Prova 1: PCV com MPI e PThreads

Jéssica Bargas Aissa

29 de maio de 2017

1 Introdução

O presente trabalho apresenta uma solução paralela para o problema do caixeira viajante (PCV), segundo o algoritmo descrito, utilizando OpenMPI e PThreads.

2 Descrição do algoritmo

Dada uma matriz NxN, o algoritmo sequencial apresentado calcula recursivamente os caminhos de menor custo a partir de 0 até um vértice v. Dessa maneira, podemos construir uma árvore de dependências da seguinte maneira:

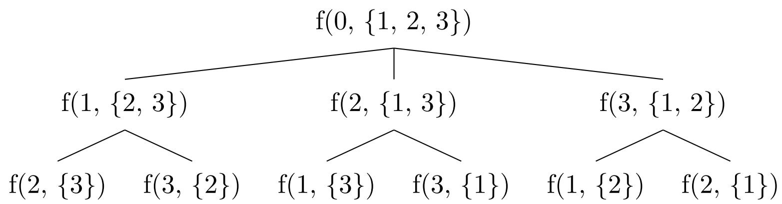

Figura 1: Árvore de tarefas independentes.

Para uma dada função $f(p, V)$, onde $V = \{v_1, v_2, ..., v_N\}$, os cálculos de $f(v, V - \{v\})$, $v \in V$, podem ser feitos independentemente, e assim de forma recursiva para todos os níveis da árvore.

3 Descrição da solução

3.1 Divisão de tarefas

Para calcular a função $f(0, \{1, 2, ..., N - 1\})$, que representa o caminho de custo mínimo do grafo inteiro, com P processos, tomam-se as N-1 funções $f(v, V')$, $V = \{1, 2, ..., N - 1\}$, $V' = V - \{v\}$, e atribui-se N/P funções a cada processo (iremos chama as funções f de tarefas). Por exemplo, se $N = 6$, $P = 2$, tem-se a seguinte atribuição de tarefas a cada processo P_i, que definem o seu conjunto de tarefas:

P_0: $f(1, \{2, 3, 4, 5\}), f(2, \{1, 3, 4, 5\}), f(3, \{1, 2, 4, 5\})$
P_1: $f(4, \{1, 2, 3, 5\}), f(5, \{1, 2, 3, 4\})$

Nota-se que quando a divisão de $N-1/P$ não é exata, o último processo P_{P-1} recebe $(N-1)\%P$ tarefas. Para o caso onde $P > N$, atribui-se uma tarefa a cada processo. Os processos restantes executam, porém não realizam nenhum cálculo efetivo para achar o caminho mínimo.

Cada processo contém duas informações importantes sobre suas tarefas: a quantidade de tarefas pelo qual é responsável (R) e o valor inicial (start) de v para o seu conjunto de tarefas. Seguindo o exemplo acima, temos:

P_0: $R = 3$, $start = 1$
P_1: $R = 2$, $start = 4$

A partir desses valores, cada processo cria uma *pool* de valores para cada v do seu conjunto de tarefas. Uma *pool* é uma estrutura do tipo pilha (implementada com vetor) que contém todos os valores de V' para $f(v, V')$ para um dado v. Tomando a função $f(4, \{1, 2, 3, 5\})$ do processo P_1, a sua respectiva *pool* seria uma pilha [5, 3, 2, 1], onde 1 é o topo da pilha e 5 o fundo da pilha. A *pool* também identifica a qual v pertence cada pilha (no caso, a pilha [5, 3, 2, 1] pertence a 4 no processo 1). Para controlar as diversas *pools*, cada processo mantém uma *pool list*. Para o exemplo dado, temos a *pool list* de cada processo:

Pool list(P_0):
Pool(1): 5, 4, 3, 2
Pool(2): 5, 4, 3, 1
Pool(3): 5, 4, 2, 1

Pool list(P_1):
Pool(4): 5, 3, 2, 1
Pool(5): 4, 3, 2, 1

Cada processo contém um número T de threads. Seria suficiente que cada thread tomasse uma função $f(v, V')$ do processo para calcular, no entanto, a pequena quantidade de tarefas atribuída a cada processo implicaria que o número de threads efetivas (que realizam algum tipo de cálculo útil) estaria restringida a R, R no máximo igual a N-1. Em vez de atribuir somente um valor v para cada thread, atribui-se também um valor v' em V'. Esses valores são atribuídos de forma aleatória: ao iniciar, cada thread busca v e v' da pool list do processo. Como a execução das threadas é não-determinística, não é possível saber qual thread irá adquirir quais v e v'. Uma possível distribuição de tarefas pode ser:

P_0:
T_0: v = 1, v' = 2
T_1: v = 1, v' = 3
T_2: v = 1, v' = 4
T_3: v = 1, v' = 5
T_4: v = 2, v' = 1
T_5: v = 2, v' = 3

P_1:
T_0: v = 4, v' = 1
T_1: v = 4, v' = 2
T_2: v = 4, v' = 3

T_3: v = 4, v' = 5
T_4: v = 5, v' = 1
T_5: v = 5, v' = 2

Dessa maneira, a ocupação das threads não está mais no segundo nível da árvore, mas no terceiro. Isso faz com que até (N-1)*R threads realizem cálculos úteis para um processo.

Cada processo mantém um vetor global de mínimos. O processo P_0, por exemplo, deve calcular os mínimos de v = 1, v = 2 e v = 3. Logo, existe um vetor de mínimos de tamanho 3, onde cada posição i representa uma função $f(v, V')$. Cada thread realiza o cálculo recursivo de $f(v', V'')$, $V'' = V' - \{v'\}$ e atualiza o valor no vetor de mínimos, caso o valor encontrado seja menor do que o valor atual do vetor. Tomando o exemplo de divisão de tarefas acima, percebe-se que, no processo 0, as threads T_0, T_1, T_2 e T_3 calculam concorrentemente $f(1, \{2, 3, 4, 5\})$, sendo que cada uma calcula $f(v', V'')$, $v' \in \{2, 3, 4, 5\}$, $V'' = \{2, 3, 4, 5\} - v'$. Ao terminar, a posição 0 do vetor de mínimos deve possuir o valor de $f(1, \{2, 3, 4, 5\})$, que representa o custo mínimo do caminho de 0 a 1, passando por 2, 3, 4 e 5.

Percebe-se que quando $T < (N - 1) * R$, nem todos os cálculos de f foram realizados. Assim, ao terminar de executar, uma thread adquire um novo v' e um novo v" e executa da mesma maneira com esses novos valores. Se a pool list do processo estiver vazia, isso significa que todos os $f(v, V')$ foram calculados e todos os mínimos foram obtidos. As threads podem ser então terminadas.

Para achar o mínimo local, o processo soma a distância de v a 0, representada pela posição $(0, v)$ na matriz de adjacências, ao seu respectivo mínimo, e depois esses valores são calculados para obter o mínimo local do processo. No exemplo apresentado, os processos retornam os seguintes mínimos locais:

P_0: $min\{c_{01} + f(1, \{2, 3, 4, 5\}), c_{02} + f(2, \{1, 3, 4, 5\}), c_{03} + f(3, \{1, 2, 4, 5\})\}$
P_1: $min\{c_{04} + f(4, \{1, 2, 3, 5\}), c_{05} + f(5, \{1, 2, 3, 4\})\}$

Com o retorno de cada processo, o mínimo entre os valores retornados representa, então, o custo mínimo do caminho $f(0, V - \{0\})$ para o grafo dado.

3.2 Cálculo recursivo do caminho mínimo

Cada thread executa, essencialmente, uma função recursiva para cálculo do caminho mínimo. Dado $f(k, K)$, $K = \{k_1, k_2, ..., k_m)\}$, chama-se recursivamente essa função para cada $k' \in K$, com $K' = K - \{k'\}$. A cada valor retornado soma-se o custo $c_{kk'}$ e o mínimo é atualizado, caso essa soma seja menor do todos os outros valores anteriores. Quando só houver um elemento k' em K, a função retorna a soma dos custos $c_{kk'}$ e $c_{k'0}$.

Essa mesma função utiliza da estrutura *pool* para empilhar os vértices visitados recursivamente, gerando assim o caminho percorrido que representa o custo mínimo. Para cada k' em K gera-se então essa pilha, e ao final retorna-se apenas a pilha que representa o custo mínimo encontrado.

4 Descrição do código

O código descrito em *pcv.c*, chamado de master, toma uma matriz NxN de um arquivo de entrada e cria P processos MPI que executam o código em *slave.c*. Os valores de N e T determinados pelo arquivo de entrada são passados ao slave de rank 0, que então realiza o broadcast para os outros processos. O master então calcula o start e o R de cada processo e realiza um scatter dos valores. Por fim, a matriz é enviada ao slave de rank 0, que então a envia por broadcast para os outros slaves.

Nesse ponto, o master espera o resultado dos slaves. Esses, por sua vez, após inicializar as variáveis necessárias, iniciam as threads, que devem calcular os custos mínimos e seus respectivos caminhos como descrito acima. Os processos então comparam os valores locais e retornam o custo mínimo e o caminho mínimo ao master com um gather. O master compara os valores para encontrar o custo mínimo global e retorna seu valor e o caminho descrito.

Para execução das threads, utilizam três mutex e duas variáveis locais essenciais a cada thread, *local_p* e *local_sub*, que representam v e v' respectivamente no algoritmo descrito acima. A primeira, *local_p*, é obtida a partir de uma variável global p do processo, iniciada com o valor de start. A variável *local_sub* é obtida dando um pop na *pool* de *local_p*. Se a pilha estiver vazia, p é incrementado e a thread tenta obter um novo *local_p* e um novo *local_sub* (ou apenas um novo *local_sub*). Caso todas as pilhas estejam vazias, todos os valores já foram processados e a thread pode ser encerrada. Os três mutex em questão servem a três propósitos: o primeiro, *pl_lock*, é responsável pelo acesso ao estado de vazio/cheio da pool list do processo; o segundo, *p_lock*, é responsável pelo acesso à variável p e à sua *pool*; e o terceiro é responsável pelo acesso ao vetor de mínimos e ao vetor de pilhas respectivas a cada caminho.

Para calcular o custo mínimo e o caminho mínimo, a função *min_path* recebe um valor p, uma lista de vertices e seu tamanho, e retorna uma *pool* e um valor inteiro que representam o caminho e o custo, respectivamente. Seu funcionamento está descrito na seção anterior. Para retornar o caminho ao master, a *pool* retornada pela função é convertida em um vetor de inteiros.

O código em *pool.c* descreve os funcionamentos das *pools* (pilhas associadas a um valor id) e das *pool lists*.