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

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 probabilistico 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.

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.

As adaptações realizadas para o problema do caixeiro viajante serão indicadas e justificadas no próximo capítulo, bem como a elucidação da estratégia de paralelização do

algoritmo adotada.

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: α e β. 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, α, β, ρ)
2:
3:      // Lê dataset TSPLIB
4:
5:      para t ← 1 até num_geracoes faça
6:          para k ← 0 até num_formigas − 1 faça
7:              para cont_movimento ← 1 até num_cidades faça
8:                  MOVIMENTA(k)                          ▷ baseado em P_ij^k
9:              fim para
                ATUALIZA_FEROMONIO_LOCAL(k)
10:         fim para
            ATUALIZA_FEROMONIO_GLOBAL()                   ▷ baseado em ΔT_ij
11:     fim para
12: fim função
```

Sendo P_{ij}^k dado por:

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

onde:

- T_{ij} é a intensidade do feromônio entre os pontos i e j;
- α 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);
- β 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:

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

e:

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

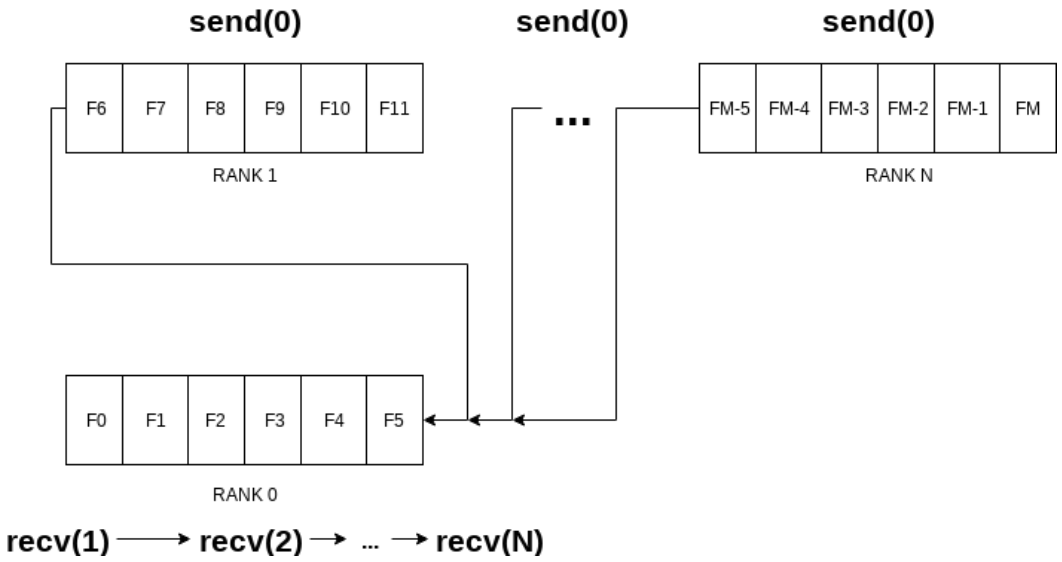
Baseado nesse pseudo-código simplificado, foi implementado um código, em *Python*, para solução do TSP utilizando a meta-heurística OCA e dois datasets da TSPLIB (TSPLIB, 1995): CHN31 e MU1979. O código-fonte, disponibilizado em um repositório público (MELO, 2019), é dividido em três arquivos principais: MAIN.PY, onde é feita a leitura do dataset, inicializada matriz de adjacencia 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.

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. O arquivo ACO_PARALLEL.PY, também disponível no repositório (MELO, 2019), possui o algoritmo paralelizado.

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 especificaçoes 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

Fora realizada a calibração dos parâmetros α, β *e* ρ de entrada do algoritmo com uma instância pequena (CHN31) 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 e MU1979 são ilustrados nas figuras 2 e 3, respectivamente.

2.4 Análise de desempenho

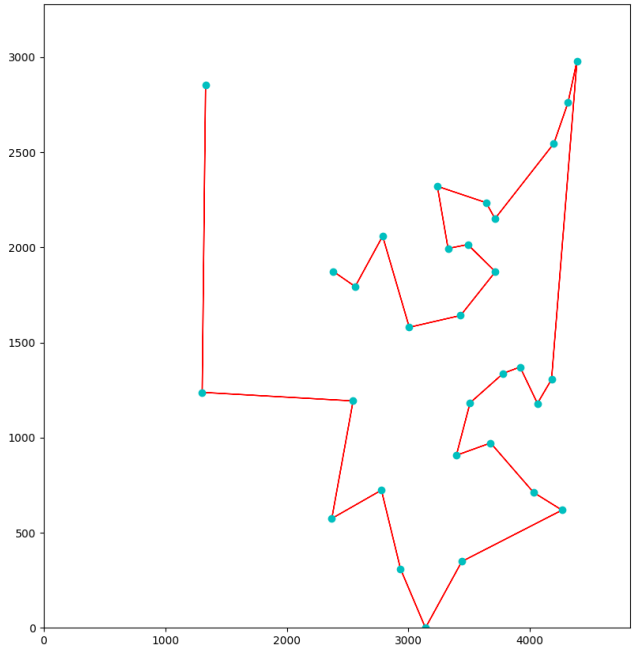
Os parâmetros de avaiação adotados foram: **speedup**, **eficiência** e a métrica **Karp-Falt** (representada pela constante empírica ϵ). O gráficos que ilustram os speedups das instâncias CHN31 e MU1979 podem ser vistos nas Figuras 4 e 3, respectivamente.

Os demais parâmetros calculados[1] podem ser analisados, respectivamente, nas figuras 6 e 7.

Observados os resultados obtidos até então, notou-se baixa eficiência do paralelismo implementado. Em um primeiro momento, creditava-se o baixo desempenho ao overhead intrínseco ao framework MPI (troca de mensagens bloqueantes), porém ao se realizar as medições desta defasagem, concluiu-se que este não era, por fim, o problema. A Tabela 2 mostra as medições de overhead paralelo obtidas.

[1] 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

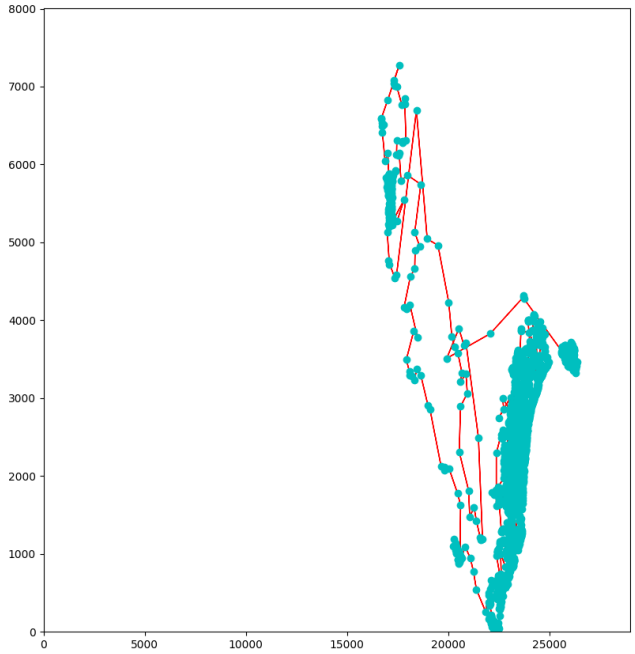
Tabela 2 – Overhead Paralelo

Processadores	Tempo (%)
2	0.023
4	0.019
8	0.017

Em busca de uma análise mais profunda do desempenho do algoritmo desenvolvido, foi realizado o *profiling* do código sequencial, visando identificar os gargalos da execução. Fora utilizado o módulo padrão cPROFILE para o levantamento estatístico. Os dados foram analisados através do módulo padrão PSTATS, no qual é possível manipular os resultados do profile para melhor visualização. A principal informação obtida é ilustrada na Figura 8. A quinta linha indica um gargalo da aplicação: a função SELECIONA_PROXIMO que não tira proveito da atual paralelização e consome, aproximadamente, 44% do tempo total de execução. Essa função é a responsável pelo cálculo da probabilidade da tomada de cada movimentação das formigas.

Diante disso, foram dispendidos esforços para melhoria no desempenho deste trecho de código através da paralelização **baseada em *threads***, uma vez que o MPI já estava implementado e realizava a concorrência **por processos**. Após o estudo de bibliotecas

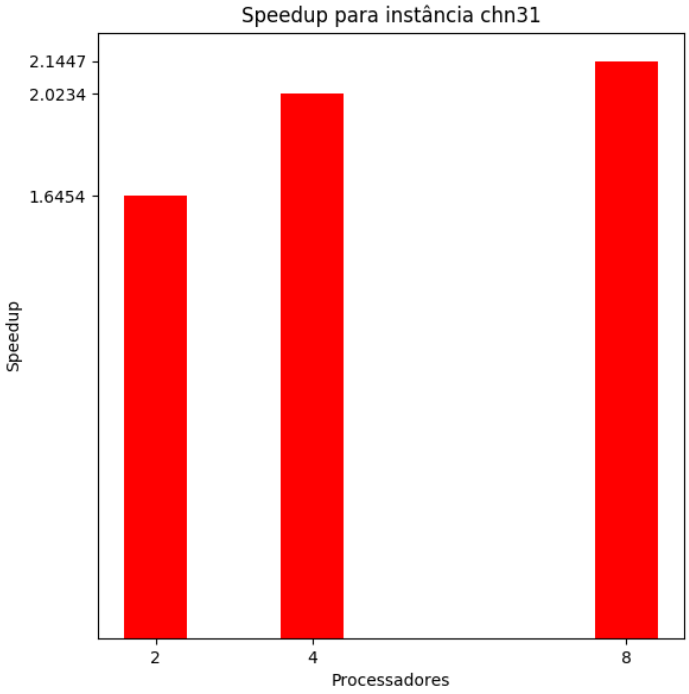
Figura 3 – mu1979



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

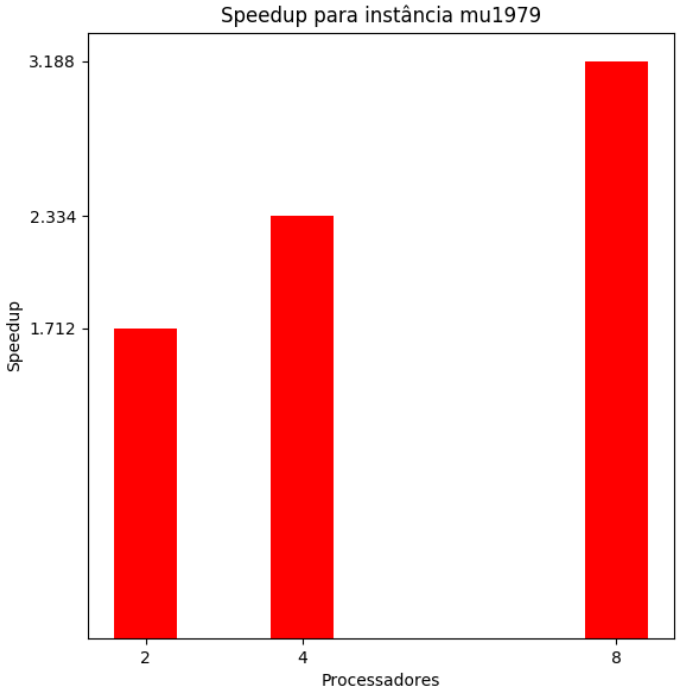
para manipulação de threads em Python, foram considerados os módulos THREADING e CONCURRENT.FUTURE. Contudo, nenhuma das duas abordagens foi satisfatória, culminando no aumento do custo de processamento neste ponto do código. A porcentagem de tempo gasto no gargalo alcançou os 76,4% (16.152/21.130).

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

Figura 8 – Pstats

```
>>> p.print_stats(10)
Thu Jul  4 22:04:50 2019    aco.profile

         1418515 function calls (1412595 primitive calls) in 8.715 seconds

   Ordered by: cumulative time
   List reduced from 2698 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    309/1    0.008    0.000    8.878    8.878 {built-in method builtins.exec}
        1    0.001    0.001    8.878    8.878 main.py:2(<module>)
        1    0.007    0.007    7.671    7.671 main.py:43(main)
        1    0.111    0.111    7.659    7.659 /home/melo/projects/ufla/ppc/tp/aco/aco.py:42(resolve)
    30000    3.823    0.000    5.625    0.000 /home/melo/projects/ufla/ppc/tp/aco/aco.py:88(_seleciona_proximo)
   930014    1.604    0.000    1.604    0.000 {method 'index' of 'list' objects}
    348/4    0.007    0.000    1.203    0.301 <frozen importlib._bootstrap>:978(_find_and_load)
    348/4    0.005    0.000    1.203    0.301 <frozen importlib._bootstrap>:948(_find_and_load_unlocked)
    332/4    0.006    0.000    1.201    0.300 <frozen importlib._bootstrap>:663(_load_unlocked)
    284/3    0.004    0.000    1.200    0.400 <frozen importlib._bootstrap_external>:722(exec_module)
```

Fonte: Autor

Figura 9 – Pstats

```
(aco-env) melo@debian:~/projects/ufla/ppc/tp/aco$ time mpirun -n 2 python main.py --parallel
Parallel!
Parallel!
16.152668714523315
custo total: 16271.665325966618, caminho: [14, 12, 11, 13, 10, 22, 15, 4, 5, 6, 1, 3, 7, 8, 9,

real    0m21.130s
user    0m32.488s
sys     0m6.772s
```

Fonte: Autor

Conclusão

Dados os resultados obtidos, pôde se avaliar que a paralelização da meta-heurística ACO aplicada ao TSP trouxe melhoria no tempo de processamento total, porém a eficiência da concorrência não foi satisfatória, principalmente quando se incrementava o número de agentes (formigas). Foi concluído, então, que este comportamento surgiu como consequência de um erro na identificação inicial do gargalo do algoritmo sequencial, devido à desatenção e a não utilização de uma ferramenta de profiling auxiliar ao programador.

Foram encontradas dificuldades na implementação **efetiva** de manipulação de threads, uma vez que a linguagem Python não oferece uma ferramenta com a robustez e alto nível para tal, como o OpenMP.

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.

MELO, G. M. de. *parallel-aco-tsp*. 2019. Disponível em: <https://github.com/GabrielMMelo/parallel-aco-tsp>. Citado 2 vezes nas páginas 6 e 7.

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