Árvore Binária de Busca Ótima - Uma Implementação Distribuída

Felipe Reis e Caio Valentim
Departamento de Informática
PUC-Rio

7 de novembro de 2010

1 Introdução

O problema de encontrar a árvore binária de busca ótima para um conjunto de frequências é estudado há anos no meio acadêmico. O melhor algoritmo exato conhecido é uma programação dinâmica de complexidade $O(n^2)$ [3].

Contudo, o algoritmo sequencial não é facilmente paralelizável. Existem algumas propostas, como em [2] e [1]. Mas, em geral, o apresentado não é uma solução exata ou é complicado demais.

Neste documento apresentamos nosso desenvolvimento e implementação de uma versão distribuída simples para o problema.

2 Definição do Problema

O problema em questão é o de encontrar uma árvore binária de busca ótima para um conjunto de chaves dadas as frequências de cada chave. Formalmente, temos as chaves $A = \{a_1 \leq a_2 \leq \ldots \leq a_n\}$, as frequências de buscas com sucesso de cada chave $F = \{f_1, f_2, \ldots, f_n\}$ e as frequências de buscas sem sucesso $Q = \{q_0, q_1, \ldots, q_n, q_{n+1}\}$. Onde q_i representa a frequência de buscas por chaves entre a_i e a_{i+1}.

Nesse trabalho iremos assumir que todas as buscas são com sucesso. Ou seja, iremos descartar o conjunto Q. Desta forma, queremos criar uma árvore binária de busca que minimize a seguinte função:

$$\sum_{i=1}^{n} f_i \times n(i)$$

onde $n(i)$ é o nível do nó i na árvore.

3 Artigos Relacionados

Os dois principais artigos estudados foram [2] e [1].

No primeiro, M. Karpinski propõe uma solução paralela capaz de calcular a solução ótima em tempo $O(n^{1-\epsilon})$ para uma constante arbitrária $0 < \epsilon \leq 1/2$. Ele agrupa as diagonais em conjuntos e calcula cada conjunto de forma eficiente a partir do pré-processamento de uma estrutura que ele chama "sub-árvores especiais". Apesar de interessante, a arbodagem não é fácil de implementar, especialmente para quem não possui grande experiência com codificação em paralelo.

Em outra linha, C. Mitica propõe em [1] uma solução paralela que começa dividindo o conjunto de chaves em intervalos menores, calculando as árvores ótimas para cada intervalo menor e, a partir das árvores de cada sub-intervalo, constroi uma solução para todo conjunto. Essa solução não foi implementada pois acreditamos que a abordagem está errada e, de fato, não produz uma árvore ótima.

4 Solução Sequencial

Abaixo dois lemas úteis para construção do algoritmo sequencial.

Lemma 4.1 (Critério de Otimalidade)**.** *Seja $OPT(i, j)$ o custo da árvore ótima para as chaves $\{a_i, \ldots, a_{j-1}\}$. A seguinte recorrência vale:*

$$OPT(i,j) = \min_{i \leq k < j} \{OPT(i,k) + OPT(k+1,j)\} + \sum_{t=i}^{j-1} f_t$$

Lemma 4.2 (Princípio da Monoticidade)**.** *Seja r a chave que será a raiz da árvore ótima do intervalo $[i, j)$ e $[i', j')$ outro intervalo tal que $i \leq i' \leq j \leq j'$ com raiz ótima r'. Vale que $r \leq r'$.*

Com o primeiro resultado é fácil construir uma tabela com a solução ótima em tempo $O(n^3)$, usando o segundo lema podemos reduzir a complexidade para $O(n^2)$. Contudo, nossa versão distribuída se basea na construção $O(n^3)$ implementada no capítulo 8 do livro [4].

5 Algoritmo Distribuído

O algoritmo, derivado do segundo lema, na prática, consiste em preencher diagonal por diagonal de uma tabela de dimensões $n \times n$. Desta forma, como o tempo para calcular cada diagonal é proporcional a $O(n^2)$, decidimos paralelizar esse cálculo. Existem n diagonais então, de forma geral, nosso algoritmos consiste em:

```
for i = 0 to n
do
1.    Cada processo calcula um pedaço da i-ésima diagonal
2.    Cada processo distribui o seu pedaço entre os outros processos
done
```

Ou seja, cada processo fica resposável por um pedaço da diagonal que está sendo calculada no momento. Após o cálculo, os processos têm que comunicar sua parcela aos outros processos. O procedimento se repete até que não existam mais diagonais para calcular.

Com isso, o tempo esperado para o cálculo de cada diagonal é proporcional a $O(n^2/p+np)$. A primeira parcela da soma se refere ao processamento paralelo da diagonal e a segunda ao custo extra de comunicação.

6 Implementação

6.1 Detalhes do Cluster

O programa foi rodado no cluster da PUC-Rio que conta com 64 nós. Abaixo temos uma descrição técnica do cluster retirada do site do suporte do departamento. A versão do MPI usada foi LAM 7.0.6/MPI 2 C++.

O site da PUC é formado atualmente por três clusters: o primeiro, com 12 estações; o segundo, com 20 estações; e o terceiro, com 32 estações. Todos estão situados fisicamente no server farm do DI. A arquitetura de cada cluster é homogênea: os nós do primeiro cluster têm CPUs Intel Core 2 Duo 2.16 GHz e 1 GB de RAM; os do segundo têm CPUs Intel Pentium IV 1.70 GHz e 256 Mb de RAM; e os do terceiro têm CPUs Intel Pentium II 400 MHz (Deschutes) e 280 Mb de RAM.

Os sistemas operacionais instalados são: no primero cluster, `Fedora Core 8 kernel 2.6.25.4-10`; e, nos demais, `Red Hat Linux versão 9 kernel 2.4.20-31.9`. A versão de globus é 2.4 em todos os clusters. As versões de MPI disponíveis são `lam-7.1.2`, `lam-7.0.6` e `mpich-1.2.6`.

O domínio do cluster é par.inf.puc-rio.br. Os nós estão numerados do n00 até o n63. O ponto de acesso (via SSH) é a máquina server.par.inf.puc-rio.br. A partir dela, os nós podem ser acessados via RSH ou SSH. As conexões entre os nós e o switch são de 100Mbs.

Cada usuário possui uma área própria de trabalho em cada nó, localizada no `/home/local/login_usuár` e acesso remoto a sua área de trabalho na máquina `server.par.inf.puc-rio.br`, através da pasta `/home/server/login_usuario`.

6.2 Executando Código MPI

Uma vez que usamos a implementação LAM do MPI disponível no cluster, alguns passos foram necessários para conseguir efetivamente executar nosso código.

Em primeiro lugar, antes de rodar o programa, deve-se criar uma associação entre um conjunto de máquinas (LAM). Isso é feito com o camando `lamboot -v machines.txt`. O flag `-v` é de verbose e o arquivo `machines.txt` contém as máquinas que serão utilizadas. Para o comando funcionar, uma série de requisitos devem ser observados. Por exemplo, as máquinas devem permitir acesso ssh sem senha. Para verificar se o ambiente está corretamente configurado, podemos usar o camando `recon -v` que lista possíveis problemas.

Além do ambiente de execução, precisamos de um código compilado. Para compilar utilizamos o `mpiCC`, compilador `C++` para código MPI. Com o código compilado, o próximo passo é distribuir o executável entre as máquinas do cluster. O suporte do DI disponibiliza o script *mrcp* para realizar esta tarefa.

Por fim, com todos os passos anteriores feitos, podemos executar, por exemplo, `mpirun -np 8 a.out`, que roda o programa `a.out` de forma distribuída entre 8 processos.

7 Experimentos

Os experimentos foram executados com $1, 4$ e 8 processos rodando em máquinas separadas. Testamos com entradas de tamanho $10, 50, 100, 500, 1000$ e 5000. O tempo de execução para cada par entrada x números de processos é a média de 10 execuções.

Abaixo a tabela dos resultados obtidos:

p	10	50	100	500	1000	3000	5000
1	0.521	0.522	0.520	0.850	3.344	107.159	503.236
4	0.555	0.571	0.580	0.910	2.569	63.275	295.765
8	0.58	0.610	0.650	1.061	2.379	38.024	172.599

E os resultados acima estão dispostos também na figura 1.

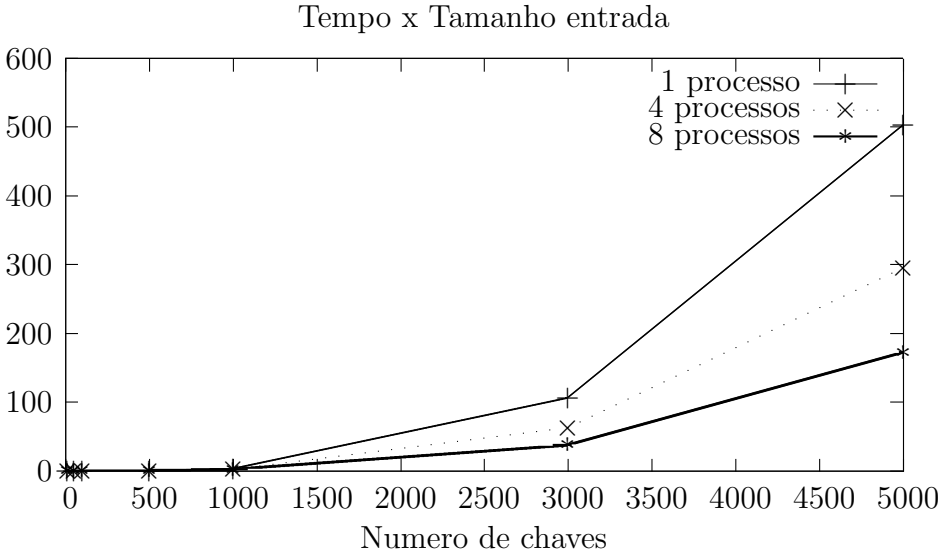

Figura 1: Tempos de Execução

8 Conclusão

De acordo com os experimentos realizados, é possível notar uma melhora significativa no tempo de execução total para entradas superiores a 1000 chaves, como esperávamos alcançar utilizando computação distribuída.

Analisando o teste de 5000 chaves, por exemplo, temos um ganho de tempo de $291,5\%$, no comparativo entre a execução sequencial e a execução distribuída em 8 processos.

Outro resultado que também é possível identificar através dos experimentos realizados, é o ônus introduzido pela comunicação entre processo, necessária para mantê-los sincronizados, na versão distribuída do algoritmo. Se observarmos a execução do algoritmo para um conjunto de 500 chaves, por exemplo, percebemos que a execução entre 8 processos é cerca de 11% mais custosa do que a execução para 4 processos, e 20% mais ineficiente do que a versão paralela.

Em suma, o algoritmo distribuído, para o problema de encontrar a árvore de binária de busca ótima, apresentou-se bem mais eficiente, se comparado a sua versão sequencial para um conjunto grande de chaves. No entanto, executando-o com poucas chaves, sua utilização não compensa o tempo gasto com comunicação e sincronização entre processos.

Referências

[1] Mitica Craus. Parallel and distributed solutions for the optimal binary search tree problem. In *IWCC '01: Proceedings of the NATO Advanced Research Workshop on Advanced Environments, Tools, and Applications for Cluster Computing-Revised Papers*, pages 103–117, London, UK, 2002. Springer-Verlag.

[2] Marek Karpinski and Wojciech Rytter. On a sublinear time parallel construction of optimal binary search trees, 1994.

[3] Donald E. Knuth. Optimum binary search trees. *Acta Inf.*, 1:14–25, 1971.

[4] Michael Quinn. *Parallel Programming in C with MPI and OpenMP*. Elizabeth A. Jones, 2004.