MAC0438 - Programação Concorrente - 1s2012
Relatório

Thiago de Gouveia Nunes
Wilson Kazuo Mizutani

May 13, 2012

Contents

1 Introdução

Esse relatório trata das decisões tomadas na implementação desse EP. Também fornece uma saída do programa para a entrada de exemplo fornecida no enunciado, além de explicitar a localização no código da implementação da barreira simétrica usada. Informações sobre como como compilar, as dependÊncias necessárias e o modo de uso do programa encontram-se no arquivo LEIAME.

2 Solução desenvolvida

2.1 Ideia geral

A ideia geral da nossa solução se divide em duas partes:

1. Estender o algoritmo de busca em largura (Breadth-First Search, um caso particular do algoritmo de Dijkstra no qual todas as arestas possuem custo 1) para que ele encontre não só o menor caminhos, mas sim os **n** menores caminhos.

2. Refatorar esse algoritmo para usar programação concorrente, de tal maneira que cada thread seja responsável por tentar encontrar um novo caminho e depois sincronizar com as demais, criando assim um processo iterativo.

2.2 Estendendo a busca em largura

Basicamente, aproveitamos a propriedade da busca em largura na qual um novo vértice retirado da fila está sendo visitado pela primeira vez através do menor caminho.

Estendemos o algoritmo para ter uma fila de caminhos ao invés de vértices, por conveniência, e ao invés de deixarmos de visitar um vértice após passar por ele apenas uma vez, o fazemos após **n** vezes (o que significa que ele já tem n caminhos mínimos terminando nele).

Assim, a propriedade do nosso algoritmo (ainda no caso não-concorrente) seria que um novo caminho retirado da fila é o próximo menor caminho que termina no mesmo último vértice que ele. Mas como o enunciado do EP pedia que cada thread cuidasse de apenas um caminho, mudamos isso de forma que, na verdade, insere-se apenas candidatos a caminho na fila. A propriedade fica portanto que um novo candidato retirado da fila *pode* ser o próximo menor caminho que termina no seu último vértice, *contanto que ele não seja um ciclo.*

Uma consequência disso é que o programa vai com certeza ficar menos eficiente, pois há menos restrições sobre quem entra na fila, e portanto ela potencialmente terá mais elementos do que na maneira anterior.

2.3 Aplicando concorrência

Pensamos em várias maneiras de fazer isso. O grande problema é que a ordem com que os caminhos são encontrados é indeterminada, então não podemos assumir que os n primeiros encontrados sejam, de fato, os n menores. Vimos duas maneiras de lidar com isso.

Uma seria controlar a inserção em uma fila compartilhada através de um buffer intermediário que garantisse sua ordem. Isso apresenta dois problemas: a necessidade de uma fila compartilhada, e o fato de que a inserção nela não seria verdadeiramente paralelizada, pois seria justamente preciso serializar as threads para que elas inserissem na ordem correta.

A outra maneira e deixar o trabalho todo para a lista de caminhos de cada vértice. Elas mesmas cuidariam de guardar o n menores caminhos, descartando os demais. Assim, cada thread poderia ter sua própria fila, e no total o algoritmo seria bem mais paralelizável. O problema seria que ua inserção de um dos n verdadeiramente menores caminho custaria com a melhor implementação que pudemos pensar O(log(n)) ao invés de apenas O(1).

No final, optamos pela segunda opção, apostando que uma maior paralelização do algoritmo compensasse a inserção de novos caminhos mínimos.

3 Implementação da solução

Fizemos uma série de classes para organizar a estrutura do programa, mas não existe uma classe principal que cuida de tudo. Quem faz isso são as funções encontradas nos arquivos **src/ep2.h** e **src/ep2.cxx**, que de fato usam as classes que fizemos.

3.1 Breve descrição das classes

Explicamos aqui o papel de cada classe, organizadas de acordo com suas características em comum.

Classes utilitárias:

Log: Responsável por imprimir a saída do programa de maneira organizada, omitindo mensagens de debug ou avisos internos quando estes não forem explicitamente requisitados pelo usuário.

Classes de manipulação de grafos:

Graph: Responsável por representar um grafo cujas arestas têm todas custo 1. Usamos uma matriz de adjacência.

Path: Responsável por representar um caminho no grafo.

Classes de manipulação de threads:

Thread: Responsável por representar uma thread do programa. É implementada usando a libpthreads.

Barrier: Responsável por representar uma barreira simétrica. Usa a implementação de barreira de disseminação.

Classes destinadas à implementação algoritmo:

PathSeeker: Interface que todo buscador de caminhos deve implementar. É importante notar que essa classe não diz nada a respeito do uso de programação concorrente. Quem faz isso são as classes que herdam dela.

SimplePathSeeker: Implementação de PathSeeker **sem concorrência**. Sua implementação segue a ideia formulada na seção 2.2.

MultiPathSeeker: Implementação de PathSeeker **com concorrência**. Sua implementação segue a ideia formulada na seção 2.3.

3.2 Alguns detalhes de implementação

Como pode ser observado nas classes destinadas à implementação do algoritmo, nosso programa tem suporte tanto à solução simples quanto à solução paralelizada (segundo as ideias apresentadas na seção 2). No arquivo LEIAME está explicado como escolher entre uma e outra na chamada do programa, mas o padrão é sempre a solução concorrente.

Outro detalhe importante é que a classe Thread não tem um atributo identificador em si, pois quem cuida de identificar as threads é a classe MultiPathSeeker. Ou seja, na prática, a saída aparece como o enunciado pedia.

Para fins de eliminar vazamento de memória, tivemos que fazer um vetor com todos os caminho criados, para ao final do programa deletarmos todos da memória.

3.3 Sobre a barreira simétrica usada

Como já mencionado, optamos por uma barreira simétrica de disseminação. A implementação exata do seu algoritmo encontra-se no método **synchronize(id)** da classe **Barrier**.

Sua declaração encontra-se no arquivo src/barrier.h, linha 17, e sua implementação no arquivo src/barrier.cxx, linhas 36 a 44.

Um detalhe meio bobo que difere nosso algoritmo do apresentado em aula é que nosso contador de estágio (s) começa do 0 e não do 1. Logo, para calcular a distância entre os pares que se sincronizam a cada estágio usamos 2^s ao invés de 2^{s-1}.

Outra informação relevante é que fizemos uma solução não muito usual para lidar com a espera de threads já mortas na barreira (o que causava *deadlocks*). Fizemos o valor de arrive[id], com id sendo o identificador da thread que já foi encerrada, receber o valor -1. Mas como o tipo de arrive[i] é unsigned, isso dá underflow e assume o valor máximo que esse tipo suporta. Sendo o valor máximo, nenhuma outra thread esperará por essa que já não existe mais, mas a estrutura da barreira permanece a mesma. Como pudemos supor em aula que os valores de arrive praticamente nunca chegariam ao máximo do tipo int e similares, então essa solução não deve causar problemas.

4 Resultados

4.1 Resultado para o grafo da NSFNet

4.2 Comparações de eficiência

5 Conclusões