Problema do Caminho Mínimo: Uma Proposta de Paralelização do Algortimo Dijkstra

Marisangila Alves[1]

[1]Programa de Pós Graduação de Computação Aplicada - PPGCA
Universidade do Estado de Santa Catarina - UDESC - Joinville - SC - Brasil

marisangila.alves@gmail.com

Resumo. *Problemas de computação com tempo de execução e complexidade não linear podem otimizados. Dentre esses, o problema do caminho mínimo possui aplicações no mundo real, tal tomo planejamento de rotas de veículos ou roteamento de pacotes. O algoritmo Dijkstra é uma solução amplamente utilizada em redes de computadores. O presente trabalho apresenta uma análise de implementação de paralelismo no algoritmo Dijkstra a partir de uma metodologia que análiza particionamento, comunicação, aglomeração e mapeamento, que pode ser chamada PCAM. Além disso, a implementação de paralelismo do problema com uso de OpenMP e MPI*

1. Introdução

Determinados problemas clássicos da computação de complexidade não linear e não triviais implementados através de lógica sequencial pode ser enviáveis, ou seja, possuem tempo de execução maior do que o aceitável, dado o tamanho da entrada e a escalabilidade do problema. Contudo, é possível reduzir o tempo de execução desse problemas de forma considerável apropriando-se de ferramentas da computação paralela [Cáceres 1192]. Dentre esse problemas, na área da matemática conhecida como teoria de grafos conhecidos, tais problemas podem ser: Grafo Euleriano, Problema da Pontes de Königsberg, Grafo Hamiltoniano, Teorema das Cinco Cores, Problema do Caixeiro Viajante, Problema do Fluxo Máximo e ademais problemas clássicos [da Costa 2011].

Diante de tais problemas, o problema do caminho mínimo é um dos mais simples problemas de fluxo de rede. De forma geral para esse problema o objetivo é encontrar o caminho de menor custo, dado um vértice de origem e um vértice de destino [Ahuja et al. 1995]. O problema do caminho mínimo é considerado importante pelas possibilidades de aplicações prática. Essas aplicações podem ser: planejamento de rota e tempo de viagens, roteamento e programação de veículos, planejamento de capacidade, expansão de redes de transporte e comunicação, programação de caminhos críticos, problema do caixeiro viajante, problema da mochila e problemas de equilíbrio de tráfego [Glover et al. 1985].

Para resolver o problema do caminho mínimo ao longo dos anos surgiram algumas soluções, sendo elas: Algoritmo de Bellman-Ford, Algortimo A, Algoritmo de Floyd-Warshall , Algoritmo de Johnson, Algoritmo Viterbi e Algoritmo de Dijkstra. o algoritmo proposto por [Dijkstra 1959] foi primeiro mais eficiente que objetiva resolver o problema em grafos com pesos não negativos. Por sua vez, Dijkstra pode ser utilizado em algoritmos de roteamento de redes de computadores ou e.g. protocolo *Open Shortest Path First* (OSPF) [Kurose 2013].

Diante o exposto o algorítimo Dijktra sequencial como solução do problema de caminho mínimo é selecionado para uma análise de implementação usando paralelismo.

Assim, o artigo apresenta a definição do problema na Seção 2. A metodologia para implementação do paralelismo é descrito na Seção 3. A implementação, realização e discussão dos testes são descritos nas Seções 4, 5 e 6. A Seção 7 apresenta as conclusões.

2. Definição do Problema

O pseudocódigo representa a implementação sequencial do algoritmo de Dijkstra. Durante a primeira etapa os vetores que armazenam as distância de cada vértice para seus adjacentes é inicializado, assim como o vetor que representa os vértices de origem. Sendo assim, para todo vértice v no grafo $V[G]$ sendo v o número de vértices, o vetor de $d[v]$ é recebe ∞ em todas as posições e o vetor $\Pi[s]$ recebe -1 em todas as posições, sendo s o vértice de origem ao vértice v. O vetor $d[s]$ recebe 0, pelo motivo de que refere-se a distância de s para s, ou seja, ele mesmo.

```
para todo v ∈ V[G]
    d[v] ← ∞
    π[v] ← -1
d[s]
Q ← V[G]
enquanto Q ≠ ø
      u ← extrair-mín(Q)
      para cada v adjacente a u
         se d[v] > d[u] + peso(u, v)
           então d[v] ← d[u] + peso(u, v)
                π[v] ← u
```

A segunda etapa é atribuir a Q o conjunto de vértices $V[G]$, no qual esses vértices não possuem menor caminho determinado em $d[v]$. Por fim a etapa de relaxamento, enquanto Q não for vazio são executadas as seguintes instruções: u recebe o vértice com a menor distância em Q. Para cada vértice v adjacente ao vértice u. Se $d[v]$ for maior que a soma de $d[u]$ e pesos do vértice u até v. O vértice v $\Pi[v]$ recebe u.

Entretanto, o algoritmo sequencial utilizado nesse trabalho, possui adaptações, que podem ser conferidas no código fonte[1]. Para representar o conjunto Q será utilizada uma estrutura de dados em forma de uma fila de prioridade, que ordena as distâncias de maneira que o menor valor esteja no início da fila de prioridades. Além disso, utiliza uma estrutura de dados em forma de lista, similar a um dicionário, para armazenar os vértices adjacentes e seu custo.

[1] https://github.com/marischatten/Dijkstra

3. Metodologia de Projeto de Paralelismo

A metologia empregada na proposta de paralelização chamada PCAM divide-se nas etapas: Particionamento, Comunicação, Aglomeração e Mapeamento. Nas subseções a metodologia será aplicada ao código descrito na Seção 2.

3.1. Particionamento

O particionamento consiste em fragmentar as operações que compõem a resolução de um problema. A fragmentação pode ser de decomposição de domínios ou decomposição funcional, sendo particionamento de dados e particionamento de operações, respectivamente [Schnorr and Nesi 2019]. Dentre a etapas descritas pelo pseudoalgoritmo é possível particionar o código de acordo com suas etapas, ou seja, a etapa de inicialização das listas de distâncias calculadas e vértices de origem e, um segundo bloco responsável pela etapa de relaxamento. Na etapa de inicialização dos vetores, é possível aplicar particionamento de dados, ou seja, decomposição por domínios, nessa etapa o objetivo é apenas executar uma operação de atribuição de valor a cada posição dos vetores. Sendo assim o particionamento é realizado por células.

Por sua vez, na etapa de relaxamento é possível particionar o problema como decomposição de domínio. Dessa forma, dado um vértice que possua a menor distância de acordo a fila de prioridades existe uma lista de adjacências para o vértice com menor distância. O tamanho da lista relaciona-se diretamente com o tamanho do problema. Portanto, a execução do laço que verifica o custo de cada um dos vértices adjacentes pode ser executado em um tempo de execução não linear. A Figura 6 exemplifica o particionamento, onde cada célula do vetor é atribuído para uma tarefa.

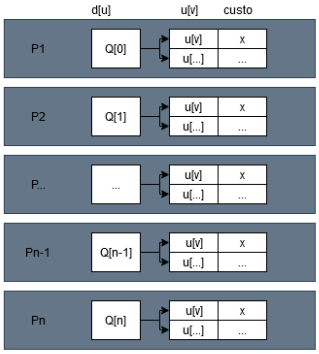

Figura 1. Particionamento do Relaxamento.

3.2. Comunicação

Comunicação é a definição de como tarefas se comunicam entre as partes fragmentas do problema, compartilhando ou não, valores e variáveis [Schnorr and Nesi 2019]. A etapa de inicialização é trivialmente paralelizável, ou seja, não necessita de comunicação entre as tarefas. Cada núcleo recebe um ponteiro e atribuir valor ao endereço de memória apontado. Entretanto, na etapa de relaxamento, é necessário comunicação entre as tarefas. os vetores d, Π e a fila de prioridade Q devem ser atualizadas a cada operação, sendo assim, é necessário comunicação global a partir de *broadcast*, de uma tarefa para

muitas. A comunicação é estruturada, portando, cada tarefa se comunica com as mesma tarefas independe da execução. Além disso, a comunicação pode ser classificada como estática, a representação é dada por uma estrutura fixa. Por fim, a comunicação deve ser síncrona, uma vez que, uma tarefa é concluída, ou seja, executa operações sobre os vetores, essa tarefa precisa enviar uma atualização dos novos valores para todas as outras tarefas. Enfatiza-se que esse comportamento possivelmente pode impactar negativamente no tempo de execução, dado que todas as tarefas param seu trabalho é aguardam a comunicação ser concluída.

3.3. Aglomeração

Aglomeração é a avaliação de requisitos de forma a unificar as etapas de particionamento e comunicação [Schnorr and Nesi 2019]. Durante a inicialização não existe comunicação entre as tarefas, portanto, não são necessárias aglomerações, por outro lado, durante o relaxamento, tendo em vista a escalabilidade do problema é necessário aglomerar tarefas. É possível aglomerar as tarefas considerando o número de núcleos existentes, dividindo número total de tarefas pelo número total de núcleos. Isso proporciona redução evidente ao número de comunicações entre as tarefas, entretanto, depende expressamente do *hardware* e tamanho do problema. Todas as tarefas decompostas podem ser executadas em concorrência, contudo, ao concluir a execução cada tarefa deve atualizar valores que devem ser replicados para todas as demais tarefas, essa situação aumenta o números de comunicações e pode prejudicar consideravelmente o desempenho.

3.4. Mapeamento

Mapeamento é atribuição das tarefas às unidades de processamento, com intensão de minimizar a comunicação entre tarefas e maximizar o uso de recursos [Schnorr and Nesi 2019]. Para distribuir as tarefas entre partições. São elencadas duas possibilidades, para possíveis critérios de escolha, a primeira considera heurísticas descentralizadas para escalonamento de tarefas, onde os núcleos podem solicitar mais tarefas quando ociosos ou então, o balanceamento de carga como mapeamento cíclico que respeita os blocos criados na etapa de aglomeração.

4. Implementação

A implementação de paralelismo seguiu uma abordagem diferente da proposta definida com a metodologia PCAM. Em vez de atribuir o paralelismo ao algoritmo Dijkstra, foi realizado paralelismo direcionado aos dados, ou seja, dado $(N^2) - N$, calcular o menor caminho entre os vértices de todos para todos, desconsiderando cálculo de um vértice para ele mesmo. A alternativa deve-se a inviabilidade de paralelizar o algoritmo Dijkstra de forma eficiente, no qual realiza tarefas consecutivamente que não podem ser realizadas de forma concorrente. Sendo assim, o objetivo é preencher uma matriz com o custo mínimo, que representa um grafo completo.

O particionamento na implementação de paralelismo com auxilio do OpenMP[2] considerou a divisão de tarefas através de linhas, ou seja, cada linha da matriz é calculado por uma *thread*. Entretanto, para a implementação através de *Message Passing Interface* (MPI), devido a comunicação entre processos, foi necessário adotar outra abordagem de

[2]https://www.openmp.org/

particionamento. Dessa forma, o particionamento considerou a divisão de tarefas por células.

Para que a comunicação fosse possível, os valores da matriz foram transformados em uma matriz, sendo que cada linha da matriz se refere a um cálculo de caminho e, as colunas armazenam as informações, no qual representam qual origem e destino a ser calculado e o resultado. Por fim, foi necessário converter objeto complexo que compunha a lista de adjacência em um vetor. Contudo, foi fundamental considerar o agrupamento com objetivo de reduzir o número de troca de mensagens entre os processos. Portanto, agrupamento de tarefas foi realizado de acordo com o total de tarefas a serem feitas divididas pelo número de processadores. Finalmente, o mapeamento foi considerado de forma dinâmica através de arquivo de configuração e de acordo com os testes executados. A Seção 6 aprofunda os resultados obtidos.

5. Experimentos

A realização de experimento na implementação com auxilio do OpenMP foi realizada com computador C, com 10 execuções por parâmetro. A execução dos testes da paralelização que utiliza MPI foi realizada em todos os computadores, com 5 execuções por parâmetro, através da biblioteca OpenMPI[3]. Ambas abordagens foram codificadas em linguagem de programação C++.

O comparação de ambas versões paralelizadas são comparadas com o mesmo código sequencial. Entretanto, para implementação com MPI foi necessário alterações consideráveis no código em relação ao código sequencial. A Tabela detalha a configuração do *hardware* em que os experimentos foram executados. A entrada de dados, ou seja, a lista de adjacência foi realizada a partir da leitura um arquivo de texto, com a ressalva de considerar o tempo de leitura do arquivo durante a execução total e, além disso, destaca-se a heterogeneidade do *hardware*. A aceleração foi obtida através do tempo de execução serial em segundos dividida pelo tempo de execução paralela em segundos.

		Núcleos	HiperThreading	Frequencia	Cache	Modelo
A/B	ens1/ens2	4	Não	2.8 GHz	512 KB	AMD Phenom(tm) II X4 B93 Processor
C	ens4	4	Sim	3.4 GHz	8 MB	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
D	ens5	4	Sim	3.5GHz	8 MB	Intel(R) Xeon(R) CPU E3-1230 v6 @ 3.50GHz

Figura 2. Hardware utilizado para testes.

6. Discussão

Essa seção apresenta a análise e discussão dos resultados oriundos dos testes realizados.

6.1. OpenMP - Escalabilidade de entrada de dados

A Figura 3 apresenta a comparação de tempo de execução em segundos entre execução serial e paralela em relação ao tamanho da entrada. Foram realizadas 10 execuções por número de vértices, e extraída média aritmética. É possível observar que o ganho de desempenho para poucos dados não difere-se entre o código serial e o código paralelo. Entretanto, para entrada de dados maiores, o código paralelo destaque-se em relação ao

[3]https://www.open-mpi.org/

código serial. Sobretudo, o resultado destaca e justifica a necessidade de paralelização para esse problema, para que seja possível proporcionar escalabilidade em versões maiores deste problema.

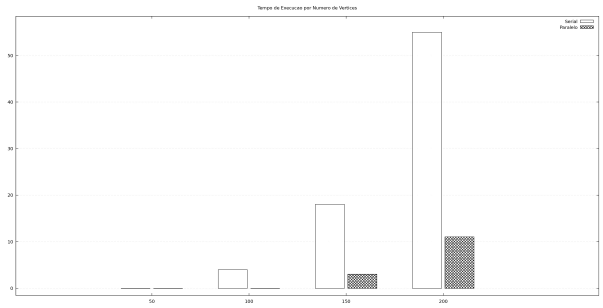

Figura 3. Escalabilidade de entrada de dados.

6.1.1. OpenMP - Aceleração

A Figura 4 apresenta o aceleração em função do número de *threads*. O resultado demonstra que ganho de aceração com número de *threads* próximas ao total de núcleos físicos, e apresenta desaceleração ao ultrapassar o número de núcleos físico. Entretanto, a execução com 8 *threads* retoma a aceleração. Esse comportamento pode ser explicado considerando o impacto na variabilidade de estado do *hardware*, como a carga ou execução de outras aplicações.

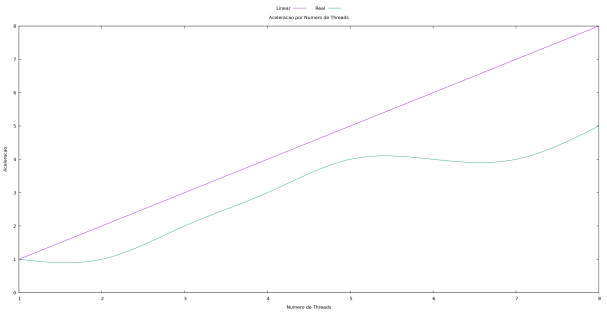

Figura 4. Aceleração com auxilio OpenMP.

6.1.2. OpenMP - Aceleração por Escalonamento

A Figura 5 apresenta a aceleração obtida com variação ao número de *threads* e configuração de escalonamento. É observável que o escalonamento guiado que desempenho super linear com menor número de *threads* e reduz ao ultrapassar o número de núcleos físicos. Por outro lado, o escalonamento estático e dinâmico demonstra comportamento parecido, ambos não apresentam ganho de aceleração em relação ao código serial. Este comportamento irregular pode ser decorrente de uma possível concorrência com outra aplicação.

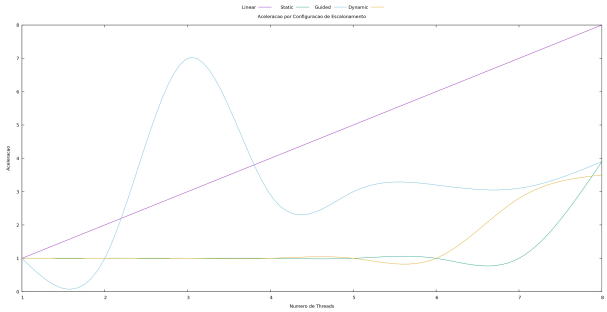

Figura 5. Aceleração com auxilio do OpenMP com diferentes configurações de escalonamento.

6.2. MPI: Aceleração

A aceleração foi calculada a partir do tempo de execução serial em segundos dividade pelo tempo de execução paralela em segundos. Essa aceleração obtida através de MPI é melhor que a obtida através do auxilio de OpenMP. Contudo a aceleração cresce quando os processos estão alocados nos mesmos computados, a medida que possuem mais processos distribuídos entres os demais computadores, devido a comunicação a aceleração reduz seu crescimento.

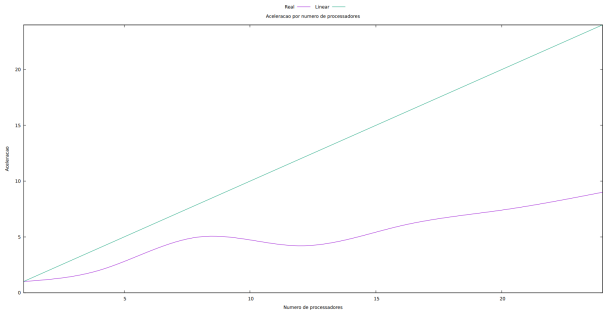

Figura 6. Aceleração com uso de OpenMPI.

6.3. MPI - Aceleração com Processadores Agrupado e Distribuídos

A relevância da etapa de mapeamento fica evidente a partir da curva de aceleração obtida, como visto na Figura 7.

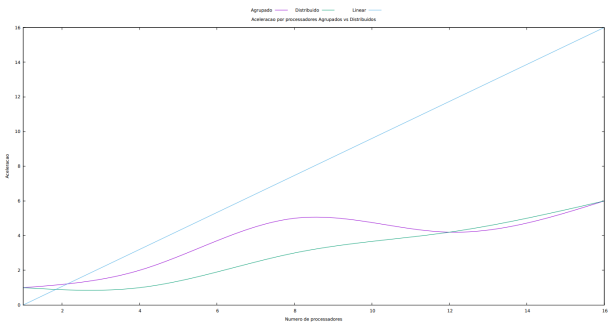

Figura 7. Aceleração com Processadores Distribuídos.

Na execução onde os processos são agrupados por computador, a aceleração obtida inicialmente é maior em relação a aceleração obtida na execução com processos distribuídos uniformemente entre os computadores. Por outro lado, ambas execuções a demonstram estabilidade quando aumenta o número de processadores, ou seja, quando não é possível agrupar os processadores em um único computador.

Processadores	4	8	12	16
Processadores Agrupados				
A	4	4	4	4
B	0	4	4	4
C	0	0	4	4
D	0	0	0	4
Processadores Distribuídos				
A	1	2	3	4
B	1	2	3	4
C	1	2	3	4
D	1	2	3	4

A Tabela apresenta as configurações utilizadas durante a execução de testes, em relação ao mapeamento de processos.

7. Considerações

O presente trabalho apresentou uma proposta de paralisação para o algoritmo Dijkstra, que é uma solução para o problema do caminho mínimo. Baseando-se na metodologia de análise de paralelização de algoritmos. Dessa forma a implementação através do auxilio de OpenMP e MPI foi realizada considerando a entrada de dados com objetivo de calcular o menor caminho entre os vértices, considerando obter o resultado de todos os vértices para todos os vértices. A primeira proposta definida através da metodologia PCAM não apresentou viabilidade, sendo assim, abordagens diferentes foram implementadas em ambas ferramentas. Por fim, a paralelização através de MPI apresentou maior aceleração em relação a implementação em OpenMP, sendo praticamente o dobro de melhora, apesar das alterações realizadas e o aumento em linhas de código para o MPI. Além disso, ficou evidente a importância do mapeamento para comunicação entre processos próximos em multicomputadores. Por fim, a implementação que utilizou auxilio do OpenMP destacou-se pela sua trivialidade de implementação comparado a implementação que utilizou MPI.

Referências

Ahuja, R. K., Magnanti, T. L., Orlin, J. B., and Reddy, M. (1995). Chapter 1 applications of network optimization. In *Network Models*. Elsevier.

Cáceres, E. N. (1192). *Algoritmos Paralelos para Problemas em Grafos*. PhD thesis, Universidade Federal do Rio de Janeiro.

da Costa, P. P. (2011). *Teoria de Grafos e suas Aplicações*. PhD thesis, Universidade Estadual Paulista "Júlio de Mesquita Filho".

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. 1:269–271.

Glover, F., Klingman, D. D., Phillips, N. V., and Schneider, R. F. (1985). New polynomial shortest path algorithms and their computational attributes. *Management Science*, 31(9):1106 – 1128.

Kurose, R. (2013). *Redes de computadores e a internet*. Pearson.

Schnorr, L. and Nesi, L. L. (2019). Capítulo 2 projetando e construindo programas paralelos. In *Escola Regional de Alto Desempenho da Região Sul*.