Igor da Silva Solecki

REPRESENTAÇÃO OTIMIZADA DE TOPOLOGIAS DE MÁQUINA

Florianópolis, SC
29 de maio de 2017

Igor da Silva Solecki

REPRESENTAÇÃO OTIMIZADA DE TOPOLOGIAS DE MÁQUINA

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Prof. Dr. Laércio Lima Pilla
Universidade Federal de Santa Catarina

Florianópolis, SC
29 de maio de 2017

LISTA DE ABREVIATURAS E SIGLAS

tst	tst
etc	...

1 IMPLEMENTAÇÃO

Esta seção apresenta como o projeto hwloc foi analisado, as considerações feitas com relação ao desempenho e as implementações resultantes. Todo o código foi desenvolvido na linguagem C++.

O hwloc possui diversas funções de percorrimento. Estas permitem acessar nós da árvore que representa a hierarquia de forma absoluta ou relativa a outros nós. Por exemplo, é possível encontrar o nó com um determinado índice dentro de um dado nível, ou, a partir de algum nó, o próximo no mesmo nível. !!! Listar funções !!!

Essas funções foram analisadas quanto à complexidade com o objetivo de identificar pontos que poderiam ser melhorados do ponto de vista do desempenho. Essa análise revelou que, em geral, elas têm tempo constante (O(1)) ou linear na altura da árvore (O(*altura*)). Além disso, foi analisado um projeto de código aberto que utiliza o hwloc, HieSchella !!! [ref] !!!, cujo objetivo é prover portabilidade de performance, característica presente quando se consegue que uma mesma aplicação rode em diferentes plataformas utilizando os núcleos com eficiência. Foram identificadas as chamadas mais importantes a funções do hwloc no HieSchella para se ter uma referência de quais funções são mais relevantes para o desempenho dentro de um projeto real.

Diante dessas considerações, a função que encontra o ancestral comum mais próximo (ACMP) entre dois nós foi escolhida como alvo de otimizações. Ela é uma das funções implementadas no hwloc com complexidade O(*altura*)e está entre as de uso mais significativo no HieSchella. Na seção a seguir, será discutido como essa função poderia ser implementada de forma mais eficiente e quais as implicações de diferentes abordagens.

1.1 IMPLEMENTAÇÕES DA FUNÇÃO ACMP

A função que encontra o ancestral comum mais próximo recebe dois nós como entrada (e possivelmente algumas estruturas adicionais, se houver necessidade) e retorna um nó (o ancestral) como saída. Cada

par de nós em uma árvore tem exatamente um ancestral comum mais próximo.

1.1.1 Abordagem inicial

A maneira provavelmente mais intuitiva de se descobrir o ACMP é "subir" pela árvore, isto é, a partir dos dois nós dos quais se deseja encontrar o ACMP, seguir as ligações em direção aos pais até se chegar ao mesmo nó. Por ser um método que utiliza apenas a estrutura de árvore em si, sem adição de outras informações, será referido como método simples. Sua complexidade é O(*altura*). Para que esse método funcione, é necessário subir de forma sincronizada – a cada passo, devem-se comparar ancestrais dos dois nós iniciais que estejam no mesmo nível. No caso do hwloc, o algoritmo se torna um pouco mais complicado, pois ele trata hierarquias assimétricas (pode haver ramos sem nó em algum nível). Neste caso, mesmo que se tenham dois nós no mesmo nível, seus pais podem estar em níveis diferentes. Um ponto que pode afetar o desempenho deste algoritmo é o fato de que é necessário acessar cada nó (os nós iniciais e todos os seus ancestrais até o ACMP), e os nós estão espalhados pela memória.

!!! (Algoritmo ACMP do hwloc) !!!

Considerando a estrutura de árvore apenas, a única informação que relaciona um nó aos seus ancestrais são as ligações entre um nó e seus filhos (ou, no outro sentido, entre cada nó e seu pai). Assim, todas as sequências de uma ou mais ligações de filho para pai a partir de um nó (ou seja, entre o nó e seu pai, entre este e o pai dele, e assim por diante) definem os ancestrais do nó. Isso indica que outras estruturas associadas aos nós ou à árvore como um todo se fazem necessárias para ser possível encontrar o ACMP com algum método além do simples. Podemos considerar a seguinte ideia para encontrar outra maneira de implementar a função de ancestral comum: Para uma dada árvore que representa uma topologia,

1. Atribuir um valor (chamado de ID) a cada nó da árvore;

2. Definir uma função ACMP_{IDs} que receba o ID de dois nós distintos e tenha como resultado o nó ACMP.

Esses IDs (em conjunto com outras informações associadas a cada nó individualmente ou à árvore como um todo conforme necessário) podem estabelecer alguma relação entre um nó e seus ancestrais além da que já existe por meio das ligações da árvore.

Idealmente, essa função ACMP_{IDs} deveria ser *processada* em tempo constante. Ou seja, é necessário encontrar um algoritmo que seja executado em tempo constante nos processadores modernos. No entanto, é preciso lembrar que, mesmo que a quantidade de instruções executadas pelo processador seja constante, a maneira como a memória é acessada pode aumentar o tempo de execução, especialmente quando há outras tarefas fazendo uso da memória, o que deve acontecer em cenários reais. !!! (? Isso pode ser visto... (exemplo à frente com matriz?)) !!!

Com isso em mente, podemos definir tal função usando as operações básicas encontradas no conjunto de instruções das arquiteturas atuais, tais como as operações aritméticas e operações lógicas bit-a-bit.

1.1.2 Matriz

Uma possibilidade é relacionar cada par de nós ao seu ACMP por meio de uma matriz em que cada linha representa um nó da árvore, assim como cada coluna, e o cruzamento contém o ACMP entre o nó da linha e o nó da coluna. Para isso, pode-se atribuir a cada um dos n nós da árvore um ID único entre 0 e $n-1$ e usar esses IDs como índices na matriz, que terá, na posição $A(i, j)$, o ACMP entre o nó de ID i e o nó de ID j. No entanto, esta é uma estratégia ingênua, pois esse espaço $\text{O}(n^2)$ ocupado na memória resultaria em problemas como sujar a cache da aplicação. Visto que a matriz é simétrica, apenas cerca de metade dela precisa realmente ser armazenada. Esta otimização foi utilizada na implementação dos testes de desempenho, conforme o !!! [algoritmo X] !!!. Isso, no entanto, não altera a complexidade espacial.

Algoritmo 1 Ancestral usando matriz

function ANCESTRAL_MATRIZ(Matriz⟨Nó⟩ M, Nó a, Nó b)
  linha $\leftarrow \max(\mathrm{id}_a, \mathrm{id}_b)$
  coluna $\leftarrow \min(\mathrm{id}_a, \mathrm{id}_b)$
  retornar $M(\text{linha}, \text{coluna})$
end function

1.1.3 Função de espalhamento

Outra possibilidade foi idealizada, dividindo a função $\mathrm{ACMP}_{\mathrm{IDs}}$em dois passos:

1. dados os IDs de dois nós, descobrir o ID do ancestral;
2. encontrar o nó que possui esse ID.

Em linhas gerais, o funcionamento do primeiro passo se baseia no seguinte: O ID de um nó é formado por um ou mais bits seguidos do ID do seu pai, de modo que, dados dois nós a e b, b descendente de a, os bits menos significativos de b são iguais ao ID de a. Desse modo, dados dois descendentes de um nó c, todos os bits menos significativos deles que coincidem (todos os que vêm antes do primeiro que difere) são iguais ao ID de c (ignorando zeros à esquerda). Usando apenas as operações que podem ser vistas !!! nos algoritmos [X] e [Y] mais adiante !!!, para as quais existem instruções que tomam poucos ciclos nas arquiteturas atuais, pode-se descobrir o ID do ACMP. A quantidade de instruções é fixa, portanto, a complexidade é constante.

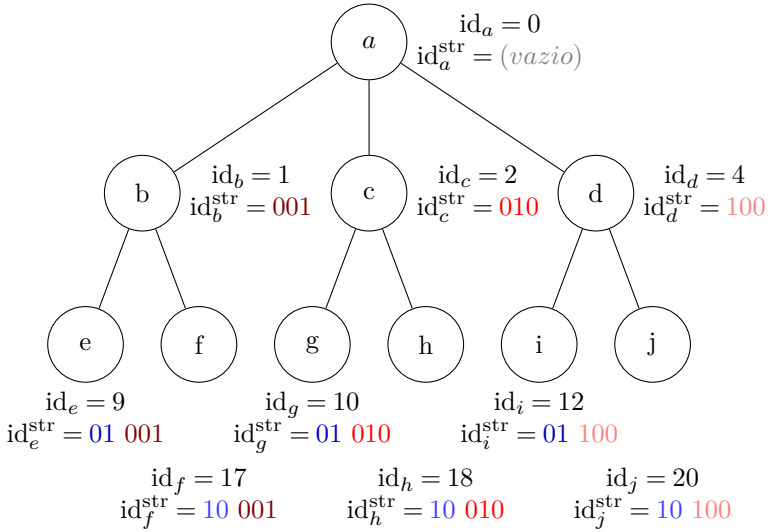
Para descrever como os IDs são formados, as seguintes definições são necessárias:

- id_a é o ID do nó a.
- $\mathrm{id}_a^{\mathrm{str}}$ é uma cadeia (*string*) de bits correspondente ao id_a em binário (com o bit menos significativo na última posição). O tamanho depende do nível de a, como será especificado adiante.

Os IDs, então, são definidos da seguinte forma:

- A raiz tem id 0, e $\mathrm{id}_{raiz}^{\mathrm{str}}$ é a cadeia vazia.

Figura 1 – Exemplo de árvore com IDs atribuídos aos nós



- Quanto aos demais, para cada nó a,

$$\mathrm{id}_a^{\mathrm{str}} = x \parallel \mathrm{id}_{pai(a)}^{\mathrm{str}}$$

  onde $\parallel$ é a concatenação e x é uma cadeia cujo tamanho é o grau do nível de pai(a). Se a é o i-ésimo filho de seu pai, x possui 1 na i-ésima posição da direta para a esquerda e 0 nas demais.

Assim, as cadeias correspondentes aos IDs de todos os nós de um nível têm o mesmo tamanho, que é o somatório dos graus dos níveis anteriores. A Figura 1 apresenta os IDs atribuídos aos nós uma árvore.

Falta, então, apenas o segundo passo, o de encontrar o nó a partir do ID. Algumas opções para isso seriam:

- Usar os IDs como índices em um arranjo: Seria simples, mas impraticável – poderiam ser necessários arranjos com milhões de posições (devido a como os IDs são formados) e apenas algumas centenas ocupadas.

- Usar uma função de espalhamento (*hash*): a maneira mais simples seria apenas aplicar a operação módulo com algum m (id mod m).

No entanto, podem haver colisões (dois IDs diferentes podem ser congruentes módulo m). Isso pode ser tratado, mas acarretaria acessos adicionais à memória, o que não é desejável. !!! Exemplo !!!

- Buscar uma função que não cause colisões: o mesmo que a função de espalhamento, porém utilizando um valor de módulo que não cause colisões. Pode exigir arranjos cujo tamanho é algumas vezes maior que a quantidade de nós, mas aparenta compensar quando comparado a tratar colisões de outros modos. !!! Exemplo !!! Este foi o método escolhido.

O !!! Algoritmo X !!! apresenta como esta
!!! [Algoritmo] !!!
!!! Estruturas foram testadas !!!
!!! Pode falhar se os dois nós de entrada são, na verdade, o mesmo nó. If resolve Funciona em casos de hierarquias assimétricas, simples não Limitação: Quantidade de bits necessária - Solução parcial: 'Esconder' níveis cujos nós têm só um filho (níveis de grau um). !!!

2 TESTES DE DESEMPENHO

Com o objetivo de avaliar o desempenho do algoritmo desenvolvido, foram realizados testes que permitiram comparar o desempenho das diferentes abordagens. Foi medido o tempo tomado pelos algoritmos ao se encontrar repetidamente o ACMP entre os nós folhas de uma árvore. Com esses dados, analisados

Um programa foi desenvolvido, também na linguagem C++, para realizar os testes de desempenho. Foi utilizada a funcionalidade de templates da linguagem para facilitar a definição equivalente dos testes para todos os algoritmos, mas ainda assim permitir que o compilador otimizasse as chamadas, evitando custos adicionais durante os testes devido à hierarquia de classes utilizada. O programa depende da biblioteca do hwloc. Os valores medidos são escritos em um arquivo no formato CSV (Valores Separados por Vírgula – *Comma-Separated Values*)

2.1 ESTRUTURA DOS TESTES

Os testes consistem em encontrar repetidamente o ACMP entre os nós folhas de uma dada árvore com os diferentes algoritmos – o existente no hwloc e os implementados. Cada vez que as estruturas necessárias são criadas e os testes são executados sobre elas com um determinado algoritmo, obtém-se uma observação, que é o tempo que levou para realizar a quantidade especificada de repetições da função ACMP entre os nós folhas da árvore com o algoritmo.

Para cada algoritmo, uma árvore de estrutura equivalente é criada (as estruturas de dados dependem do algoritmo) a partir dos graus fornecidos como entrada. Os graus são recebidos como uma lista de inteiros positivos. O i-ésimo será o grau do nível $i-1$. Deste modo, se os graus recebidos são $(g_1, g_2, ..., g_n)$, a raiz terá g_1 filhos, cada um com g_2 filhos, e assim por diante, até os nós do nível $n-1$, que terão g_n filhos cada.

Para obter uma observação de um algoritmo, uma lista contendo

todos os pares possíveis de nós folhas (todas as combinações de duas folhas) é criada. Esta lista é embaralhada, usando uma semente fixa, de modo que a ordem pseudo-aleatória é a mesma para todas as execuções de todos os algoritmos sobre esta árvore.

Uma observação é obtida por meio de uma etapa de aquecimento seguida de uma de medição, na qual o tempo total das repetições da função ACMP é medido. Ambas as etapas consistem em algum número de rodadas, o qual geralmente deve estar na casa de alguns milhares, dependendo do tamanho da árvore, para que o tempo medido seja significativo. Em cada rodada, a lista previamente embaralhada de pares de folhas é varrida e, para cada par, o ACMP é encontrado usando o algoritmo em questão.

As observações dos diferentes algoritmos são realizadas de forma intercalada. O número de observações obtidas para cada algoritmo em uma execução do programa depende de dois parâmetros, número de iterações externas e de iterações internas. O número de iterações internas é a quantidade de observações que serão obtidas para um dos algoritmos antes de passar para outro. A execução da quantidade de iterações internas para cada algorimo compõe uma iteração externa. Esta forma de especificar a quantidade de observações originou-se nas etapas iniciais dos testes, para facilitar a visualização dos resultados, mas foi mantida. No entanto, julga-se melhor usar poucas iterações internas e mais externas para evitar que eventuais condições temporárias da máquina, causadas por elementos externos ao programa, afetem diversas observações de apenas um dos algoritmos.

O programa também permite escolher quais algoritmos serão testados. Os possíveis são o simples, o que utiliza as novas estruturas, o que utiliza uma matriz e o implementado no hwloc.

(Tabela com os parâmetros?)

(Os resultados de testes feitos indicaram que a operação mais custosa no novo método era o módulo. No entanto, Divisão com denominador previamente conhecido [referência] Não em tempo de compilação, mas quando se está montando as estruturas. Portanto, é possível substituir essa operação de módulo por outras operações mais

baratas, a saber, duas multiplicações, um deslocamento e uma subtração. Com esta alteração, os tempos diminuiram consideravelmente, conforme mostra a tabela [X])

2.1.1 RESULTADOS

Inicialmente, os testes foram feitos sem acessar o ACMP obtido a cada execução, ou seja, para cada par de folhas, simplesmente se encontrava o ponteiro para o ancestral, mas nenhum dado do ancestral era obtido (apenas era feito algo com o ponteiro retornado para evitar otimizações que descartassem tudo devido ao resultado não ser usado). Nessas condições e na máquina A, os resultados apresentados (na tabela [X]) foram obtidos.

Comparação - O desempenho do algoritmo desenvolvido foi comparado com o do algoritmo simples, o do hwloc (e o que usa a matriz)(?)

As árvores utilizadas nos testes correspondem à hierarquia de memória de máquinas reais, apresentadas no site do projeto hwloc, como representadas pelo programa lstopo. Assim, os resultados refletem a diferença dos algoritmos quando operando sobre hierarquias reais. (Referência: The Best of lstopo - https://www.open-mpi.org/projects/hwloc/

Outras medições?