(1-\rho)\tau_{ij}}_{\text{evaporação}} + \underbrace{\sum_{k=1}^{m} \Delta\tau_{ij}^k}_{\text{incremento}} \tag{2.2}$$

onde m é o número de formigas, $0 < \rho \leq 1$ é o fator de evaporação do feromônio e $\Delta\tau_{ij}^k$ é a quantidade de feromônio que uma formiga k deposita em uma aresta visitada por ela, que é definido por:

$$\Delta\tau_{ij}^k = \begin{cases} 1/C^k, & \text{se a aresta (i, j) pertence para } T^k \\ 0, & \text{caso contrário} \end{cases} \tag{2.3}$$

onde C^k é o custo da rota T^k percorrida pela formiga k.

É possível observar a existência de diversos parâmetros que influenciam diretamente no comportamento do algoritmo, e o ajuste refinado desses parâmetros é crucial para se obter um bom desempenho do algoritmo, como também uma boa solução.

2.2.1.3 Ant Colony System - ACS

O *Ant Colony System* foi introduzido por Dorigo e Gambardella (1997) para melhorar o desempenho do AS, o qual permitiu encontrar boas soluções em um tempo razoável para problemas de instâncias pequenas. O ACS se difere em três aspectos do *Ant System*:

- a modificação na regra de escolha do próximo nó;
- a adição de uma regra de atualização local da trilha de feromônios;
- o uso das regras de atualização local e global para favorecer a diversificação.

A regra de transição do próximo nó, também chamada de *pseudorandom proportional rule*, foi modificada para permitir explicitamente a diversificação. A probabilidade de uma formiga k se mover de uma cidade i para uma cidade j é dado pela seguinte equação:

$$j = \begin{cases} \text{argmax}_{l \in N_i^k} \{\tau_{il}[\eta_{il}]^\beta\}, & \text{se } q \leq q_0; \\ J, & \text{se } q > q_0; \end{cases} \tag{2.4}$$

onde q é uma variável aleatória distribuída uniformemente entre $[0, 1]$, q_0 é uma constante que varia entre 0 e 1 inclusos, e J é uma variável selecionada de acordo com a equação 2.1.

Portanto a regra de transição do ACS é idêntica a do AS quando $q > q0$, e é diferente quando $q \leq q0$. Mais precisamente, $q \leq q0$ corresponde para a diversificação do conhecimento disponível sobre o problema, isto é, o conhecimento heurístico sobre as distâncias entre as cidades e o conhecimento adquirido na forma de trilha de feromônios, enquanto que $q > q0$ favorece mais a intensificação.

Após uma formiga k se mover de uma cidade i para uma cidade j através da regra de transição, é executado a atualização local da trilha de feromônio, ou atualização *offline*. Na atualização local é removido uma certa quantidade de feromônio da aresta (i, j), encorajando assim, que outras formigas explorarem caminhos alternativos, aumentando a diversificação do algoritmo. A fórmula é dada por:

$$\tau_{ij} \leftarrow (1 - \varepsilon)\tau_{ij} + \varepsilon\tau_0, \tag{2.5}$$

onde $\varepsilon \in (0, 1]$ determina a taxa de decaimento e τ_0 é o valor inicial da trilha de feromônios, que é definido por $1/nC^{nn}$, onde n é o número de cidades de uma instância do TSP e C^{nn} é o custo de uma rota construída através da heurística do vizinho mais próximo.

Ao final de cada iteração é realizado a atualização global das trilhas de feromônios, que somente é executado pela melhor formiga da iteração (*iteration-best ant*) ou a melhor formiga encontrada até o momento (*best-so-far ant*), em contraste com *Ant System* que ao final da iteração todas as formigas eram permitidas à atualizar o feromônio. Durante a atualização global é realizado tanto a evaporação dos arcos como o depósito de feromônio nas arestas visitadas pela melhor formiga. A atualização global no ACS é dado pela seguinte equação:

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \rho\Delta\tau_{ij}^{bs}, \tag{2.6}$$

onde bs é a melhor formiga global, e $\Delta\tau_{ij}^{bs}$ é o inverso do custo da melhor rota. Durante a atualização global, somente as aresta percorridas pela melhor formiga são atualizadas, o que caracteriza um efeito de intensificação, uma vez que, ela tende a reforça o nível de feromônio presente nessas arestas.

2.2.1.4 Otimização por Colônia de Formigas

A meta-heurística Otimização por Colônia de Formigas é uma parte do enorme campo da inteligência coletiva em que os cientistas estudam o comportamento padrão de abelhas, cupins, formigas e outros insetos sociais, a fim de resolver problemas complexos de otimização combinatória. O método ACO foi introduzido por Marco Dorigo (DORIGO, 1992) em sua tese de PhD, que utiliza um conjunto de agentes (formigas artificiais) em um processo de construção estocástico, para percorrer um grafo completo $G = (V, A)$ em busca das melhores soluções para problemas de otimização combinatória.

A estrutura geral da meta-heurística ACO consiste de uma etapa de inicialização e de três fases principais, são elas: construção de soluções, aplicar ações extras e atualização de feromônio. A construção da solução é repetida até que um critério de parada, geralmente definido pelo número máximo de iterações ou tempo máximo de processamento, seja satisfeito. O algoritmo 1 ilustra a estrutura proposta por Dorigo e Stützle (2004).

Na fase de construção de soluções é utilizado um conjunto de formigas para a construção de uma solução factível para um problema. O processo de construção inicia-se com uma solução parcial vazia e através de uma regra probabilística as formigas vão percorrendo o grafo G e incrementalmente construindo a solução do problema em questão.

Depois que uma formiga completa a fase de construção da solução, algumas ações extras podem ser realizadas sobre essa solução. Essa fase é opcional. Comumente a

Algoritmo 1 Meta-heurística ACO - Pseudocódigo

```
1: procedimento META-HEURÍSTICA ACO
2:     Inicializar parâmetros
3:     enquanto condição de parada não satisfeita faça
4:         ConstruirSoluções
5:         AplicarAçõesExtras      % opcional
6:         AtualizarFeromônio
7:     fim enquanto
8: fim procedimento
```

ação mais utilizada consiste de uma **busca local** como tentativa de melhorar a solução encontrada pela formiga, obtendo assim um maior efeito de intensificação. De acordo com Stvtzle e Hooos (1999), usar a busca local nesta etapa é uma forma de hibridizar[2] o ACO e as melhores implementações ACO geralmente se valem desse recurso.

Por ultimo, temos a etapa de atualização de feromônios. Esta fase consiste de duas partes. Primeiro, é realizada a evaporação do feromônio, onde ele é decrementado uniformemente do percurso em que as formigas passaram ao construir a solução. O que na prática evita a rápida convergência do algoritmo a uma região sub ótima, favorecendo assim, a descoberta de novos espaços de busca. Segundo, é realizado o depósito do feromônio, aumentando a probabilidade de outras formigas escolherem essa solução no futuro.

2.3 Arquitetura Paralela

Nesta seção, são discutidas as características da arquitetura de von Neumann, e logo em seguida é realizado uma breve descrição sobre memória cache e a sua influência nos computadores atuais. Posteriormente é apresentado a arquitetura *multicore*.

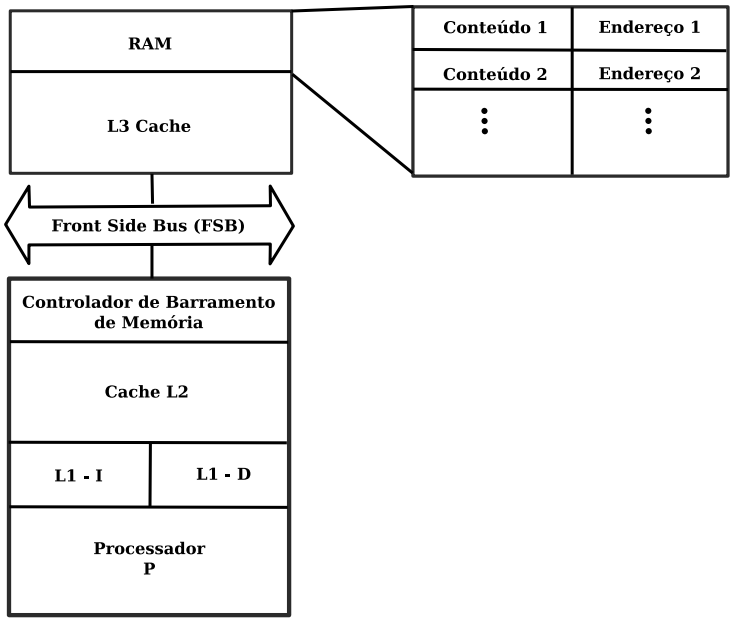
2.3.1 Arquitetura de Von Neumann

A arquitetura *clássica* de von Neumann consiste de uma memória principal, uma unidade central de processamento (CPU) ou processador ou núcleo, e uma interligação entre a memória e a CPU (PACHECO, 2011). A memória principal consiste de um conjunto de locais, onde cada um é capaz de armazenar instruções e dados. Cada local consiste de um endereço, o qual é usado para acessar à localização e o conteúdo, ou seja, as instruções ou os dados armazenados no local (ver figura 3).

A separação entre a CPU e a memória leva para um conhecido problema, chamado de **gargalo de von Neumann**, que consiste na diferença de *clock* entre a CPU e a memória. Uma vez que, a taxa de transferência de dados é muito menor do que a taxa com

[2] Uma meta-heurística é chamada de hibrida quando está é combinada com outras técnicas de otimização, podendo ser métodos exatos, outras meta-heurísticas, ou com heurísticas de busca específicas para um problema

Figura 3 – A arquitetura de Von Neumann



Fonte: Adaptado de (LIN; SNYDER, 2008).

que o processador pode trabalhar. Isso limita seriamente a velocidade de processamento, principalmente quando a CPU tem de executar o processamento de grandes quantidades de dados. A CPU é continuamente forçada a esperar por dados que precisam ser transferidos para ou a partir da memória. Como a velocidade da CPU e o tamanho da memória têm aumentado mais rapidamente que a taxa de transferência entre eles, a gravidade do problema vem crescendo a cada geração de CPU.

Na tentativa de minimizar o gargalo de von Neuman, e, de modo mais geral, melhorar o desempenho da CPU, tem sido realizada diversas modificações na arquitetura básica de von Neumann, como: memória cache, memória virtual e paralelismo intrínseco. Em razão de a memória cache representar uma grande melhoria da arquitetura de Von Neumann, e como também, influenciou diretamente neste trabalho, a seguir será realizada uma breve descrição dela.

2.3.2 Memória Cache

A memória cache é um conjunto de memória utilizado pela CPU para diminuir o tempo de acesso a memória principal. Ela é muito menor do que a memória principal, tanto em *bytes* quanto espaço físico, muitas vezes cabendo no próprio circuito integrado do processador. O tempo de acesso a memória cache é muito mais rápido do que um *microchip* de memória comum.

Embora a memória cache possa ser vista como uma estrutura monolítica, na prática, ela é normalmente divida em níveis: o primeiro nível (L1) é o menor e mais rápido, e níveis

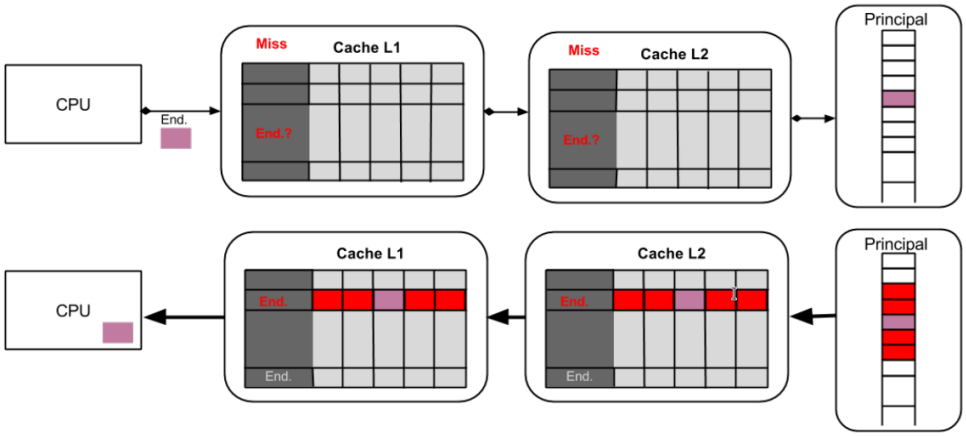
mais elevados (L2, L3, ...) tem maior capacidade de armazenamento, mas, no entanto, são mais lentos. A maioria dos sistemas atualmente, tem pelo menos dois níveis, e o fato de alguns possuírem três níveis já é bastante comum (PACHECO, 2011).

A memória cache segue o princípio de localidade de referências, que é a propriedade do sistema concentrar seus acessos em poucas áreas de memória a cada instante. Existem ao menos dois tipos de localidade de referências: a **localidade temporal** implica que se um recurso usado há pouco tempo será provavelmente usado novamente em um futuro próximo; já **localidade espacial** implica que um recurso será mais provavelmente acessado se outro recurso próximo a ele já foi acessado.

A fim de explorar o princípio da localidade, o sistema utiliza efetivamente uma ampla interconexão para acessar os dados e instruções. Isto é, um acesso à memória que irá operar em blocos de dados e instruções, ao invés de instruções individuais e itens de dados individuais. Esses blocos são chamados blocos de cache ou linhas de cache.

Quando a CPU necessita acessar uma palavra da memória, ela trabalha o seu caminho para baixo na hierarquia do cache. Inicialmente, verifica o cache de nível 1, em seguida o de nível 2, e assim por diante. Se a palavra estiver no cache, então temos um *cache hit*, e a palavra é entregue ao processador a uma velocidade muito alta. Se, no entanto, o processador não encontra a palavra que precisa no cache, então temos um *cache miss*, e os dados de acesso esperam até que o conteúdo da cache seja atualizado a partir da memória principal (GEBALI, 2011). O bloco contendo a palavra é recuperado da memória principal e carregado na memória cache, e posteriormente entregue ao processador (ver figura 4).

Figura 4 – Exemplo de acesso a uma palavra que não se encontra no cache, provocando um *cache miss*.



Fonte: (COUTINHO, 2014).

2.3.3 Multicore

Durante as últimas décadas, a capacidade de processamento dos processadores deu-se, basicamente, através do aumento da frequência de *clock*. A cada geração, o número de transistores presentes nos processadores, mais que dobravam a cada dezoito meses. Esse grande aumento de performance acostumou os programadores a enxergarem o computador como uma máquina que executava instruções de maneira serial e a esperarem, a cada nova geração de processadores, uma melhoria natural no desempenho de seus códigos. No entanto, devido a limitações físicas, este avanço de velocidade chegou ao fim (JAGTAP, 2009). Restrições como o aumento da dissipação de calor e miniaturização de novos processadores, obrigaram a indústria dos computadores a mudar de abordagem: o ganho de desempenho do processador não mais seria baseado no aumento da frequência de processamento, mas através da adição de mais núcleos em um mesmo *chip*. Estes processadores são comumente conhecidos como processadores *multicores*. Através dos processadores *multicores*, pode-se diminuir a frequência de *clock* de cada núcleo, amenizando assim, o problema do aquecimento.

Um sistema *multicore* consiste em colocar dois ou mais núcleos de processamento no interior de um único *chip* que trocam informações via memória compartilhada[3]. Uma vez que, o sistema operacional tem a responsabilidade de dividir as tarefas entre os núcleos disponíveis, permitindo assim, que diversas tarefas sejam executadas em paralelo.

O desenvolvimento de processadores *multicore* tem como motivação duas realidades: o grande consumo de energia e a dificuldade de continuar aumentando a frequência de operações dos processadores, como prevê a Lei de Moore[4]. O aumento de desempenho alçando pelos processadores se baseia na aplicação de técnicas que exploram o paralelismo de instruções, e também pelo aumento de frequência de *clock*. Entretanto, o consumo de energia também atinge níveis impraticáveis, já que é proporcional ao aumento da frequência de operações. Os processadores multicores, já usados em outras áreas, se tornaram alternativas para computadores de propósito geral.

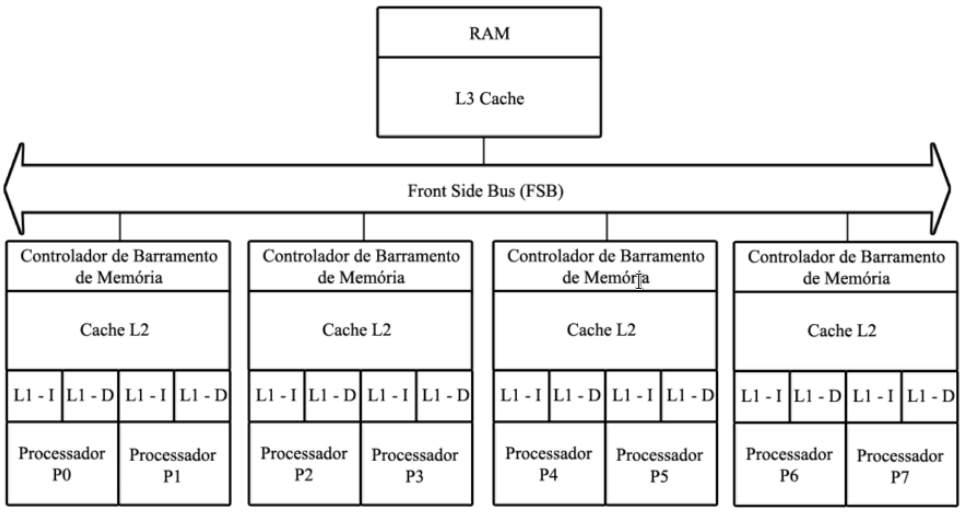
Uma arquitetura *multicore* é geralmente um multiprocessamento simétrico (SMP, do inglês *Simultaneous Multiprocessing*), onde dois ou mais os núcleos idênticos compartilham um segundo ou terceiro nível de cache comum e interconectado, implementado em um único circuito VLSI - *Very Large Scale Integration* (figura 5).

De maneira geral, a arquitetura *multicore* trouxe uma série de vantagens, tais como uso eficiente de energia, redução do custo de desenvolvimento e as possibilidades de exploração do paralelismo de *threads* e processos (PATTERSON; HENNESSY, 2013).

[3] Memória compartilhada é uma memória que pode ser acessada simultaneamente por múltiplos processos/*threads* com a intenção de prover comunicação entre eles ou para evitar cópias redundantes.

[4] Gordon Moore (MOORE, 1965) observou que o número de transistores de *chips* teria um aumento de 60%, pelo mesmo custo, a cada 18 meses. Posteriormente, essa observação ficou conhecida com a Lei de Moore.

Figura 5 – Exemplo de um arquitetura *multicore*.



Fonte: (LIN; SNYDER, 2008).

Sendo que neste trabalho, é explorado como os *softwares* podem se beneficiar do paralelismo por *threads*.

2.4 Programação Paralela

Tradicionalmente, a grande maioria dos programas são escritos como um fluxo de instruções sequenciais, para serem executados em um único computador com uma única CPU. Já programação paralela, por outro lado, faz o uso de múltiplos elementos de processamento, podendo assim, executar partes diferentes de um mesmo programa simultaneamente. Neste contexto, essa sessão tem como objetivo apresentar dois softwares que podem ser utilizados para a programação paralela: o MPI baseado em memória distribuída e o OpenMP baseado em memória compartilhada.

2.4.1 MPI

MPI (*Message-Passing Interface*) é um padrão que define a sintaxe e a semântica de um conjunto de rotinas que devem ser implementados numa biblioteca de transmissão de mensagens. Este padrão surgiu a partir da colaboração de um grupo de pesquisadores e instituições representando a indústria, o meio acadêmico e importantes centros de pesquisas. Como resultado inicial desta colaboração surgiu o MPI Forum em 1992, e dois anos mais tarde foi lançado o padrão MPI 1.0. Em 1997, após um longo período de discussão foi estabelecido diversas melhorias do padrão, resultando na versão 2.0, e em 2008 foi realizada uma nova atualização de menor porte. Atualmente existem diversas implementações *open source* do padrão MPI, sendo que, as mais populares são OpenMPI, MPICH e MVAPICH (BAGCHI, 2015).

As principais vantagens de estabelecer um padrão de troca de mensagens são a portabilidade e facilidade de uso. Além disso, a definição de um padrão fornece aos fabricantes um conjunto bem definido de rotinas à serem implementadas, e em alguns casos eles podem fornecer suporte de hardware, aumentando, assim a escalabilidade do programa.

No padrão MPI, uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos. Inicialmente, na maioria das implementações, um conjunto fixo de processos é criado e distribuído entre diferentes processadores. Os processos podem usar mecanismos de comunicação ponto a ponto, que são operações para enviar mensagens de um determinado processo a outro. Um grupo de processos pode invocar operações coletivas (*collective*) de comunicação para executar operações globais. O MPI é capaz de suportar comunicação assíncrona e programação modular, através de mecanismos de comunicadores (*communicator*) que permitem ao usuário MPI definir módulos que encapsulem estruturas de comunicação interna.

2.4.2 OpenMP

OpenMP (*Open Multi-Processing*) é uma API (*Application Programming Interface*) multiplataforma que especifica um padrão de programação paralela, onde o paralelismo é feito através de *threads* e memória compartilhada. Ela foi lançada em 1997 pela OpenMP ARB (*Architecture Review Board*), que é uma organização sem fins lucrativos composta por empresas de hardware e software como AMD, ARM, Cray, Fujitsu, HP, IBM, Intel, Micron, NEC, Nvidia, Oracle Corporation e outras. Projetada para ser utilizada com C/C++ e Fortran, as especificações são desenvolvidas e mantidas pela ARB, que tem como objetivo fornecer uma abordagem portátil, de fácil utilização e eficiente para programação paralela com memória compartilhada.

Basicamente a OpenMP disponibiliza um conjunto de diretivas para o compilador, funções de biblioteca e variáveis de ambiente as quais especificam a implementação de um programa paralelo. O uso adequado desses recursos em um programa sequencial vai permitir que muitas, talvez a maioria, das aplicações se beneficiem da arquitetura paralela de memória compartilhada com o mínimo de modificações no código. Na prática, muitas das aplicações tem considerável paralelismo que podem ser exploradas (CHAPMAN; JOST; PAS, 2008).

A programação *multithread* com OpenMP tem uma série de vantagens sobre outras formas de implementação de paralelismo em arquiteturas de memória compartilhada, por exemplo, o Pthreads[5]. Dentre elas podemos citar a facilidade de compreender o uso das

[5] Pthreads é uma interface de manipulação de *threads* padronizada em 1995 pelo IEEE (IEEE POSIX 1003.1c).

diretivas de compilador, a maneira de se explorar o paralelismo e facilidade de converter incrementalmente um programa sequencial em paralelo.

2.5 Paralelismo Aplicado

Nesta seção serão discutido alguns conceitos sobre paralelismo, como algumas das métricas mais utilizadas na literatura para à analise de algoritmos paralelos.

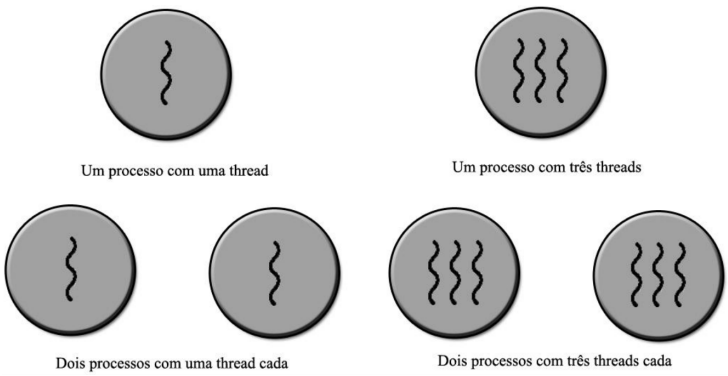
2.5.0.1 Threads e Processos

Processo é geralmente entendido como um programa em execução, porém, na realidade, trata-se de uma estrutura mais complexa. Conceitualmente um processo pode ser visto como um contêiner que armazena e controla um conjunto de recursos necessários à execução do programa. Esses recursos são: o espaço de endereçamento do programa executável, dados do programa, pilha, registradores, contador de programa, ponteiro de pilha, *signal handlers* (manipuladores de sinais), entre outros. Pô-los todos juntos na forma de um processo facilita o gerenciamento desses recursos (TANENBAUM, 2010).

Em virtude de mecanismos de proteção de hardware, processos são isolados entre si, ou seja, um processo p_1 não pode acessar diretamente dados de um outro processo p_2. Devido a esse isolamento, processos normalmente se comunicam com outros processos pelo envio de mensagens.

Threads representam outra característica de um processo, a capacidade de poder dividir o seu próprio fluxo de controle em duas ou mais tarefas, que podem ser executadas concorrentemente. Todo processo por padrão tem no mínimo uma *thread*, chamada de *master thread*. Basicamente *threads* apresentam as mesmas características de um processo, no entanto, a principal diferença é que o espaço de memória é compartilhado com todas as outras *threads* do processo, dessa forma, elas podem se comunicar diretamente escrevendo ou lendo na memória. A diferença entre *threads* e processos é ilustrada na figura 6.

Figura 6 – Diferença entre *threads* e processo



Fonte (ROSÁRIO, 2012).

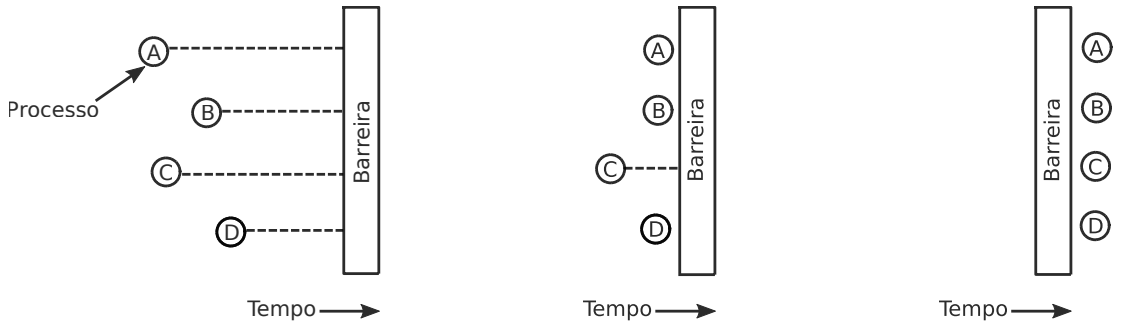
A programação com *threads* é conhecida como programação paralela de memória compartilhada. Em contra partida, a programação paralela com o uso de processos é geralmente referida como programação paralela por passagem de mensagens (LIN; SNYDER, 2008).

2.5.0.2 Barreira

Uma barreira é um método de sincronização para um grupo de processos ou *threads*. O seu uso se faz necessário quando desejamos dividir um programa em fases, onde um grupo de processos ou *threads* não podem seguir para a próxima fase até que todos os processos/*threads* tenham completado a fase atual. Para assegurar essa restrição é colocado uma barreira no final de cada fase. Quando um processo/*thread* alcança uma barreira ele fica bloqueado até que todos os processos/*threads* alcancem a barreira. A figura 7 ilustra o mecanismo de uma barreira.

Na figura 7a, é possível observar quatro processos se aproximando da barreira. Depois de um tempo, o primeiro processo termina toda a computação atribuída a ele para a primeira fase. Então ele executa a primitiva *barrier*. O processo é, então, suspenso. Mais tarde, a primeira fase é terminada por um segundo e depois por um terceiro processo, que também executam a primitiva *barrier*. Essa situação é ilustrada na figura 7b. Por fim, quando o último processo, *C*, atinge a barreira, todos os processos são liberados, conforme a figura 7c.

Figura 7 – Uso de uma barreira.



(a) Processos se aproximando de uma barreira

(b) Todos os processos, exceto um, estão bloqueados pela barreira.

(c) Quando o último processo chega à barreira, todos passam por ela.

Fonte: Adaptado de (TANENBAUM, 2010)

Em programas paralelos é comum a utilização de barreiras, mas o seu uso pode degradar o desempenho do programa, pois os primeiros processos/*threads* tem que esperar os outros alcançarem a barreira, antes de seguir para a próxima fase.

2.5.0.3 Região Crítica

Quando *threads* ou processos tentam acessar simultaneamente um recurso compartilhado na memória, e o acesso pode resultar em um erro, é dito que o programa tem uma **condição de corrida**. Isto é, a corretude do resultado vai depender de qual *thread* ganha a corrida. Por exemplo, suponha que duas *threads* estão tentando modificar uma mesma variável compartilhada x, com valor inicial igual a 5. Ambas têm a seguinte instrução:

$$x := x + 1.$$

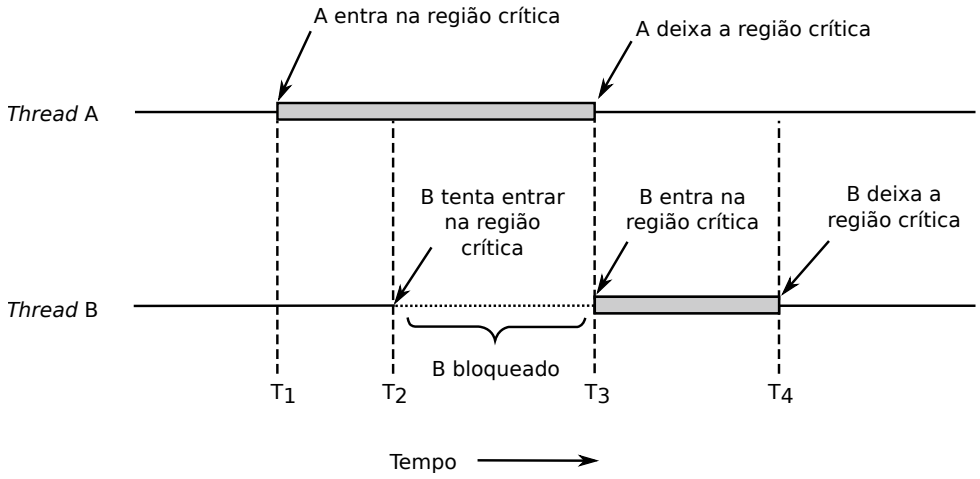
Agora suponha, que ambas as *threads* leram x ao mesmo tempo, e estão recebendo o mesmo valor 5 dentro do escopo de memória de cada um. Se for adicionado o valor 1 à variável x, as *threads* terão o valor 6, e consequentemente terão que escrever 6 de volta a variável compartilhada x. Sendo assim, o resultado final de x será igual a 6, ao invés de 7.

Então para que o programa execute corretamente, é necessário garantir que apenas uma *thread* execute o bloco de código de cada vez. Um bloco de código que pode somente ser executado por uma *thread* de cada vez é chamado de **região crítica** (PACHECO, 2011).

Existe diversas soluções para resolver problemas de região crítica, uma delas é o uso de mecanismos de exclusão mútua como um *mutual exclusion* ou *mutex* ou *lock*. Um *mutex* é um tipo especial de dado que garante a uma região crítica ser executada somente por uma *thread* de cada vez. Ao entrar em região crítica a *thread* deve obter um *mutex*, bloqueando assim a entrada de outras *threads* até que ela saia da seção crítica.

Na figura 8 é ilustrado o uso da exclusão mútua, onde temos duas *threads* executando em paralelo. Quando a *thread* A chega a região crítica no tempo T_1 ele obtém um *mutex*, e então bloqueia a entrada de novas *threads*. Um pouco depois, a *thread* B tenta entrar na região crítica, mas é bloqueada. Então, a *thread* B aguarda a *thread* A deixar a região crítica no tempo T3, permitindo que B entre imediatamente. Por fim, B sai em T_4 voltando a situação inicial, sem *threads* em suas regiões críticas.

Figura 8 – Exclusão mútua usando regiões críticas.



Fonte Adaptado de (TANENBAUM, 2010).

Embora o mutex seja uma forma eficiente de garantir a exclusão mútua, seu uso pode reduzir severamente o desempenho, principalmente se a verificação de disponibilidade de acesso estiver dentro de um loop realizada com certa frequência. A independência de acesso entre threads nem sempre pode ser implementada, mas quando é possível acabará sendo mais eficiente.

2.5.0.4 Overhead

Ao utilizar, por exemplo, dois núcleos para que um programa seja executado, pode-se razoavelmente esperar que um programa seja executado duas vezes mais rápido. No entanto, em programas paralelos este é raramente o caso, devido a uma variedade de *overheads* gerais associados ao paralelismo (GRAMA et al., 2003). De maneira geral, o *overhead* em um programa paralelo pode ser caracterizado por todo trabalho adicional que não existia em um programa sequencial.

Os *overheads* são provinientes da interação entre processos, *threads*, ociosidade e excesso de computação. Eles são uma combinação direta ou indireta de tempo excessivo do uso de computação, de memória, de largura de banda, ou de outros recursos durante algum objetivo computacional. Ou seja, *overhead* é a quantidade de tempo requerido para coordenar as tarefas em paralelo, ao invés de fazer o trabalho útil. *Overhead* paralelo pode incluir fatores como: o tempo gasto em comunicação entre elementos de processamento, sincronização de tarefas, presença de regiões críticas, desequilíbrio de carga entre processadores, tempo de criação de um processo ou *thread*, preempção[6], entre outros.

[6] Preempção é a capacidade de alterar a ordem de (ou tirar de execução) um processo em detrimento de outro com uma prioridade maior, fazendo com que o processo atual fique bloqueado.

2.5.1 Métricas de Escalabilidade Paralela

Além de reduzir o tempo de execução, uma questão essencial é estudar o desempenho dos programas paralelos, entendendo até que ponto é vantajoso utilizá-los. Esse tipo de análise tem o objetivo de determinar os benefícios do paralelismo aplicado a um problema, e, também, o quanto o algoritmo é capaz de continuar eficiente.

Antes de discutir as métricas, deve-se perceber que todo algoritmo possui uma fração do código inerentemente sequencial, que não pode ser paralelizada. Isto implica, que a aceleração dos algoritmos paralelos serão limitados por essa porção sequencial. Em vista disso, uma série de análises deve ser utilizada para que, com base no resultado alcançado e desejado, sejam realizados ajustes com o intuito de reduzir ao máximo as partes sequencias e aumentar a parte paralela do código. Sendo assim, as próximas subseções tem como objetivo abordar as principais métricas de algoritmos paralelos.

2.5.1.1 Speedup e Eficiência

O ***speedup*** avalia o quão mais rápido um algoritmo paralelo é em relação ao seu correspondente algoritmo sequencial, podendo ser calculado através da razão entre o tempo de execução do algoritmo sequencial (T_S) e tempo de execução da versão paralela, usando P processadores/*threads* (T_P). Sendo assim, o *speedup* pode ser definido pela seguinte equação:

$$S_P = \frac{T_S}{T_P} \tag{2.7}$$

Idealmente, o *speedup* deveria ser P vezes mais rápido que o sequencial, ou seja, $S = P$. Por exemplo, caso seja utilizado 2 *threads* para realizar algum processamento paralelo, o ganho deveria ser 2 vezes mais rápido. Se utilizar 4, então 4 vezes mais rápido e assim por diante. Quando isso acontece, é dito que o programa paralelo tem um *speedup* linear. Quando $S < P$, diz que o *speedup* é sublinear, que é forma mais comum de *speedup*. Já em circunstâncias muito raras, quando o *speedup* excede o número de núcleos, sendo $S > P$, o *speedup* é conhecido como super linear (BRESHEARS, 2009).

Na prática, o *speedup* linear dificilmente é alcançado, devido a adição de *overheads* não existentes no algoritmo sequencial, como: sobrecarga de comunicação e sincronização entre processos e partes do código que são inerentemente sequenciais, forçando o uso de mecanismos de exclusão mútua, como um *mutex*. Sendo assim, é incomum que programas paralelos obtenham um *speedup* linear. Além disso, o problema tende a se agravar com o aumento do número de *threads* ou processos, ou seja, mais *threads* implicam em mais acessos a uma região crítica.

Já a super linearidade, embora, seja algo raro, pode acontecer em alguns casos específicos. Ela geralmente acontece quando o trabalho realizado por um algoritmo sequencial

é maior do que a sua formulação paralela ou devido a recursos de hardware que colocam a implementação sequencial em desvantagem. Por exemplo, os dados de um problema podem ser demasiado grande para caber dentro da memória cache de um único processador, degradando desse modo o seu desempenho, devido a utilização de memória mais lentas, o que nos leva ao já conhecido problema do gargalo de von Neumann. Contudo, quando dividido entre vários processadores, as partições de dados individuais seriam pequenas o suficiente para caber em suas respectivas memórias cache, fornecendo assim um aumento significativo de desempenho, o que muitas vezes pode provocar a super linearidade.

Considerando que o *speedup* nos fornece uma métrica para determinar o quão mais rápido um programa paralelo é em relação a um programa sequencial, a **eficiência** diz o quanto estão utilizando os processadores para executar um algoritmo paralelo. A eficiência pode ser expressa por:

$$E = \frac{S}{P} = \frac{(\frac{T_S}{T_P})}{P} = \frac{T_S}{P \cdot T_P} \tag{2.8}$$

Como resultado a eficiência retorna a média percentual de uso de cada processador. Por exemplo, se um *speedup* = 53 sobre 64 *threads*, a eficiência será de 82.8% (53/64 = 0.828). No caso ideal (*speedup* = P), a eficiência corresponderia para $E_P = 1 = 100\%$, e no melhor caso a eficiência pode exceder 1, o que caracteriza uma eficiência super linear. O *speedup* cresce com o aumento da quantidade de processos/*threads*, contudo tende a saturar após certo ponto. A eficiência tende a diminuir com o aumento da quantidade de processos/*threads*, isso ocorre devido a própria definição de eficiência mostrada na equação 2.8 ($E = S/P$)

Uma outra característica da eficiência, é que ela tende a diminuir com o aumento do número de processos/*threads*.

2.5.2 Lei de Amdahl

Em 1967, o arquiteto da IBM Gene Amdahl enunciou o que hoje é conhecido como a Lei de Amdahl. Ele propôs a existência de um limite máximo que o *speedup* adquirido por paralelismo em um problema de tamanho fixo pode obter, por mais que seja utilizado uma quantidade demasiadamente grande de processadores. Segundo Amdahl, a pequena porção do programa que não pode ser paralelizada limitará o aumento da velocidade obtida através do paralelismo. Como consequência, haverá um limite onde o processamento paralelo pode ser aplicado.

O tempo total de execução de um programa paralelo é dado pela soma do tempo da fração paralela do código com o tempo da fração sequencial, como mostra a equação

abaixo:

$$T_P = (1 - f)T_S + f(\frac{T_S}{N}) \tag{2.9}$$

onde, T_P é o tempo paralelo, T_S é o tempo sequencial, f é a fração paralela do código, $(1 - f)$ é a fração sequencial e N é o número de processadores.

Ao manipular a equação 2.9, obtemos o *speedup* definido pela lei de Amdahl, que é dado por:

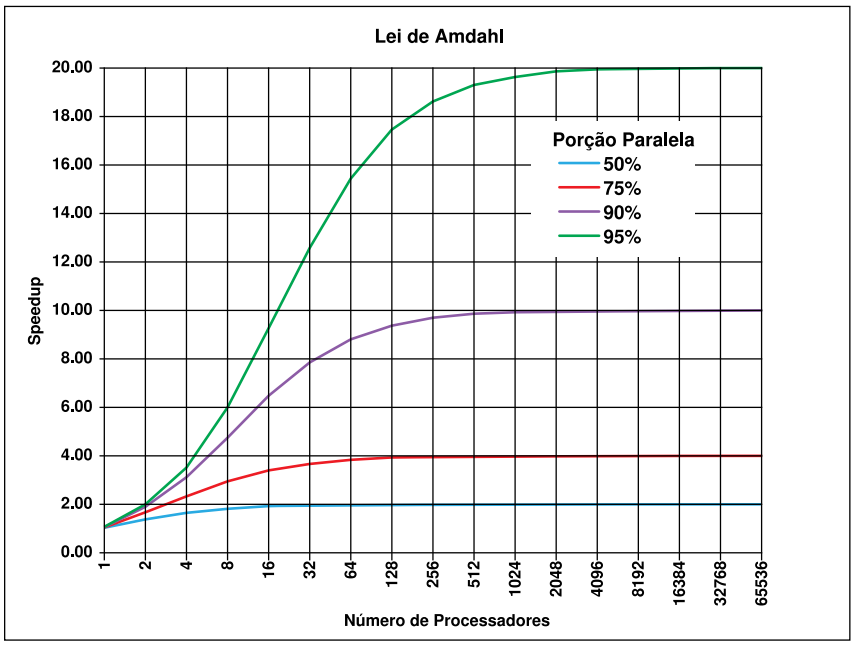
$$S(N) = \frac{T_S}{T_P(N)} = \frac{N}{(1 - f)N + f} = \frac{1}{(1 - f) + \frac{f}{N}} \tag{2.10}$$

De acordo com a equação 2.10 quando N tende ao infinito, a parte paralelizável é anulada, e o *speedup* máximo tende a ser

$$S \approx \frac{1}{r} \tag{2.11}$$

onde, $r = (1 - f)$, ou seja, a porção sequencial do algoritmo. Logo, a lei de Amdahl diz, que se uma fração r do código do programa sequencial permanece não paralelizável, então o *speedup* nunca será melhor do que $1/r$.

Figura 9 – *Speedup* de acordo com a Lei de Amdahl



Fonte: Adaptado de <https://pt.wikipedia.org/wiki/Lei_de_Amdahl>

A figura 9 demonstra como o *speedup* de um programa usando múltiplos processadores em computação paralela é limitado pela fração sequencial do programa. Por exemplo,

se f é 90%, então $(1 - f)$ é 10%, então o problema pode ter um *speedup* de no máximo 10, não importando o quão grande seja o valor de N. E, como esperado, os maiores *speedups* ocorrem quando a fração sequencial é menor. Também é possível observar que o *speedup* tende a saturar quando N aumenta, o que é comum em programas paralelos.

A fim de obter um programa mais rápido, pode-se concluir que é necessário reduzir ao mínimo a fração que não pode ser paralelizada, para assegurar o equilíbrio de carga das tarefas ao nível do processador, e também para minimizar o tempo dedicado a comunicação e sincronização.

2.5.3 Escalabilidade

Apesar de o paralelismo ter fascinado muitos pesquisadores há pelo menos três décadas, a programação paralela sempre foi, de forma geral, tida como uma simples alternativa da programação sequencial. Como consequência disso, atualmente, a maioria dos algoritmos não suporta paralelismo. Portanto, dentre os muitos desafios da era *multicore*, um dos mais iminentes é a identificação de algoritmos com problemas de escalabilidade de desempenho. Para que os avanços da era *multicore* sejam traduzidos em mais desempenho é necessário que o algoritmo suporte a tendência de avanços dos *microchips* multiprocessados através de execução paralela.

2.5.3.1 Escalabilidade de Amdahl

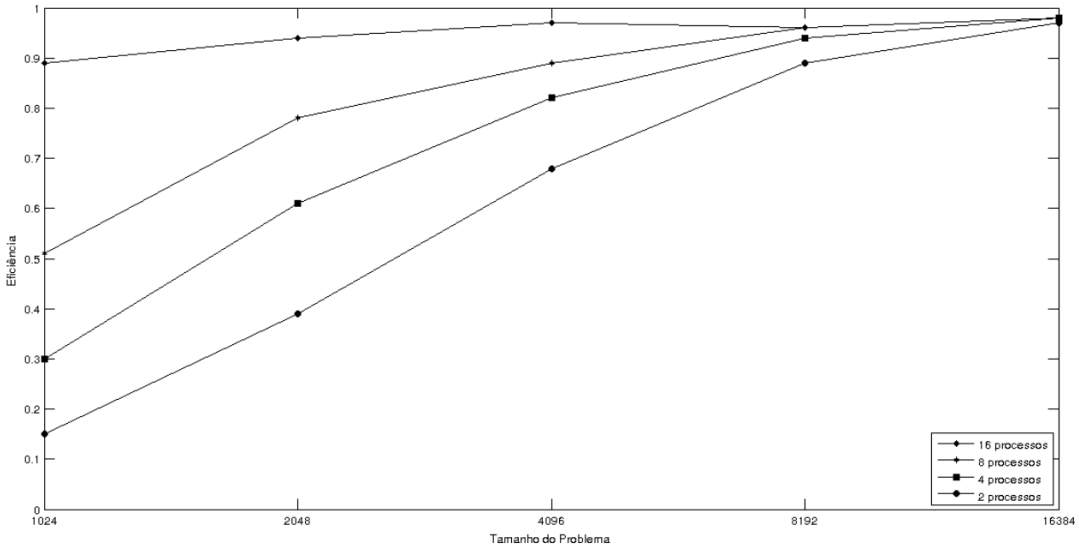
Segundo a Lei de Amdahl, a velocidade de um algoritmo paralelo será limitada pela sua porção serial. Como consequência, haverá um limite, onde o processamento paralelo pode ser aplicado. Os algoritmos escaláveis por Amdahl terão um comportamento semelhante ao ilustrado na figura 9, onde ao executar um algoritmo paralelo com um problema de tamanho fixo, o *speedup* do algoritmo tende a melhorar com o aumento do número de processadores. Todavia, o *speedup* possui um valor máximo devido à porção serial do código, dessa forma, se faz necessário otimizar a fração serial do algoritmo para que ele obtenha maiores *speedups*, e como consequência uma melhor escalabilidade.

2.5.3.2 Escalabilidade de Gustafson

Devido às limitações teóricas previstas na Lei de Amdahl, diversos pesquisadores das décadas de 1970 e 1980 evitaram construir máquinas paralelas massivas, acreditando que o ganho seria limitado. No entanto, em um estudo realizado por Gustafson, intitulado *Reevaluating Amdahl's Law* (GUSTAFSON, 1988), ficou provado que somente a fração serial do algoritmo paralelo não é o suficiente para limitar o seu *speedup*. Ao comparar diversos experimentos, Gustafson, notou que o *speedup* não estava limitado pela parte serial do algoritmo, na realidade, o paralelismo cresceu de acordo com o tamanho do problema.

Desta maneira, Gustafson percebeu que ao aumentar o tamanho do problema, o percentual da porção serial do algoritmo diminuía, resultando em um aumento da eficiência obtida através do paralelismo. Logo, para se comprovar a escalabilidade de um algoritmo paralelo por Gustafson, é necessário aumentar o tamanho do problema e verificar se há um aumento na eficiência. A figura 10 ilustra um algoritmo escalável por Gustafson.

Figura 10 – Eficiência de um programa paralelo escalável por Gustafson



Fonte: Adaptado de (PACHECO, 2011)

3 Algoritmo Ant Colony System Paralelo

Este capítulo trata da implementação do algoritmo *Ant Colony System* (ACS) aplicado ao Problema do Roteamento de Veículos Capacitado (CVRP) utilizando a linguagem de programação C++ e o modelo de programação paralela OpenMP.

Para facilitar o entendimento do pseudocódigo da seção 3.2, a figura 11 ilustra as duas principais estruturas de dados utilizada no algoritmo. No intuito de tirar um maior proveito das linhas de cache foi utilizada a técnica de substituir matrizes por ponteiros de vetores simulando matrizes ($matriz[iz][ix] = vetor[iz * nz + ix]$), permitindo o acesso sequencial aos elementos das matrizes na memória.

Figura 11 – Estrutura de dados utilizada no algoritmo

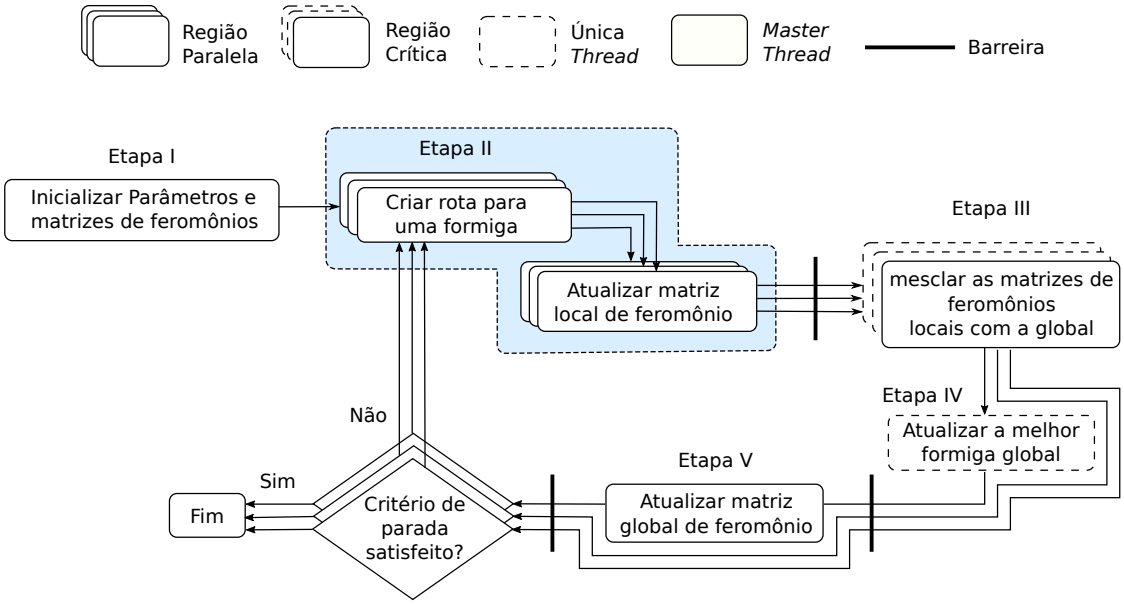
```
1  struct formiga {
2      consumidor *rota;     // vetor que armazena todos os clientes da rota.
3      bool *visitado;       // memória da formiga, armazena todos os consumidores visitados.
4      int tamanho_da_rota;  // armazena o tamanho da rota atual da formiga.
5      int tam_visitado;     // armazena o tamanho do vetor: visitado.
6      int capacidade_total; // capacidade máxima da formiga.
7      int capacidade_atual; // capacidade atual da formiga, durante a fase de construção
8      int custo_da_rota;    // custo da rota.
9  };
10
11 struct feromonio {
12     double *matriz_de_feromonio;  // matriz de feromônio, usando a notação de vetor.
13     double *matriz_de_escolha;    // matriz de informação de escolha, usando a notação de vetor.
14     double trilha_0;              // valor inicial da trilha de feromônio.
15 };
16
```

Fonte: Autoria própria.

3.1 Esquema Geral de Paralelização

O algoritmo exposto neste trabalho é baseado na meta-heurística ACS apresentada na seção 2.2.1.3. De acordo com a figura 12 o algoritmo pode ser dividido em cinco etapas.

Figura 12 – Fluxograma do algoritmo ACS Multicore



Fonte: Autoria própria.

A primeira etapa consiste em uma fase de inicialização de dados onde inicialmente é realizada a leitura do problema a ser resolvido, e posteriormente é calculado a matriz de distâncias, que armazena a distância euclidiana entre os consumidores; após a criação da matriz de distâncias, é criada uma solução inicial aleatória para o problema; a partir do custo da solução inicial é calculado o valor inicial da matriz de feromônio e da informação heurística.

A segunda etapa consiste de um laço paralelizado, em que cada *thread* gerencia um grupo de formigas, uma vez que, cada formiga tem como responsabilidade a construção de uma solução completa para o CVRP. A fase de construção é regida de acordo com a regra definida na equação 2.4, aplicando as restrições impostas pelo CVRP, abordadas na seção 2.1.1. Ao finalizar a construção da solução, a formiga deverá realizar a atualização local da trilha de feromônios definida pela equação 2.5. No entanto, ao paralelizar o ACS, a atualização local da trilha de feromônios pode ser considerado o maior gargalo do algoritmo, devido todas as formigas atualizarem uma mesma matriz compartilhada, e como já explicado anteriormente, isso caracteriza uma região crítica. Então, para minimizar esse problema, e com um custo adicional de memória, foi criado uma matriz de feromônio para cada *thread*, ou seja, uma matriz para um grupo independente de formigas. Dessa forma, foi possível minimizar a quantidade de sincronização que essa etapa teria. Ao final dessa etapa existe uma barreira, uma vez que, para a correta execução da próxima etapa é necessário que todas as formigas tenham finalizado esta fase.

Por mais que a região crítica da etapa II tenha sido eliminada, para remover a região crítica do laço, ela somente foi postergada à etapa III, onde todas as *threads* locais

realizam uma intersecção das matrizes locais com a matriz global de feromônios. Durante essa fase, somente as arestas percorridas pelas formigas são intercaladas com a matriz global de feromônio.

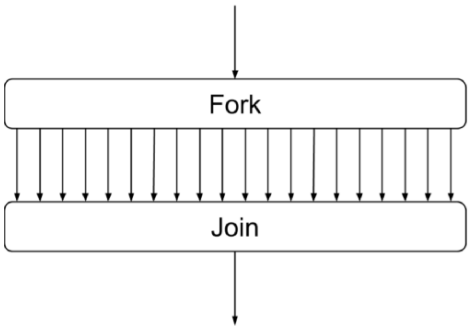
A quarta etapa é executada por apenas uma *thread*, que consiste em atualizar a melhor formiga global. Se o custo da rota da melhor formiga da iteração atual for menor que o custo da formiga global, a rota da formiga global será atualizada com a rota da melhor formiga da iteração. Como a execução da quinta etapa depende da melhor formiga global, ao final dessa etapa existe uma barreira para sincronizar todas *threads*.

A etapa V é somente executada pela *master thread*, que tem como objetivo realizar a atualização global das trilhas de feromônios, conforme a equação 2.6, a partir da melhor formiga encontrada na etapa anterior. No final dessa fase também tem uma barreira, para que todas as formigas iniciem a próxima fase com a matriz de feromônio atualizada. Logo após atualização global, também, é verificado se o critério de parada foi satisfeito, que nesse caso, é utilizado um número fixo de iterações.

3.2 Implementação do Algoritmo Ant Colony System Multicore

A paralelização da meta-heurística ACS foi feita em C++ utilizando o OpenMP (*Open Multi-Processing*), o qual tem como base o modelo *fork* (bifurcar) e *join* (unir). Existe um fluxo de execução principal chamado *master thread* que ao chegar em uma definição de região paralela, ela cria um conjunto de *threads* e passa ser a *thread* mestre. A partir do início da região paralela, todas as *threads* começam a executar o código da região paralela, e ao fim da seção é feito um *join* (ver figura 13).

Figura 13 – Modelo de Execução *Fork-Join*



Fonte: (COUTINHO, 2014).

Após a fase de inicialização dos dados, que é executada fora da região paralela, dá-se inicio as modificações necessárias para tornar o algoritmo paralelo. Para criar um programa paralelo com o OpenMP, a primeira coisa a se fazer é declarar uma região paralela

através da diretiva `#pragma omp parallel`, onde será criado um grupo de *threads*, como mostrado na figura 13. Dentro de uma região paralela serão criadas N *threads*, que por padrão, é a mesma quantidade de núcleos disponíveis. No entanto, caso o programador necessite especificar a quantidade de *threads*, ele pode fazer isso chamando a função `omp_set_num_threads`. Esta chamada tem prioridade sobre a determinação do sistema em relação ao número de núcleos disponíveis, podendo assim criar um número de *threads* maior que o número de núcleos.

Figura 14 – Pseudocódigo do algoritmo ACS paralelo

```
 1  omp_set_num_threads(num_threads);
 2  #pragma omp parallel escopo_de_variaveis
 3  {
 4    tid = omp_get_thread_num();
 5    for (iteracao = 0; iteracao < num_max_iteracoes; iteracao++) {
 6        //ETAPA II
 7        #pragma omp for
 8        for (k = 0; k < num_formigas; k++) {
 9          // Cada formiga cria uma rota, de acordo com a equação (2.4).
10          criar_rota(formigas[k], estrutura_global_feromonio, instancia_problema);
11          // Cada thread tem a uma matriz local que um grupo de formigas pode acessar.
12          atualizar_matriz_local_feromonio(estrutura_local_feromonio[tid], formigas[k].rota);
13        }
14        // ETAPA III
15        // Mesclar as matrizes de feromônios locais com a global.
16        #pragma omp for nowait
17        for (k = 0; k < num_formigas;  k++) {
18          for (i = 0; i < formigas[k].tamanho_da_rota - 1; i++) {
19            // Calcula o nó/índice a ser atualizado;
20            #pragma omp atomic
21            // Intercala a matriz local com a matriz global de feromônios.
22            estrutura_global_feromonio.matriz_feromonio[no] +=
23                    (1.0 / num_threads * estrutura_local_feromonio[tid].matriz_feromonio[no]);
24            #pragma omp atomic
25            // Intercala a matriz local com a matriz global de informação de escolhas.
26            estrutura_global_feromonio.informacao_heuristica[no] +=
27                    (1.0 / num_threads * estrutura_local_feromonio[tid].info_escolhas[no]);
28          }
29        }
30        // ETAPA IV
31        // Atualizar a melhor formiga/rota
32        #pragma omp single
33        {
34          // Seleciona a formiga que teve a rota com menor custo.
35          if (melhor_formiga_iteracao.custo_da_rota < melhor_formiga.custo_da_rota) {
36            // Copia a rota e o custo da rota da melhor formiga da iteração para a melhor fomiga.
37          }
38        }
39        // ETAPA V
40        // Atualizar matriz global de feromônio
41        #pragma omp master
42        atualizar_matriz_global_feromonio(estrutura_global_feromonio, melhor_formiga);
43        #pragma omp barrier
44    }
45  }
```

Fonte: Autoria própria.

Na linha 2 (figura 14) foi definido o "escopo de variáveis", que se refere ao nível de compartilhamento das variáveis entre as *threads* numa região paralela. No OpenMP existem seis diferentes tipos de escopo, que são:

- **shared**: os dados dentro de uma região paralela são compartilhados, o que significa visível e acessível por todas as *threads* em simultâneo.

- **private**: os dados em uma região paralela são individuais para cada thread, com este recurso cada thread terá uma cópia local e será utilizada como uma variável temporária. Uma variável não é inicializada e o valor não é mantido para o uso fora da região paralela;

- **none**: força o programador a declarar cada variável na região paralela utilizando as cláusulas atribuídas ao compartilhamento de dados (*data sharing attribute clauses*);

- **firstprivate**: como *private* exceto ao inicializar pelo valor original;

- **lastprivate**: como *private* exceto que o valor original é atualizado depois da construção.

A diretiva `#pragma omp for` (linha 7; figura 14) especifica que as iterações de um "loop" sejam distribuídas e executadas em paralelo pelo grupo de *threads*. Por padrão, o compilador irá dividir o "loop" de maneira contínua e em partes iguais (quando possível), para cada *thread*. Ao final do *loop* existe uma barreira implícita que sincroniza todas as *threads*, impedindo que o resto do programa seja executado até que todas as formigas tenham criado uma rota e feito a atualização local da matriz de feromônios. Como foi visto na seção 2.5.0.4, o uso de barreiras introduz *overhead* no programa, então caso não exista a necessidade de uma barreira ao final do "loop", é possível retirá-la adicionando o atributo `nowait`.

Na etapa III ocorre a sincronização entre a matriz local de feromônios, de cada *thread*, com a matriz global de feromônios. Percebe-se que ao fazer a sincronização dos dados, ocorre a existência de uma região crítica com duas operações independentes, uma na linha 22 e outra na linha 26. Para resolver esse tipo de problema, o OpenMP disponibiliza duas diretivas chamadas de: `#pragma omp critical` e `#pragma omp atomic`. A diretiva *critical* garante que um bloco de código será executado por somente uma *thread* de cada vez. Já a diretiva *atomic* que pode ser considerado um "mini-critical", pois garante que apenas uma posição de memória seja atualizada por uma *thread* de cada vez. A vantagem de utilizar a cláusula *atomic* é que ela recebe otimizações de compilador, tendo assim, um melhor desempenho que a diretiva *critical*. E como os dados a serem atualizados não possuem dependências entre eles, é possível que duas *threads* diferentes estejam atualizando essas variáveis ao mesmo tempo, ou seja, enquanto uma *thread* atualiza a variável referente a linha 22, outra *thread* pode estar atualizando a variável da linha 26. Vale ressaltar, que se o programa tem muitas regiões críticas, o uso da diretiva *atomic* pode ser muito custosa, sendo assim, a cláusula *critical* a melhor opção. Como não existe dependências de dados entre está etapa e a próxima, a barreira implícita do `omp for` é removida.

A cláusula `#pragma omp single` (linha 32; figura 14) específica que um bloco de código será executado por apenas uma *thread*, que nesse caso, seria utilizado para encontrar

a melhor formiga da iteração. Se o custo da melhor formiga da iteração for melhor do que o custo da melhor formiga global, então a sua rota será copiada para a melhor formiga global. Ao final da diretiva *single* também existe uma barreira implícita, que pode ser removida com o *nowait*.

A quinta e última etapa do algoritmo consiste em realizar a atualização global da matriz de feromônios, onde serão atualizadas todas as arestas pertencentes a rota da melhor formiga global. A fase de atualização global, é executada somente pela *master thread*, através do uso da cláusula `#pragma omp master` (linha 41; figura 14). A diretiva *master* não tem barreira implícita, devido a isso, foi colocado uma barreira explícita na linha 43 (`#pragma omp barrier`), para que na próxima iteração, todas as *threads* utilizem a matriz de feromônios atualizada.

4 Experimentos e Resultados

Para a realização dos experimentos, a plataforma computacional usada é baseada em um sistema com dois processadores Intel® Core™ i3-23330M com dois *cores* 2.20GHz, 4GB DDR3 RAM, 2 x 64KB de cache L1, 2 x 256KB de cache L2, 1 x 3MB de cache L3, contendo o sistema operacional *Arch Linux* de 64 bits. O conjunto de problemas utilizados nos experimentos foram retirados do *benchmark* CVRPLIB (2015), e ilustrados na tabela 1:

Problema	Núm. consumidores	Núm. veículos	Capacidade	Melhor solução conhecida
M-n121-k7	121	7	200	1034
X-n256-k16	256	16	1225	18880
X-n524-k153	524	153	125	154594
X-n655-k131	655	131	5	106780
X-n1001-k43	1001	43	131	72742

Tabela 1 – Instâncias dos problemas usados neste estudo.

Por convenção, daqui em diante, os problemas serão referenciados apenas pelo número de consumidores do problema. Para a realização dos experimentos foram realizados testes com 2, 4, 6 e 8 *threads* e com 250, 500 e 1000 formigas. Cada caso de teste foi executado 10 vezes, combinando o número de *threads* e de formigas.

Os valores dos parâmetros utilizados podem ser visualizados na tabela 2:

Parâmetro	valor
α	1.0
β	2.0
ρ	0.1
ε	0.1
q_0	0.9

Tabela 2 – Parâmetros utilizados.

4.1 Análise dos Valores Obtidos

Essa seção apresenta os custos das soluções do Problema do Roteamento de Veículos Capacitado (CVRP), obtidos através da meta-heurística *Ant Colony System*.

Os dados apresentados na tabela 3, estão organizados da seguinte forma: custo da melhor solução conhecida (CMSC); custo médio da solução do algoritmo sequencial (CMSS); custo médio da solução do algoritmo paralelo (CMSP); desvio padrão do custo

da solução serial, em relação a melhor solução (DPSS); desvio padrão do custo da soluções paralela, em relação a solução serial (DPSP).

Formigas	Dimensão	CMSC	CMSS	CMSP	DPSS	DPSP
250	121	1034	1190	1182	110	5
500	121	1034	1190	1177	110	9
1000	121	1034	1177	1174	101	2
250	256	18880	24755	23639	4154	789
500	256	18880	24230	23738	3783	347
1000	256	18880	24398	23621	3901	549
250	524	154594	208877	213159	38383	3027
500	524	154594	208875	213013	38382	2926
1000	524	154594	208044	202358	37794	4020
250	655	106780	113026	111760	4416	895
500	655	106780	112830	111577	4277	886
1000	655	106780	112866	111606	4303	890
250	1001	72477	101924	94086	20822	5542
500	1001	72477	102045	93707	20907	5895
1000	1001	72477	102906	93409	21516	6715

Tabela 3 – Valores médios das soluções obtidas através da meta-heurística ACS

Ao analisar a tabela, percebe-se que para a maioria dos casos teste, a solução obtida ainda está muito distante da ótima. O problema que mais se aproximou do valor ideal, foi o de 121 consumidores com 1000 formigas, onde o desvio padrão entre o custo da melhor solução e a solução sequencial é de 101 unidades. Um dos possíveis motivos da baixa qualidade das soluções, deve-se ao fato de que não foi implementado nenhum algoritmo de busca local, uma vez que, as melhores soluções são comumente encontradas através do uso desse recurso.

4.2 Análise de Escalabilidade

Nesta seção, será analisado a escalabilidade do algoritmo paralelo de acordo com as teorias de escalabilidades propostas por Amdhal e Gustafson.

4.2.1 Escalabilidade de Amdahl

Baseando-se na teoria discutida nas seções 2.5.2 e 2.5.3.1 referente ao trabalho de Amdahl, foram elaboradas as seguintes figuras 15, 16, 17, que mostram graficamente o *speedup* dos problemas usando 250, 500 e 1000 formigas respectivamente.

A figura 15 mostra um bom *speedup* para o problema com 1001 consumidores, uma vez que para 2 *threads* ele apresenta a super linearidade, sendo 2.06 vezes mais rápido que o algoritmo sequencial. Com 4 *threads* e 250 formigas, o *speedup* ficou próximo de 3,

ou seja, o algoritmo paralelo foi aproximadamente 3 vezes mais rápido que o algoritmo sequencial.

Figura 15 – Escalabilidade de Amdahl para 250 formigas

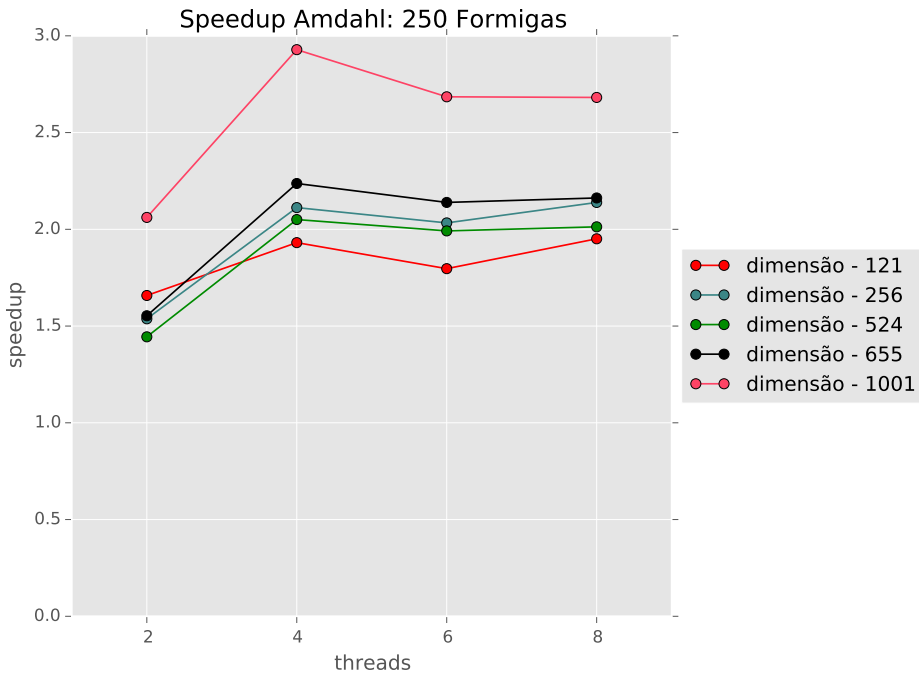


Figura 16 – Escalabilidade de Amdahl para 500 formigas

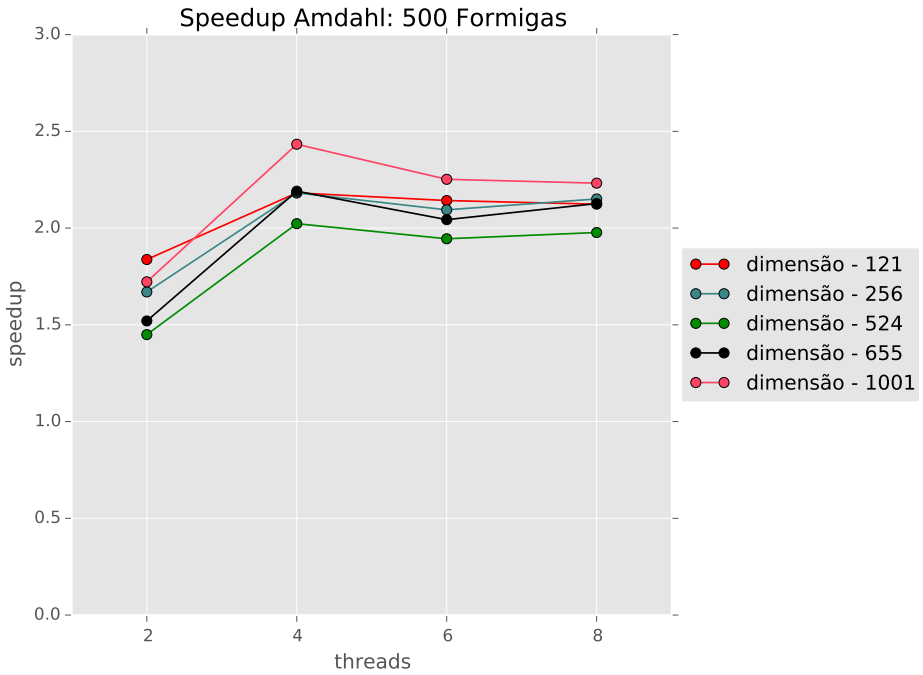
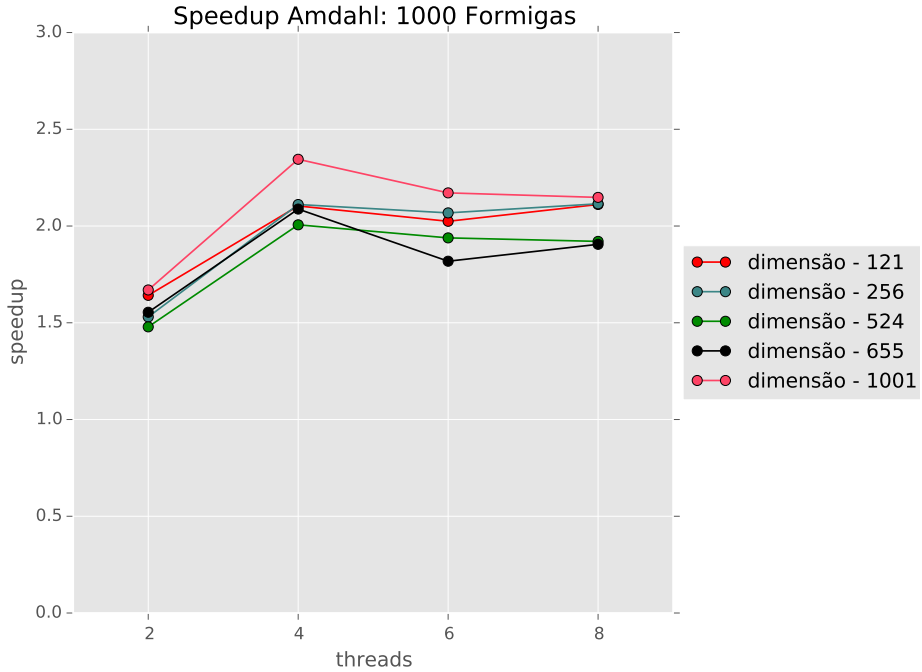


Figura 17 – Escalabilidade de Amdahl para 1000 formigas



Nas figuras 16 e 17, a curva do gráfico apresenta uma forma semelhante a da figura 15, no entanto, o *speedup* se mostra inferior. Um dos possíveis motivos para a diminuição do *speedup*, pode estar relacionado ao número de formigas. Quando se aumenta a quantidade formigas, está aumentando a quantidade de comunicação, se aumenta a comunicação, aumenta o *overhead*, degradando assim o *speedup*.

Perceba que quando chega a 4 *threads* se tem um *speedup*, em um ponto maior, isso acontece, devido, o computador ter 4 núcleos, então nesse momento é utilizado o poder total do paralelismo. Vale ressaltar que o *speedup* teve uma queda, para os teste com mais de 4 *threads*, por que o computador, tem somente 4 núcleos.

Sendo assim, de acordo com a lei de Amdahl, os gráficos 15, 16 e 17, aparesentam a escalabilidade de Amdahl, pois o seu *speedup* tende a aumentar quando o número de núcleos aumenta.

4.2.2 Escalabilidade de Gustafson

Nesta seção será abordado os aspectos da escalabilidade de Gustafson, apresentado na seção 2.5.3.2. As figuras 18, 19 e 20 apresentam os gráficos dos problemas usando 250, 500 e 1000 formigas.

Na figura 18, percebe-se que a eficiência está aumentando conforme a dimensão dos problemas aumentam, o que caracteriza a escalabilidade de Gustafson, descrita na seção 2.5.3.2. Nota-se que o algoritmo paralelo tem melhor eficiência para problemas maiores

que 256, onde apresenta uma melhora significativa em relação ao algoritmo sequencial. O gráfico também demonstra uma eficiência super linear, para o problema de tamanho 1001 executado com 2 *threads*, ou seja, com 2 *threads* é como se o algoritmo estivesse utilizando mais de 2 processadores, tendo uma eficiência acima de 1.0. Já o caso de teste com 4 *threads* e dimensão igual a 524, o algoritmo está utilizando apenas metade do poder de processamento disponível.

Figura 18 – Escalabilidade de Gustafson para 250 formigas

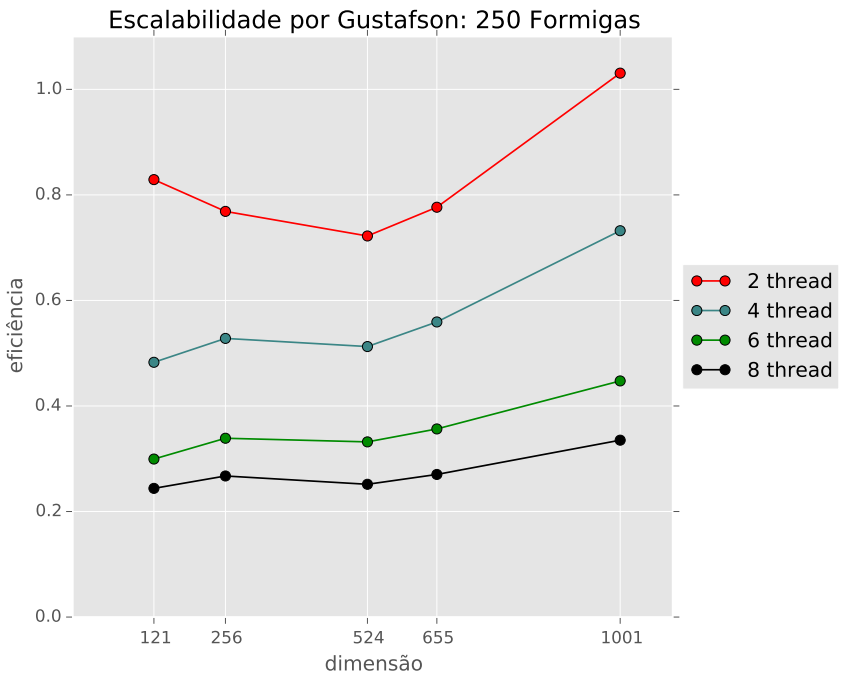


Figura 19 – Escalabilidade de Gustafson para 500 formigas

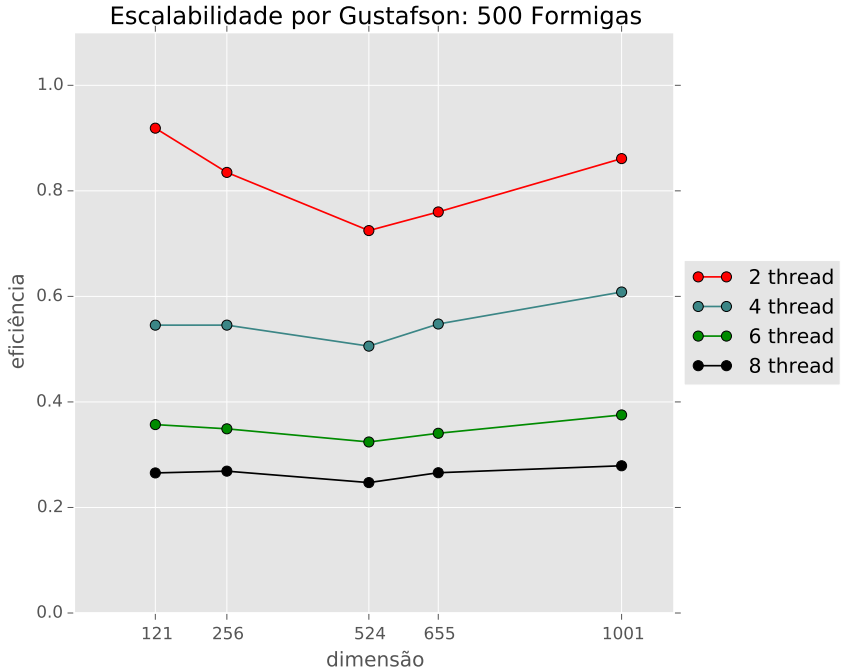
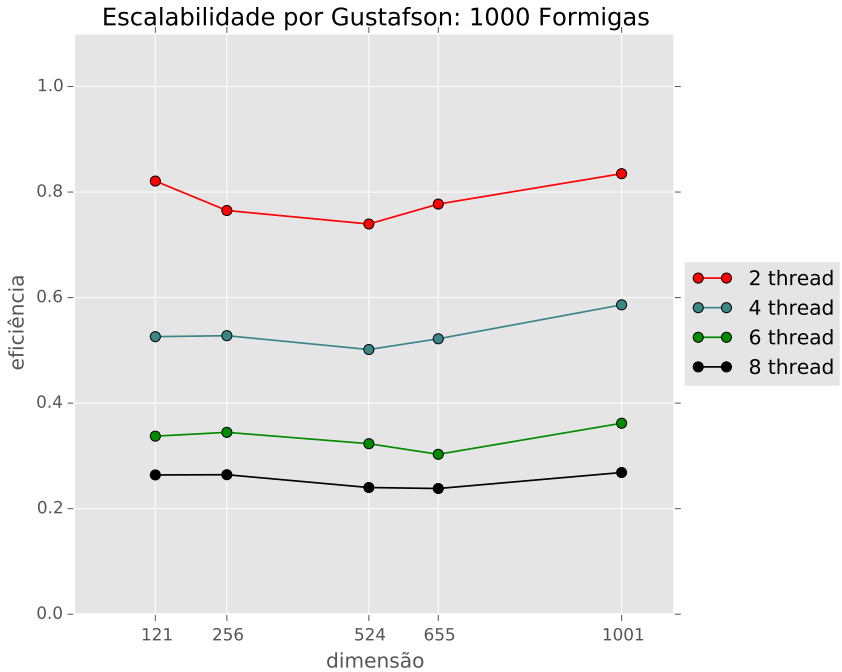


Figura 20 – Escalabilidade de Gustafson para 1000 formigas



Assim como o gráfico exposto na figura 18, a figura 19 apresenta uma eficiência sublinear e tende a crescer ao aumentar o tamanho do problema. Nota-se que o gráfico apresenta uma melhora na eficiência para o conjunto de problemas com a dimensão de 524, 655 e 1001. Já a figura 20 apresenta uma eficiência semelhante a figura 16, no entanto

tem uma perca de eficiência, que pode ser decorrente da grande quantidade de formigas, como já explicado anteriormente.

Nota-se que, aumentar o número de formigas implica na redução da eficiência. O aumento da quantidade de formigas contribuí para o crescimento da porção serial do algoritmo, ou seja, tem um número maior de formigas disputando a região crítica do algoritmo, o que tem como resultado um maior tempo de espera para entrar na região. Quando isso acontece, as formigas não realizam nenhuma computação, prolongando o tempo de processamento do algoritmo e consequentemente, reduzindo a eficiência do ACS paralelo.

Diante dos gráficos apresentados, podemos considerar o algoritmo escalável por Gustafson, pois vem apresentando uma melhora na eficiência ao aumentar o tamanho do problema. E, também podemos concluir, que, para este algoritmo a execução com 250 formigas vai fornecer um melhor desempenho.

5 Conclusão

Este trabalho apresentou uma implementação paralela escalável da meta-heurística *Ant Colony System* aplicada ao Problema do Roteamento de Veículos Capacitado na arquitetura *multicore*. Na fundamentação teórica foi discutido as restrições do CVRP, assim como, foi realizada uma breve descrição sobre os algoritmos ACO. Também foi realizada uma breve descrição da arquitetura *multicore*. Posteriormente foram apresentados conceitos sobre métricas de escalabilidade paralela. No terceiro capítulo deste trabalho, foi apresentado o esquema geral de paralelização da meta-heurística ACS. Logo depois, foi detalhado cada etapa da implementação paralela proposta em OpenMP.

Através da análise dos gráficos de escalabilidade de Gustafson e Amdahl, é possível concluir que a implementação paralela do ACS é escalável tanto por Amdahl como por Gustafson. Também foi possível chegar a conclusão, de que, os gráficos de 250 formigas apresentam uma maior escalabilidade. Onde o problema com 1001 consumidores e com 2 *threads* alcançou super linearidade. Percebe-se que em todos os gráficos a eficiência diminui quando aumenta a quantidade de formigas, o que implica, que a melhor configuração para estes casos de teste é a escolha 250 formigas.

Apesar de o algoritmo ter alcançado bons resultados de escalabilidade, ainda é necessário testá-lo em um computador com mais núcleos, como também, para problemas maiores que 1001 consumidores. Outra análise que poderia ser feita no algoritmo seria o da distribuição de carga entre os processadores, utilizando a diretiva ***schedule*** do OpenMP. Poderia ser adicionado outro critério de parada do algoritmo com base na análise de uma possível processo estagnação melhorando assim o desempenho do algoritmo. Também poderia ser experimentado novos valores para os parâmetros afim de encontrar um bom equilíbrio entre diversificação e intensificação, assim como, a utilização de algoritmos de busca local, no intuito de melhorar a solução do problema.

Referências

ALBA, E. *Parallel metaheuristics: a new class of algorithms*. [S.l.]: John Wiley & Sons, 2005. Citado na página 13.

BAGCHI, S. *Emerging Research in Cloud Distributed Computing Systems*. [S.l.]: IGI Global, 2015. (Advances in Systems Analysis, Software Engineering, and High Performance Computing:). ISBN 9781466682146. Citado na página 29.

BLUM, C. Ant colony optimization: Introduction and recent trends. *Physics of Life reviews*, Elsevier, v. 2, n. 4, p. 353–373, 2005. Citado na página 21.

BLUM, C.; ROLI, A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, ACM, v. 35, n. 3, p. 268–308, 2003. Citado 2 vezes nas páginas 13 e 18.

BONABEAU, E.; DORIGO, M.; THERAULAZ, G. *Swarm intelligence: from natural to artificial systems*. [S.l.]: Oxford university press, 1999. Citado na página 13.

BRESHEARS, C. *The art of concurrency: A thread monkey's guide to writing parallel applications*. [S.l.]: "O'Reilly Media, Inc.", 2009. Citado na página 35.

BRESLIN, P.; KEANE, A. The capacitated arc routing problem: Lower bounds. University College Dublin, Management Information Systems Department, 1997. Citado na página 17.

CARDOSO, S. R. d. S. N. Optimização de rotas e da frota associada. 2009. Citado na página 18.

CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP: portable shared memory parallel programming*. [S.l.]: MIT press, 2008. Citado na página 30.

COUTINHO, D. A. M. Implementação paralela escalável e eficiente do algoritmo simplex padrão em arquitetura multicore. Universidade Federal do Rio Grande do Norte, 2014. Citado 2 vezes nas páginas 27 e 42.

CVRPLIB. *CVRPLIB - Capacitated Vehicle Routing Problem Library*. 2015. [Online; acessado em 2 de setembro de 2015]. Disponível em: <http://vrp.atd-lab.inf.puc-rio.br/index.php/en/>. Citado na página 46.

DANTZIG, G. B.; RAMSER, J. H. The truck dispatching problem. *Management science*, Informs, v. 6, n. 1, p. 80–91, 1959. Citado na página 16.

DORIGO, M. *Optimization, Learning and Natural Algorithms (in Italian)*. Tese (Doutorado) — Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1992. Citado 2 vezes nas páginas 13 e 24.

DORIGO, M.; CARO, G. D.; GAMBARDELLA, L. M. Ant algorithms for discrete optimization. *Artificial life*, MIT Press, v. 5, n. 2, p. 137–172, 1999. Citado na página 20.

DORIGO, M.; GAMBARDELLA, L. M. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, IEEE, v. 1, n. 1, p. 53–66, 1997. Citado na página 23.

DORIGO, M.; MANIEZZO, V.; COLORNI, A. The ant system: Optimization by a colony of cooperating agents. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS-PART B*, v. 26, n. 1, p. 29–41, 1996. Citado 2 vezes nas páginas 21 e 22.

DORIGO, M.; STüTZLE, T. *Ant Colony Optimization*. [S.l.]: BRADFORD BOOK, 2004. (A Bradford book). ISBN 9780262042192. Citado 2 vezes nas páginas 21 e 24.

FOULDS, L. R. *Combinatorial optimization for undergraduates*. [S.l.]: Springer Science & Business Media, 2012. Citado na página 18.

GAMBARDELLA, L. M. et al. Macs-vrptw: A multiple colony system for vehicle routing problems with time windows. In: *New Ideas in Optimization*. [S.l.]: McGraw-Hill, 1999. p. 63–76. Citado na página 17.

GEBALI, F. *Algorithms and Parallel Computing*. [S.l.]: Wiley, 2011. (Wiley Series on Parallel and Distributed Computing). ISBN 9780470934630. Citado na página 27.

GOLDBARG, M. C.; LUNA, H. P. *Otimização Combinatória e Programação Linear - Modelos e Algoritmos*. [S.l.]: Campus, Rio de Janeiro, Brazil, 2000. Citado na página 16.

GOSS, S. et al. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, Springer, v. 76, n. 12, p. 579–581, 1989. Citado na página 20.

GRAMA, A. et al. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. [S.l.]: Addison-Wesley Publishing, 2003. ISBN 0201648652. Citado na página 34.

GUSTAFSON, J. L. Reevaluating amdahl's law. *Communications of the ACM*, ACM, v. 31, n. 5, p. 532–533, 1988. Citado na página 38.

JAGTAP, M. P. Era of multi-core processors. *Power*, v. 2, p. 2, 2009. Citado na página 28.

LIN, C.; SNYDER, L. *Principles of parallel programming*. [S.l.]: Addison-Wesley Publishing Company, 2008. Citado 3 vezes nas páginas 26, 29 e 32.

LUCKA, M.; PIECKA, S. Parallel posix threads based ant colony optimization using asynchronous communication. In: *Proceedings of the 8th International Conference on Applied Mathematics*. [S.l.: s.n.], 2009. v. 2, p. 229–236. Citado na página 14.

MOHAMED, W.; ELBIALY ABOUELFARAG, A. A.; ALY, A. G. Performance analysis and tuning for parallelization of ant colony optimization by using openmp. In: *Computer Information Systems and Industrial Management*. [S.l.]: Springer, 2015. p. 73–85. Citado na página 14.

MOORE, G. E. Cramming more components onto integrated circuits. *Electronics, IEEE*, v. 38, p. 114–117, 1965. Citado na página 28.

PACHECO, P. *An introduction to parallel programming*. [S.l.]: Elsevier, 2011. Citado 5 vezes nas páginas 14, 25, 27, 33 e 39.

PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design: the hardware/software interface*. [S.l.]: Newnes, 2013. Citado na página 28.

PEDEMONTE, M. et al. A survey on parallel ant colony optimization. *Applied Soft Computing*, Elsevier, v. 11, n. 8, p. 5181–5197, 2011. Citado 2 vezes nas páginas 13 e 14.

ROSÁRIO, D. A. N. d. Escalabilidade paralela de um algoritmo de migração reversa no tempo (rtm) pré-empilhamento. Universidade Federal do Rio Grande do Norte, 2012. Citado na página 31.

STÜTZLE, T. Parallelization strategies for ant colony optimization. In: SPRINGER. *Parallel Problem Solving from Nature—PPSN V*. [S.l.], 1998. p. 722–731. Citado na página 14.

STVTZLE, T.; HOOOS, H. H. Max-min ant system. *Preprint submitted to Elsevier Science*, p. 13–17, 1999. Citado na página 25.

TANENBAUM, A. *Sistemas operacionais modernos*. [S.l.]: Prentice-Hall do Brasil, 2010. ISBN 9788576052371. Citado 3 vezes nas páginas 31, 32 e 34.

TOTH, P.; VIGO, D. An exact algorithm for the vehicle routing problem with backhauls. *Transportation science*, INFORMS, v. 31, n. 4, p. 372–385, 1997. Citado na página 17.