Documentação do TP 2 de AEDS 3

Pedro Araujo Pires

13/05/2011

Introdução

O algoritmo de Kemeny-Young é um algoritmo de ranqueamento, usado para determinar o resultado de uma eleição. A votação nesta eleição é feita através de listas de preferência, onde os nomes de todos os candidatos aparecem ordenados de acordo com a preferência do eleitor.

A idéia por trás do algoritmo é achar qual é a escolha mais popular. Para isso, primeiro é montada uma matriz que compara par a par as popularidades dos candidatos. Depois, para cada permutação da ordem dos candidatos é calculada uma pontuação, somando os valores contidos na matriz. A permutação com a maior pontuação indica a configuração vencedora.

Para este trabalho o algoritmo de Kemeny-Young foi implementado utilizando técnicas de paralelização. A geração de todas as permutações, e a geração de todas as combinações de candidatos para uma dada permutação, são executadas paralelamente.

Implementação

Estruturas de dados

Para a implementação foi criada uma estrutura de dados para armazenar os dados de uma eleição. Essa estrutura é composta por três structs. A struct *Voto* armazena as informações de um voto, que são o total de pessoas que escolheram aquela ordem de candidatos, e um array com a ordem dos candidatos. A struct *Resultado* é similar à struct Voto, e armazena os dados do resultado da eleição, que são a ordem de candidatos com a maior pontuação, e a maior pontuação. Por fim, a struct *Eleicao* armazena os dados de uma eleiçao, que são:

- um array com os nomes de todos os candidatos
- uma matriz que contém as preferências dos eleitores
- o número de candidatos participando da eleição
- o número total de listas-voto
- um array de structs *Voto*, contendo todos os votos daquela eleição
- uma struct *Resultado*, para armazenar o resultado da eleição.

Também foram criadas duas structs, para armazenar os dados das permutações e das combinações geradas paralelamente. Essas structs estão definidas no arquivo enumerations.h. Ambas as structs são muito similares, armazenando os dados com as mesmas funcionalidades:

- O número de elementos no conjunto
- A cardinalidade do conjunto de permutações/combinações
- Um array (permutação) e uma matriz (combinação) com valores usados para gerar uma permutação/combinação.
- Um array para armazenar a permutação/combinação gerada.

Níveis de Paralelismo

Durante a execução do algoritmo existem duas operações muito caras: o cálculo de todas as permutações do conjunto de candidatos, e o cálculo de todas as combinações 2 a 2 dos candidatos. Para otimizar o algoritmo, essas duas operações são feitas paralelamente.

A estratégia para implementar essas paralelizações foram similares: foi utilizado um algoritmo que enumera a permutação/combinação do conjunto. Cada algoritmo recebe como entrada um número inteiro, e retorna a permutação/combinação correspondente.

Para um conjunto de três elementos, as seguintes permutações e combinações serão geradas:

Número	Permutação	Combinação
0	[0, 1, 2]	[0, 1]
1	[0, 2, 1]	[0, 2]
2	[1, 0, 2]	[0, 3]
3	[1, 2, 0]	[1, 2]
4	[2, 0, 1]	[1, 3]
5	[2, 1, 0]	[2, 3]

Esse forma de enumerar as permutações/combinações é perfeita para a paralelização, pois ela permite que cada thread execute a mesma quantidade de trabalho, evitando assim o problema de desbalanceamento de carga.

Fluxo de execução do programa

Quando o programa é executado, primeiramente ele lê o nome do arquivo de entrada passado como parâmetro, e abre o arquivo para leitura. O número de instâncias do problema é lido do arquivo, e depois os seguintes passos são executados para cada instância do problema: os dados da eleição são lidos e armazenados em uma struct *Eleicao*. Em seguida é montada a matriz com as preferências, e o resultado da eleição é calculado. O resultado da eleição é impresso na tela, e a memória alocada para a eleição é liberada.

Na hora de executar o programa, é passado pela linha de comando quants threads irão fazer o trabalho em cada nível. Como inicialmente o programa não era paralelizado, a execução serial foi mantida. Para executar algum dos níveis de forma serial, basta passar 0 como o número de threads a executar o nível. Essa opção é boa para testar a eficiência da paralelização de um nível específico.

O programa aceita um parâmetro adicional, que serve para indicar se o tempo de execução deve ser medido ou não. Caso esse parâmetro seja passado para o programa, a função gettimeofday é utilizada para medir o tempo de execução de cada eleição da entrada. Na hora de imprimir o resultado na tela, o tempo de execução é impresso logo após o resultado.

Execução

Para executar o programa, deve ser utilizado o seguinte comando:

```
./tp2 entrada.txt 2 0 bench
```

O primeiro parâmetro é o nome do arquivo que contém os dados de entrada do programa, o segundo e o terceiro parâmetros são o número de threads em cada nível de paralelização. O último parâmetro, opcional, indica se o tempo de execução de cada instância do problema deve ser marcado, e impresso na saída.

Ao se escolher 0 threads em um nível, significa que aquele nível deve ser executado de forma serial, como era no programa original.

Testes

Para testar o algoritmo, o programa foi executado variando-se o a entrada, e o número de threads. Como o número de candidatos em uma eleição é o que determina o total de permutações e combinações que serão calculadas, cada eleição possui somente uma lista de votos. A entrada utilizada para os teste foi a seguinte:

```
10
1
10
a b c
1
10
a b c d
1
10
a b c d e
```

```
1
10
a b c d e f
1
10
a b c d e f g
1
10
a b c d e f g h
1
10
a b c d e f g h i
1
10
a b c d e f g h i j
1
10
a b c d e f g h i j k
1
10
a b c d e f g h i j k l
```

As tabelas abaixo mostram os tempos de execução para os vários testes feitos. O tempo foi medido com a execução completamente serial, somente com a permutação paralelizada, somente com a combinação paralelizada, e com os dois níveis paralelizados. Foi estabelecido um limite máximo de execução de 10 minutos. Todos os testes foram feitos em um computador com um processador Intel Core 2 T7400, com dois processadores de 2.16 GHz.

Número de candidatos	Serial	2 threads	4 threads	8 threads	16 threads
3	0.000073	0.000505	0.000763	0.001348	0.002349
4	0.000034	0.000157	0.000238	0.000773	0.001435
5	0.000109	0.000196	0.000251	0.000579	0.001667
6	0.000732	0.000563	0.000582	0.000863	0.001444
7	0.006233	0.003864	0.004158	0.003882	0.004430
8	0.061892	0.042799	0.044263	0.054508	0.040576
9	0.663807	0.367295	0.351577	0.354884	0.346300
10	7.781069	4.375756	4.019479	4.046856	4.021331
11	101.423584	59.114468	54.203957	52.483597	52.473965

Tempo de execução gerando somente as permutações paralelamente.

Número de candidatos	Serial	2 threads	4 threads	8 threads	16 threads
3	0.000073	0.001123	0.001710	0.004565	0.014833
4	0.000034	0.002866	0.004618	0.018752	0.042700
5	0.000109	0.018628	0.030919	0.077892	0.193709
6	0.000732	0.100111	0.151315	0.439779	1.112055
7	0.006233	0.633984	0.997258	3.066122	7.650696
8	0.061892	5.135738	7.991803	24.723742	61.101009
9	0.663807	47.901089	73.388771	228.238983	-
10	7.781069	491.659973	-	-	-
11	101.423584	-	-	-	-

Tempo de execução gerando somente as combinações paralelamente.

Número de candidatos	Serial	2 threads	4 threads	8 threads	16 threads
3	0.000073	0.001541	0.003598	0.008549	0.020905
4	0.000034	0.002215	0.012974	0.028409	0.046652
5	0.000109	0.015282	0.039474	0.089054	0.210914
6	0.000732	0.069231	0.187151	0.501726	1.234960
7	0.006233	0.451423	1.260014	3.416110	8.697781
8	0.061892	3.623625	10.100651	27.428198	69.473404
9	0.663807	33.237671	90.119881	245.820053	-
10	7.781069	338.344696	-	-	-
11	101.423584	-	-	-	-

Tempo de execução gerando as permutações e combinações paralelamente.
Foi utilizado o mesmo número de threads para os dois níveis.

A partir dos dados acima, é possível verificar que a paralelização das permutações obteve um grande ganho de eficiência. O tempo de execução ficou próximo da metade do tempo gasto na execução serial, quando se utilizou 2 threads. Ao aumentar mais o número de threads, o tempo não sofreu alterações significativas, pois havia somente 2 processadores disponíveis para executar a tarefa.

É possível verificar que a paralelização das combinações resultou em uma perda de eficiência, ao invés de ganho. Isso se deve ao fato de, a cada permutação gerada, são criadas as threads para fazerem o cálculo das combinações. Isso gera um overhead, que gasta muito mais tempo do que a geração das combinações em si.

Além desses testes, foi utilizado o programa *valgrind* para verificar o gerenciamento de memória do programa. A saída do *valgrind* foi:

```
LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 936 bytes in 5 blocks
```

Devido ao pouco tempo para a implementação do trabalho, e à complexidade do código, não foi possível achar os 5 blocos que não foram devidamente desalocados.

Análise de complexidade

O algoritmo possui várias fases, onde cada uma possui uma complexidade diferente. Na parte da montagem da matriz, são feitas (n * (n-1))/2 atribuições para cada lista-voto, onde n é o número de candidatos participantes da eleição. Portanto essa parte possui complexidade $O(n^2)$.

Depois que a matriz foi montada, um método recursivo calcula todas as permutações dos candidatos, e calcula a pontuação de cada configuração. Esse método recursivo consiste em trocar os elementos de um arranjo gerando assim as permutações. A seguinte equação de recorrência pode ser obtida, onde n é o número de candidatos:

$T(1) = 2$
$T(2) = 4$
$T(3) = 12$
…
$T(n) = n * T(n-1) = n!$

Logo, a função recursiva possui compexidade O(n!).

Como consideramos somente a função mais demorada na hora de calcular a complexidade, o algoritmo é O(n!),

onde n é o número de candidatos.

Decisões de implementação

Como não é sabido *a priori* o número total de instâncias do problema, foi definido que, para cada instância do problema, seria alocado o espaço necessário para armazenar os dados da eleição, o resultado seria calculado, e o espaço alocado seria liberado. Dessa forma, o total de memória usada pelo programa em qualquer momento de sua execução é o espaço necessário para armazenar os dados de uma únca eleição.

Como o programa que fazia o cálculo serial do resultado já estava pronto, ficou definido que essa funcionalidade não seria retirada do programa paralelizado. Dessa forma seria possível fazer testes comparativos entre a execução serial e a execução paralela, assim como a eficiência de cada nível de paralelização.

Para algumas configurações de eleições, dois ou mais resultados irão obter a mesma pontuação. Foi definido que o último resultado vencedor obtido será considerado o vencedor da eleição, independente de quantos resultados obtidos antoriormente possuam a mesma pontuação.