Análise de Desempenho da Paralelização do Problema de Caixeiro Viajante

Gabriel Freytag[1], Guilherme Arruda[1], Rogério S. M. Martins[1], Edson L. Padoin[1,2]

[1]Universidade Reg. do Noroeste do Estado do Rio G. do Sul (UNIJUI) - Ijuí - RS - Brasil

[2]Universidade Federal do Rio Grande do Sul (UFRGS) - Porto Alegre - RS - Brasil

{gabriel.freytag, guilherme.arruda, rogerio.martins, padoin}@unijui.edu.br

Resumo. *Um dos problemas de otimização mais conhecidos é o Problema do Caixeiro Viajante (PCV). Para solução deste problema altos tempos computacionais são necessários. Uma técnica utilizada para melhorar a eficiência e diminuir o tempo de execução é a paralelização. Este trabalho apresenta uma comparação de desempenho da paralelização do PCV com as ferramentas pthread, OpenMP, MPI e CUDA. Obtendo pthread e MPI um tempo 2 vezes menor.*

1. Introdução

O Problema do Caixeiro Viajante é um clássico problema de otimização combinatória que pode ser aplicado a inúmeros problemas práticos. Problemas de otimização combinatória são problemas onde existe um conjunto discreto e finito de dados sobre o qual é preciso selecionar o melhor subconjunto que se enquadre em determinadas restrições. A maioria destes problemas são intratáveis, ou seja, requerem grandes tempos computacionais para serem resolvidos e não possuem um algoritmo eficiente conhecido.

Além disso, problemas onde o tempo aumenta exponencialmente conforme o tamanho das entradas, pertencem a classe de problemas NP, cuja solução pode ser encontrada em tempo polinomial por um algoritmo não determinístico. Quando todos os problemas de busca podem ser reduzidos para um destes problemas, este pertence a classe NP-completo. O PCV é um problema NP-Completo.

Este trabalho motivou-se pelo estudo de ferramentas de paralelização como pthread, OpenMP, MPI e CUDA, que podem diminuir o tempo de resolução do PCV.

1.1. Problema

O Problema do Caixeiro Viajante consiste num vendedor que necessita visitar cada uma das *n* cidades uma única vez, iniciando em uma cidade e retornando ao local de partida. A questão é encontrar a rota que minimize a distância total a ser percorrida pelo vendedor. [Dantzig et al. 1954, Lin 1965].

Matematicamente, pode ser determinado de duas formas, segundo Lin [Lin 1965]:

1. Dada uma matriz de custo *D* = (d_{ij}), onde $d_{i,j}$ = custo de ir da cidade *i* para a cidade *j*, encontrar uma permutação *P* = $(i_1, i_2, i_3, \cdots, i_n)$ dos inteiros de 1 até *n* que minimize a quantidade $d_{i_1 i_2} + d_{i_2 i_3} + \cdots + d_{i_n i_1}$.
2. Dada uma matriz de custo *D* = (d_{ij}), determinar x_{ij} que minimize a quantidade *Q* = $\sum_{i,j} d_{ij} x_{ij}$ sujeito a: (a) x_{ii} = 0; (b) x_{ij} = 0, 1; (c) $\sum_i x_{ij} = \sum_j x_{ij} = 1$; (d) para um subconjunto *S* = $\{i_1, i_2, \cdots, i_r\}$ dos inteiros de 1 até *n*, $x_{i_1 i_2} + x_{i_2 i_3} + \cdots + x_{i_{r-1} i_r} + x_{i_r i_1} \begin{cases} < r \; para \; r < n \\ \leq r \; para \; r = n. \end{cases}$

Neste trabalho, utilizamos a primeira forma para resolver o problema.

2. Trabalhos Relacionados

Gressler [Gressler and Cera 2012] paraleliza a implementação de um algoritmo genético sequencial para a resolução do Problema de Roteamento de Veículos, utilizando a API OpenMP. Fujimoto [Fujimoto and Tsutsui 2011] apresenta um novo método para resolver o PCV com GPU baseada na arquitetura CUDA, paralelizando uma metaheurística Algoritmo Genético. Xu [Xu et al. 2013] apresenta um algoritmo de colônia de abelhas artificial paralela utilizando MPI. Milanovi [Milanovic et al.] paraleliza uma otimização da resolução do PCV com Colônia de Formigas com a ferramenta OpenMP.

3. Contribuições do Artigo

Tradicionalmente, algoritmos que implementam o PCV são desenvolvidos seguindo o método de busca do vizinho mais próximo. Devido a inúmeros trabalhos com esta abordagem, nosso trabalho propõe uma metodologia diferenciada, baseada na média aritmética do custo das ligações entre as cidades não visitadas. A partir da cidade em que o vendedor se encontra, é calculada a média aritmética do custo das ligações com as cidades que ainda não foram visitadas. Em seguida, as cidades são separadas em dois vetores, um contendo as cidades com custo de ligação maior que a média aritmética e outro com custo de ligação menor. Este cálculo ocorre a cada nova cidade visitada.

3.1. Descrição do Algoritmo Sequencial

Ao iniciar o algoritmo sequencial, é criada uma solução aleatória. Esta solução fornece um custo inicial a ser otimizado pelo algoritmo. Da mesma forma que a heurística construtiva, o algoritmo sequencial primeiramente preenche o vetor de cidades não visitadas e define aleatoriamente uma destas cidades como a origem. A partir desta cidade, é calculada a média aritmética do custo das ligações entre as cidades e então as cidades são separadas em dois vetores, um com cidades com custo menor e outro com custo maior que a média. Esta separação ocorre novamente a cada iteração, pois cada cidade possui diferentes ligações com outras, estas com diferentes custos. Com as cidades separadas, a cada iteração o algoritmo seleciona aleatoriamente uma cidade com custo de ligação menor que a média aritmética para ser visitada. Caso o vetor com cidades com custo menor que a média esteja vazio, o algoritmo seleciona a cidade com o menor custo de ligação contida no vetor de cidades com custo maior que a média aritmética.

Todo este ciclo se repete até que todas as cidades sejam visitadas e o vendedor volte à cidade origem. São analisadas tantas rotas quanto o limite de iterações definidas, sendo o número de iterações reiniciado a cada nova melhor rota encontrada.

3.2. Descrição dos Algoritmos Paralelos

No algoritmo paralelo utilizando pthread, cada *thread* executa os mesmo passos que o algoritmo sequencial. Entretanto, cada *thread* possui um determinado número de cidades para traçar uma rota parcial. Como as *threads* são novas instâncias e não possuem as informações que a instância principal possui, é necessário passá-las por parâmetro. Estas informações são passadas por parâmetro através de *structs*. Desta forma, cada *thread* possui as informações necessárias para sua execução. Quando todas as *threads* concluíram sua execução, as rotas parciais de cada uma são unidas e os custos destas são somados.

Utilizando OpenMP, o algoritmo paralelo possui a mesma estrutura do algoritmo sequencial, porém, apenas os laços *for* são executados em paralelo.

Assim como no algoritmo com Pthread, no algoritmo paralelo utilizando MPI as informações precisam ser passadas para as *threads*. As informações são enviadas por mensagem através de uma *struct*. Essa *struct* é uma estrutura com tipo de dado conhecido pela MPI e que armazena as informações necessárias para as *threads*. A *thread* mestre envia as informações para as escravas e elas retornam seus resultados, que são somados.

Por fim, no algoritmo paralelo utilizando CUDA, primeiramente é alocada memória na CPU e na GPU. Os dados necessários para a execução das *threads* são copiados para a GPU e a execução é iniciada. Ao final, o resultado é copiado para a CPU.

4. Avaliação de Desempenho

Nesta seção descreve-se o ambiente de teste e os resultados dos algoritmos da Seção 3.

4.1. Ambiente de Teste e Plataformas de Experimentação

Os algoritmos foram desenvolvidos na linguagem de programação C++. Os testes foram realizados em um equipamento com processador Intel Core i7 Ivy Bridge 3630QM de 8 núcleos, 8 GB de memória RAM e uma placa gráfica Nvidia GeForce GT 640M de 1 GB de memória. Possuía o sistema operacional Linux Elementary OS kernel 3.13.5 instalado.

Para os testes, definiu-se o número de iterações igual a 100.000 e o número de *threads* em que seriam executados os testes igual a 2, 4, 8 e 16. Os testes foram realizados com cada algoritmo paralelo executando 10 testes nas quatro configurações de *threads*. A entrada em todos os testes foi de tamanho único, 58 cidades brasileiras.

4.2. Apresentação e Análise dos Resultados

A paralelização do algoritmo com pthread obteve resultados satisfatórios, conforme Figura 1. Com a divisão do problema em partes menores e executadas em paralelo, o tempo tomado pela ferramenta reduziu a medida que o número de *threads* aumentou, voltando a aumentar o tempo com 16 *threads*. O melhor desempenho se deu com 8 *threads*.

Na paralelização utilizando OpenMP os resultados não foram satisfatórios, como mostra a Figura 1. O algoritmo possui algumas variáveis compartilhadas, acarretando em concorrência entre as *threads*. Estas concorrências no acesso às variáveis precisam ser controladas, o que torna a resolução extremamente demorada. Por ter sido paralelizado apenas os laços *for*, devido a dificuldade enfrentada na paralelização do problema com esta ferramenta, e não estando otimizado, obteve o pior desempenho entre os algoritmos.

Com a paralelização do algoritmo utilizando MPI, o desempenho foi próximo ao obtido com pthread, conforme Figura 1. Pelo fato da comunicação entre as *threads* ser através de mensagens, é possível enviar e receber informações facilmente. Desta forma, com o problema dividido em partes menores e a comunicação entre as *threads* ter um custo mínimo, o tempo tomado pelo algoritmo é reduzido.

Por fim, utilizando CUDA, os resultados obtidos com 2 *threads* foram próximos aos obtidos com pthread e MPI, porém, não diminuíram com o aumento do número de *threads*, como mostra a Figura 1. A cópia das informações contidas na memória da CPU para a memória da GPU resulta num custo adicional. Mas isto por si só não é a razão do

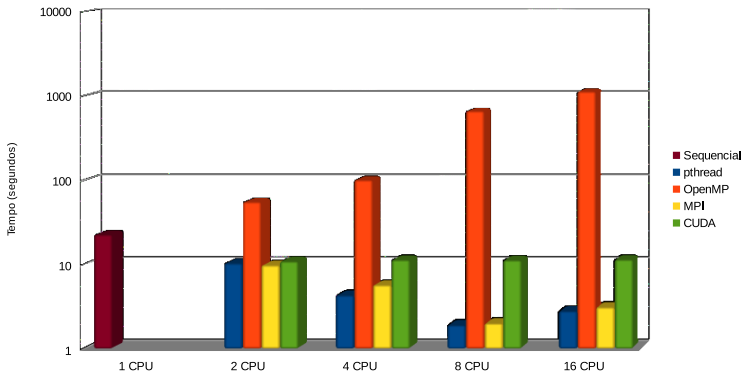

Figura 1. Tempo de execução de cada algoritmo

desempenho não ter sido melhor. O principal motivo é o tamanho pequeno do problema comparado com a capacidade da GPU, assim como o número reduzido de *threads*.

5. Conclusão e Trabalhos Futuros

Através da comparação do tempo tomado por cada ferramenta para resolver o problema, observa-se que os algoritmos com melhor desempenho foram pthread e MPI. Entretanto, devido ao tamanho reduzido do problema e poucas *threads*, comparado com a capacidade da CUDA, além da falta de otimização, seu desempenho foi comprometido. Com OpenMP, pela falta de otimização e dificuldades enfrentadas, teve os piores resultados.

Um trabalho futuro seria a otimização do algoritmo paralelizado com OpenMP. Outro trabalho futuro poderia ser realizar testes com entradas de diferentes tamanhos e números maiores de *threads*, onde possivelmente CUDA teria um desempenho melhor.

Referências

Dantzig, G., Fulkerson, R., and Johnson, S. (1954). Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410.

Fujimoto, N. and Tsutsui, S. (2011). A highly-parallel tsp solver for a gpu computing platform. In *Numerical Methods and Applications*, pages 264–271. Springer.

Gressler, H. D. O. and Cera, M. C. (2012). Paralelizando um algoritmo genético aplicado ao problema do roteamento de veículos. *Anais do Salão Internacional de Ensino, Pesquisa e Extensão*, 4(2).

Lin, S. (1965). Computer solutions of the traveling salesman problem. *Bell System Technical Journal, The*, 44(10):2245–2269.

Milanovic, A., Duranovic, M., and Nosovic, N. Performance analysis of the traveling salesman problem optimization using ant colony algorithm and openmp. In *MIPRO, 2013 36th International Convention on*.

Xu, K., Jiang, M. Y., and Yuan, D. F. (2013). Parallel artificial bee colony algorithm for the traveling salesman problem. *Advanced Materials Research*, 756:3254–3259.