

Gabriel Marques de Melo

# **Implementação paralela do algoritmo TSP utilizando a meta-heurística ACO e MPI**

Brasil

Julho, 2019

Gabriel Marques de Melo

# **Implementação paralela do algoritmo TSP utilizando a meta-heurística ACO e MPI**

Implementação paralela do algoritmo TSP  
utilizando a meta-heurística ACO e MPI

Universidade Federal de Lavras – UFLA  
Departamento de Ciência da Computação  
Programação Paralela e Concorrente - GCC177

Brasil  
Julho, 2019

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>3</b>
	Introdução	3
<b>2</b>	<b>DESENVOLVIMENTO</b>	<b>5</b>
	Desenvolvimento	5
2.1	Adaptação da ACO para TSP	5
2.2	Paralelização do código	6
	Resultados	8
2.3	Resultados do algoritmo	8
2.4	Análise de desempenho	8
	Conclusão	13
	REFERÊNCIAS	14

# 1 Introdução

O problema do caixeiro viajante, ou **TSP** (Travelling Salesman Problem), representa uma grande classes de problemas conhecidos como problemas de otimização combinacional. Estes são difíceis de serem resolvidos usando métodos tradicionais e, caso sejam resolvíveis, suas computações tendem a ter elevadas complexidades de tempo.

A forma do TSP foi introduzida por Euler, em 1759. Também conhecido como Classical Travelling Salesman Problem (CTSP), o problema consiste em iniciar uma rota em uma cidade sendo então requerido que o caixeiro visite todas outras cidades apenas uma vez, de forma que a distância total percorrida seja minimizada. Ao aumentar o número de cidades, a complexidade do problema aumenta exponencialmente devido ao número de possíveis soluções crescer consideravelmente.

Este algoritmo - e suas extensões - possui aplicação direta em problemas de LOGÍSTICA, ROTEAMENTO e ESCALONAMENTO DE VEÍCULOS, MINIMIZAÇÃO DE CUSTOS e em muitos outros problemas reais.

Desde seu conhecimento, o TSP tem sido foco de estudo de muitos pesquisadores e inúmeras abordagens de solução diferentes foram propostas, principalmente, através das aplicações de heurísticas e meta-heurísticas. Uma dessas é a meta-heurística baseada em população **ACO** (ou *Ant Colony Optimization*), proposta por (DORIGO; CARO, 1999).

Essa meta-heurística surgiu da observação do comportamento das formigas na busca pelos alimentos. Inicialmente, cada formiga segue um caminho aleatório, porém, após algum tempo (gerações), elas tendiam a seguir um único caminho, considerado ótimo. Cada formiga utiliza de um mecanismo de comunicação indireta para indicar, para as demais, o quão bom foi seu caminho tomado, utilizando uma substância chamada **feromônio**.

Na Ant colony optimization cada indivíduo da população é um agente artificial que constrói, incrementalmente e estóticamente, uma solução para o problema considerado. A cada passo, o movimento destes define quais componentes de solução são adicionadas à solução em construção. Um modelo probabilístico é associado com o grafo e usado para ajustar as escolhas dos agentes. Esse modelo é atualizado por cada agente, fazendo com que a probabilidade de que futuros agentes tomem boas soluções aumente.

As adaptações realizadas para o problema do caixeiro viajante serão indicadas e justificadas no próximo capítulo.

Apesar da aplicação da ACO no TSP oferecer uma boa abordagem ao problema (principalmente quando comparada com algoritmos gulosos), este trabalho busca otimizar seu processamento através de sua PARALELIZAÇÃO, utilizando o framework MPI.

A estratégia de paralelização do algoritmo adotada será apresentada no capítulo seguinte.

## 2 Desenvolvimento

### 2.1 Adaptação da ACO para TSP

As adaptações da meta-heurística para atendimento ao TSP foram poucas, considerando que a ACO possui uma dinâmica semelhante ao do caixeiro viajante, onde as formigas (ou o caixeiro viajante) saem de seu ninho (cidade de partida), alcançam uma fonte de alimento (percorre todas as cidades) e retornam para seu ponto de origem.

Uma restrição adicionada foi a inclusão de um vetor PERMITIDOS que armazena os nós ainda não visitados por uma formiga e, por isso, são possíveis candidatos a serem visitados na sequência. Desta forma, a rota gerada configura um circuito, como no problema do caixeiro viajante.

A ACO possui dois importantes parâmetros de entrada:  $\alpha$  e  $\beta$ . O primeiro diz respeito à influência do feromônio na solução; e o último à influência da visibilidade dos nós (proximidade dos nós) na solução. De forma empírica, foram ajustados os valores destas para 1.0 e 10.0, respectivamente, de forma a valorar mais o menor caminho.

Um pseudo-código da implementação de uma solução ao TSP utilizando a meta-heurística *Ant Colony Optimization* (ACO) é mostrado abaixo.

---

**Algorithm 1** TSP com meta-heurística ACO

---

```

1: função TSP_ACO(num_geracoes, num_formigas,  $\alpha$ ,  $\beta$ ,  $\rho$ )
2:
3:   // Lê dataset TSPLIB
4:
5:   para  $t \leftarrow 1$  até num_geracoes faça
6:     para  $k \leftarrow 1$  até num_formigas faça
7:       para cont_movimento  $\leftarrow 1$  até num_cidades faça
8:         MOVIMENTA( $k$ ) ▷ baseado em  $P_{ij}^k$ 
9:       fim para
10:      CALCULA_L( $k$ )
11:    fim para
12:    ATUALIZA_FEROMONIO() ▷ baseado em  $\Delta T_{ij}$ 
13:  fim função

```

---

Sendo  $P_{ij}^k$  dado por:

$$P_{ij}^k = \begin{cases} \frac{[T_{ij}]^\alpha [N_{ij}]^\beta}{\sum_{s=1}^n \mathbb{I}_{\text{permitido}_k}[T_{is}]^\alpha [N_{is}]^\beta} & , \text{ se } j \in \text{permitido} \\ 0 & , \text{ caso contrário} \end{cases}$$

onde:

- $T_{ij}$  é a intensidade do feromônio entre os pontos  $i$  e  $j$ ;
- $\alpha$  é o parâmetro para regular a influência de  $T_{ij}$ ;
- $N_{ij}$  é a visibilidade do ponto  $j$  a partir do ponto  $i = \frac{1}{d_{ij}}$  ( $d_{ij}$  é a distância entre  $i$  e  $j$ );
- $\beta$  é o parâmetro para regular a influência de  $N_{ij}$ ;
- *permitido* é a lista de pontos ainda não visitados por uma formiga.

e  $T_{ij}$  sendo:

$$T_{ij} = \sum_{k=1}^n \Delta T_{ij}^k$$

onde:

$$\Delta T_{ij}^k = \begin{cases} \frac{Q}{L_k} & , \text{ se } k \text{ viaja na aresta } (i, j) \\ 0 & , \text{ caso contrário} \end{cases}$$

e:

- $Q$  é a intensidade do feromônio;
- $L_k$  é o tamanho do trajeto da formiga  $k$ .

Baseado nesse pseudo-código simplificado, foi implementado um código, em *Python*, para solução do TSP utilizando a meta-heurística OCA e dois datasets da TSPLIB (TSPLIB, 1995): *chn31.tsp* e *mu1979.tsp*. O código-fonte, disponível no GitHub<sup>1</sup>, é dividido em três arquivos principais: MAIN.PY, onde é feita a leitura do dataset, inicializada matriz de adjacência e os objetos da meta-heurística; ACO.PY, onde é implementada a meta-heurística de forma sequencial e PLOT.PY, onde é implementado um *plotter* para exibição da melhor solução encontrada.

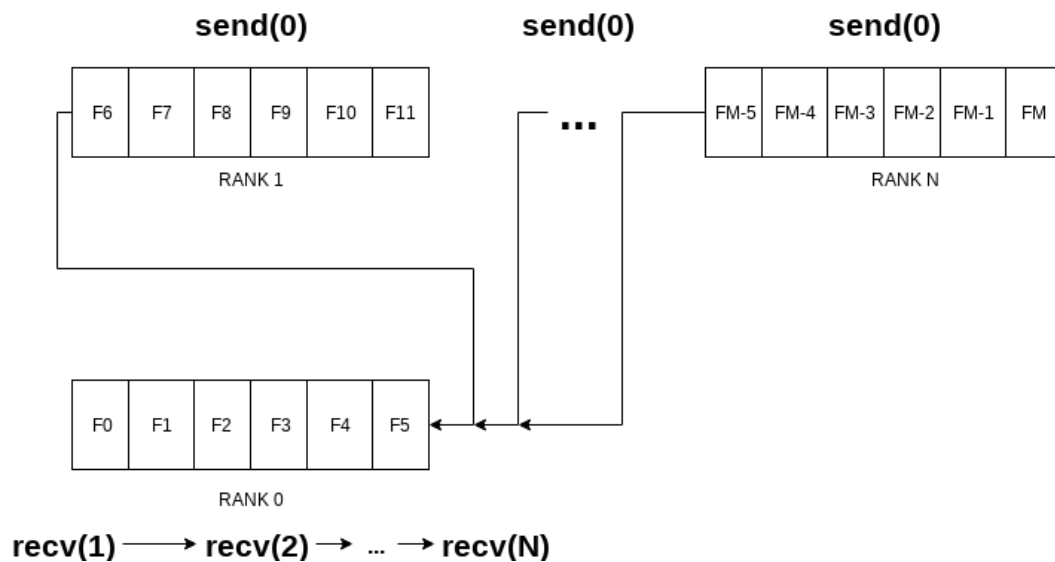
## 2.2 Paralelização do código

Na meta-heurística ACO, as formigas atuam como unidades de processamento. A cada geração, cada formiga percorre um circuito, saindo de sua posição inicial (aleatória), passando por todos os vértices e atualizando sua matriz própria de feromônios liberados. Ao final da geração, os valores locais dos feromônios de cada formiga são processados, atualizando os feromônios globais.

<sup>1</sup> <https://github.com/GabrielMMelo/parallel-aco-tsp>

Como pode se perceber, o processamento de uma formiga em uma geração é independente de todas as outras. Desta forma, a estratégia adotada para paralelização do algoritmo foi a distribuição do processamento das formigas em diferentes processadores, realizando, ao final da geração, a junção de seus resultados, através de mensagens, como ilustrado na Figura 2.2.

Figura 1 – Mensagens MPI



Fonte: Autor

Os resultados obtidos, bem como as métricas de avaliação de desempenho aplicadas serão mostrados no próximo capítulo.



# Resultados

Os testes de execução da meta-heurística foram realizados em uma máquina com as especificações dadas pela Tabela 1.

Característica	Valor
SO	Linux, Ubuntu 18.08, Kernel 4.9.16
Arquitetura	64 bits
Processador	Intel(R) Xeon(R) CPU Octacore E5620
Memória RAM	8 GB

Tabela 1 – Especificações da máquina do teste

Foi realizada a calibração dos parâmetros  $\alpha, \beta, \rho$  de entrada do algoritmo com uma instância pequena (*chn31.tsp*) e os melhores valores obtidos empiricamente foram, respectivamente, 1.0, 10.0 e 0.5. Esses valores foram utilizados para o cálculo de todas as demais execuções neste trabalho. As únicas alterações entre elas foram o tamanho da instância, o número de formigas e o número de gerações.

## 2.3 Resultados do algoritmo

Os melhores resultados obtidos para as instâncias CHN31.TSP e MU1979 são ilustrados nas figuras 2 e 3, respectivamente.

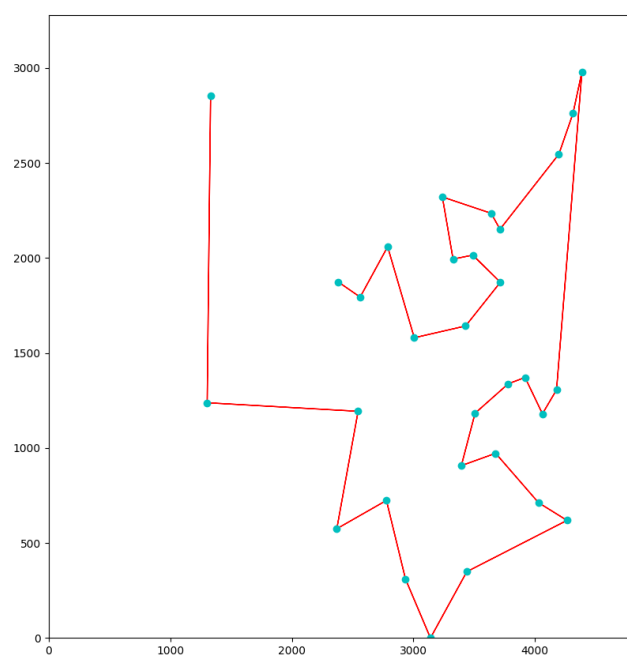
## 2.4 Análise de desempenho

Os parâmetros de avaliação adotados foram: **speedup**, **eficiência** e a métrica **Karp-Falt** (representada pela constante empírica  $\epsilon$ ). Os gráficos que ilustram os speedups das instâncias *chn31* e *mu1979* podem ser vistos nas Figuras 4 e 3, respectivamente.

Os demais parâmetros calculados<sup>2</sup> podem ser analisados, respectivamente, nas figuras 6 e 7.

<sup>2</sup> Os cálculos estão disponíveis em *utils/speedup.py* no repositório do GitHub

Figura 2 – chn31

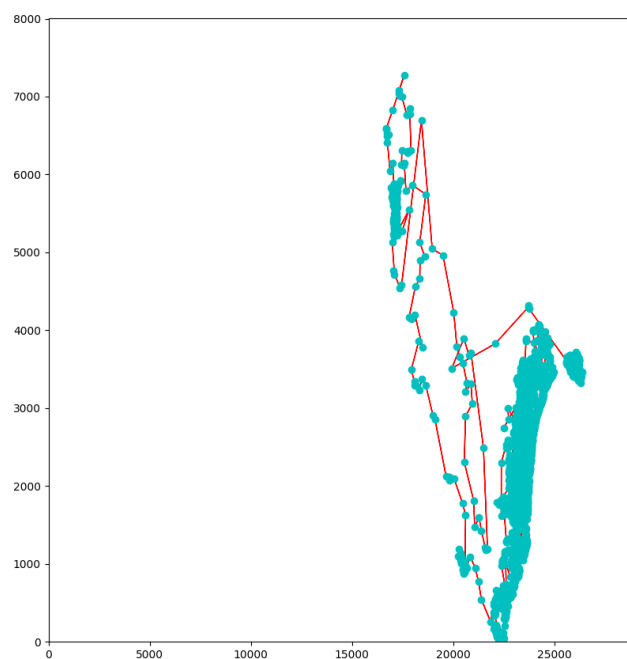


Número de formigas: 100

Número de gerações: 400

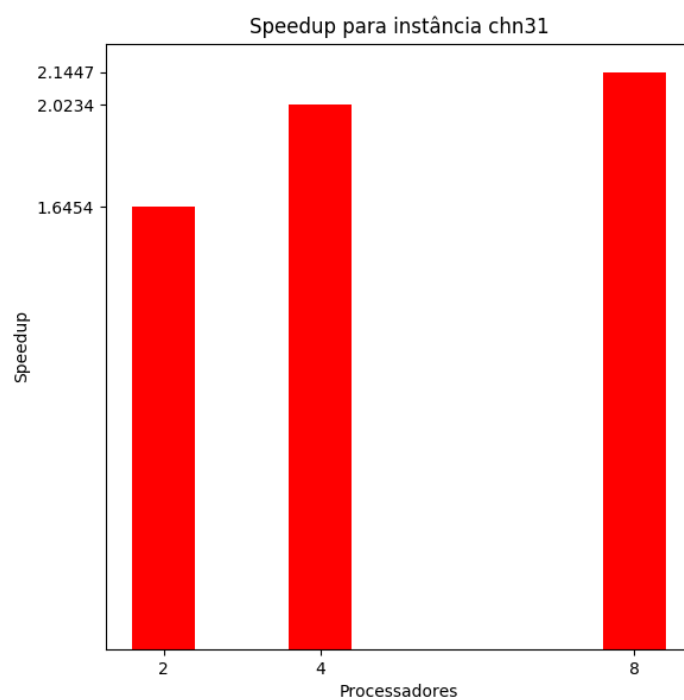
Custo total: 15601.919532918737

Figura 3 – mu1979



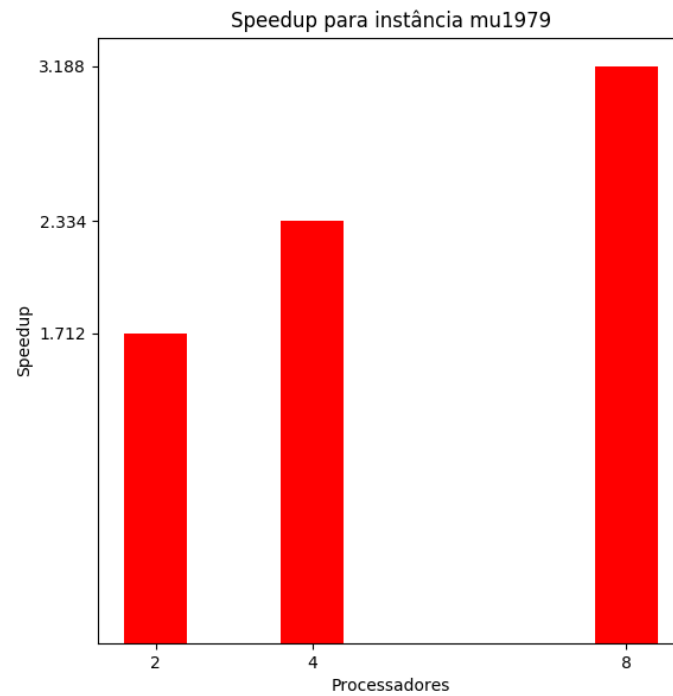
Número de formigas: 100  
Número de gerações: 10  
Custo total: 115883.64623438116

Figura 4 – Speedup chn31



Fonte: Autor

Figura 5 – Speedup mu1979



Fonte: Autor

Figura 6 – Medições chn31

```
### chn31 ###  
Speedup (8): 2.1446793085479965  
Eficiência (8): 0.26808491356849956  
e (8): 0.3900230589996498  
Speedup (4): 2.023367740590136  
Eficiência (4): 0.505841935147534  
e (4): 0.3256340439682604  
Speedup (2): 1.6453575277227377  
Eficiência (2): 0.8226787638613688  
e (2): 0.2155412828530383
```

Fonte: Autor

Figura 7 – Medições mu1979

```
### mu1979 ###  
Speedup (8): 3.1876962221551413  
Eficiência (8): 0.39846202776939266  
e (8): 0.21566420397448174  
Speedup (4): 2.3337348893629475  
Eficiência (4): 0.5834337223407369  
e (4): 0.23799691476408472  
Speedup (2): 1.7121822478185704  
Eficiência (2): 0.8560911239092852  
e (2): 0.16809995112852483
```

Fonte: Autor

## Conclusão

Dados os resultados obtidos, pode se avaliar que a paralelização da meta-heurística ACO aplicada ao TSP traz melhores resultados quando a instância é maior (mais cidades a visitar) e o número de formigas (entrada do algoritmo) é suficientemente pequeno, pois para grande número de formigas o overhead de paralelização foi maior.

## Referências

DORIGO, M.; CARO, G. D. Ant colony optimization: a new meta-heuristic. In: IEEE. *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*. [S.l.], 1999. v. 2, p. 1470–1477. Citado na página 3.

TSPLIB. *TSPLIB Documentation*. 1995. Disponível em: <<https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp95.pdf>>. Citado na página 6.