Um estudo computacional de dois algoritmos de programação dinâmica com utilização eficiente de cache

Guilherme S. Ribeiro[1], Marcus Ritt[1], Luciana S. Buriol[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{gsribeiro, mrpritt, buriol}@inf.ufrgs.br

Resumo. *Com a maior velocidade de evolução dos processadores em relação às memórias, o desempenho de programas passou a ser muito influenciado pela maneira como eles acessam os dados que necessitam. Para otimizar o acesso à memória foi criada uma hierarquia, que começa com a cache, passa pela memória RAM e termina com as memórias não-voláteis. Algoritmos cache-oblivious minimizam a quantidade de vezes que os programas precisam buscar dados na memória RAM lenta, ao invés de buscar na cache, que é mais rápida. Este artigo tem como objetivo analisar o desempenho de algoritmos cache-oblivious de resolução do problema da maior subsequência comum e do problema do gap, ambos muito utilizados na bioinformática.*

1. Introdução

Durante as últimas décadas os avanços da informática foram expressivos no desenvolvimento de microprocessadores, no entanto as memórias não evoluíram tão rápido. Para diminuir a diferença de velocidade entre esses dois componentes foi desenvolvida uma hierarquia de memória onde o processador procura os dados que necessita primeiro numa memória pequena, rápida e cara (a cache) e caso não a encontre ele procura numa memória grande, lenta e barata (a RAM).

Esses diferentes níveis de hierarquia de memória nas arquiteturas de computadores influem no desempenho dos algoritmos executados. O acesso à cache de um processador, por exemplo, é 100 vezes mais rápido que o acesso à memória principal [Hennessy e Patterson 2003]. Por isso, um algoritmo que exibe localidade no acesso aos seus dados, executa mais rápido. A literatura relata uma aceleração da execução até um fator 10 [Chowdhury 2005, Chowdhury e Ramachandran 2006].

Portanto, se desenvolvermos algoritmos que minimizem a necessidade do processador de procurar dados na memória RAM iremos acelerar o seu processamento. Os algoritmos desenvolvidos com esse propósito são conhecidos como algoritmos cache-eficientes. Essa é uma área recente e muito promissora de projeto e análise de algoritmos.

Para realizar a análise de complexidade de entrada/saída de algoritmos foi criado um modelo de cache ideal. Nesse modelo existem dois níveis, a memória cache, finita, com tamanho M, organizada em Z palavras e um tamanho de linha de L palavras, e a memória principal, infinita, ambos com acesso aleatório de tempo constante. Ao buscar um dado na memória, o processador acessa primeiramente a cache, se esse dado não se encontra disponível, ocorre um *cache-miss* e um bloco é buscado da memória principal para a cache. A análise de complexidade de algoritmos assim leva em conta tanto o número de operações quanto o número de cache misses (I/Os).

2. Algoritmos cache-eficientes estudados

Foram implementados dois algoritmos cache-oblivious propostos em [Chowdhury e Ramachandran 2006] e posteriormente foram analisados os seus desempenhos comparando com os algoritmos tradicionais. O primeiro foi o algoritmo que resolve o problema da maior subsequência comum e o segundo foi o algoritmo que resolve o problema do gap, ambos explicados a seguir.

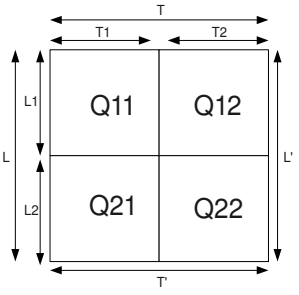
No primeiro problema, dadas duas sequências $X = (x_1, ..., x_n)$ e $Y = (y_1, ..., y_n)$, nós queremos encontrar a maior subsequência $Z = (z_1, ..., z_k)$ comum a X e a Y. O algoritmo tradicional para resolver esse problema é o algoritmo de Hirschberg. Em [Chowdhury e Ramachandran 2006] foi proposto um algoritmo cache-oblivious para resolver esse problema da maior subsequência comum que divide a tabela de programação dinâmica em 4 quadrantes e depois recursivamente resolve cada uma dessas instâncias dividindo-as em 4 novamente até que se alcance o caso base desejado. Dessa forma obtemos o tamanho da maior subsequência através da chamada propaga_valores(0, 0, n, n, vet(0), vet(0), A, B) do algoritmo descrito abaixo. Posteriormente aplicamos o mesmo princípio da divisão em quadrantes para obter a subsequência em sí, rastreando o seu caminho do quadrante inferior a direita até o quadrante superior a esquerda recursivamente.

Ramachandran e Chowdhury (2006) apresentam uma análise da complexidade dos algoritmos para o problema da LCS e constata-se que o algoritmo tradicional possui complexidade de I/O de $\Omega(mn/L)$ e que o algoritmo cache-oblivious descrito abaixo possui complexidade de I/O de $\Omega((mn)/(ZL))$, onde m e n são os tamanhos das duas sequências a serem comparadas. No entanto, ambos possuem a mesma complexidade de tempo $\Theta(mn)$.

Algoritmo cache-oblivious para o problema da LCS sem recuperação da maior subsequência

Entrada. (ib, jb) índices do ponto superior esquerdo do quadrante a ser processado
(ie, je) índices do ponto inferior direito do quadrante a ser processado
vetor L com os valores da coluna esquerda do quadrante
vetor T com os valores da linha superior do quadrante
strings A e B a serem comparadas
Saida. vetor T' com os valores da linha inferior do quadrante
vetor L' com o valores da coluna direita do quadrante

```
propaga_valores(int ib, int ie, int jb, int je, vetor L, vetor T, string A, string B) {
   if (ib + base >= ie) {
      lcs_linspace(ib, ie, jb, je, L, T, A, B);    // processa o caso base da recursão
   } else {
      int im = (ie + ib) / 2;    // divide o quadrante em 4 outros quadrantes
      int jm = (je + jb) / 2;
      propaga_valores(ib, im, jb, jm, L1, T1, A1, B1);    // processa Q11
      propaga_valores(ib, im, jm, je, L1, T2, A1, B2);    // processa Q12
      propaga_valores(im, ie, jb, jm, L2, T1, A2, B1);    // processa Q21
      propaga_valores(im, ie, jm, je, L2, T2, A2, B2);    // processa Q22
   }
}
```



Dadas duas sequências X e Y, o problema do gap consiste em encontrar o custo mínimo necessário para transformar X em Y com uma série de inserções, deleções e substituições de elementos com um custo associado a cada operação e a cada elemento. O problema e variações são conhecidos também pelo nome *edit distance* ou distância de Levenshtein. O algoritmo tradicional de programação dinâmica que resolve esse problema calcula linearmente a matriz de possibilidades. Já o algoritmo cache-oblivious proposto em [Chowdhury e Ramachandran 2006] realiza um processo semelhante ao

problema da maior subsequência comum. Ele divide o problema em 4 quadrantes e os resolve recursivamente dividindo cada um deles em 4 novamente até alcançar o caso base desejado.

Ambas as abordagens para o problema do gap possuem complexidade de tempo $\Theta(n^3)$ e utilizam $\Theta(n^2)$ espaço. No entanto, o algoritmo tradicional possui complexidade de I/O de $\Theta(n^3/L)$ enquanto o algoritmo cache-oblivious possui complexidade de I/O de $\Theta(n^3/(L\sqrt{M}))$.

3. Ambiente de testes

As implementações foram desenvolvidas na linguagem C++, compiladas com g++ e com os parâmetros de otimização: -O3 -mtune=core2. Elas foram testadas em um computador Intel Core 2 Quad Q6600 com 2.4 GHz, 8 MB de cache L2, 4 GB de memória RAM e com Sistema Operacional Ubuntu 8.10 – 64 bits.

Para a contagem da quantidade de cache misses dos algoritmos foram usados dois programas, o Valgrind e o PerfSuite. O primeiro é um *framework* de instrumentação de código que disponibiliza uma ferramenta chama CacheGrind que emula uma memória cache e faz a contagem das cache misses do algoritmo. Essa ferramenta é útil para casos de teste que tenham pouco tempo de execução pois a emulação aumentou o tempo de execução aproximadamente um fator 10, no entanto a contagem é livre de interferências de outros processos e problemas de arquitetura. Já o segundo é uma coleção de ferramentas para medidas de desempenho de softwares. Ele tem a vantagem de não acrescentar muito mais tempo na execução do algoritmo pois ele utiliza registradores do processador para acessar a quantidade de cache misses de um programa e a sua contagem mostra valores práticos em arquiteturas reais.

4. Resultados

Nessa seção apresentaremos os resultados dos experimentos computacionais dos algoritmos tradicionais e cache-oblivious. Nesses testes utilizamos sequências de mesmo tamanho (n) geradas aleatoriamente a partir do alfabeto.

Tabela 1. Desempenho do algoritmo cache-oblivious do problema gap.

n	tamanho do ponto de parada	tempo total (segundos)	no. de cache misses (PerfSuite)
512	1	0,34	406
	16	0,31	817
	64	0,33	637
	512	0,33	4.434
1024	1	3,91	210.330
	16	3,88	121.793
	64	3,88	105.729
	512	3,86	110.145
2048	1	34,58	2.634.999
	16	34,49	2.958.302
	64	34,53	3.632.968
	512	35,63	43.664.021

Tabela 2. Desempenho do algoritmo de resolução do problema gap que utiliza programação dinâmica tradicional.

n	tempo total (segundos)	no. de cache misses (PerfSuite)
512	0,36	8.279
1024	4,62	3.318.533
2048	43,56	230.977.070

Tabela 3. Desempenho do algoritmo cache-oblivious para o problema da LCS (sem recuperação da maior subsequência).

n	tamanho do ponto de parada	tempo (segundos)	no. de cache misses (Valgrind)
16.384	1	7,13	8.732
	16	1,26	8.740
	64	1,08	8.740
	512	1,03	8.740
32.768	1	28,76	11.297
	16	5,05	11.304
	64	4,34	11.304
	512	4,08	11.304
65.536	1	111,89	16.416
	16	20,14	16.423
	64	17,21	16.424
	512	16,22	16.424
131.072	1	448,72	26.634
	16	80,66	26.641
	64	68,75	28.694
	512	64,88	27.288

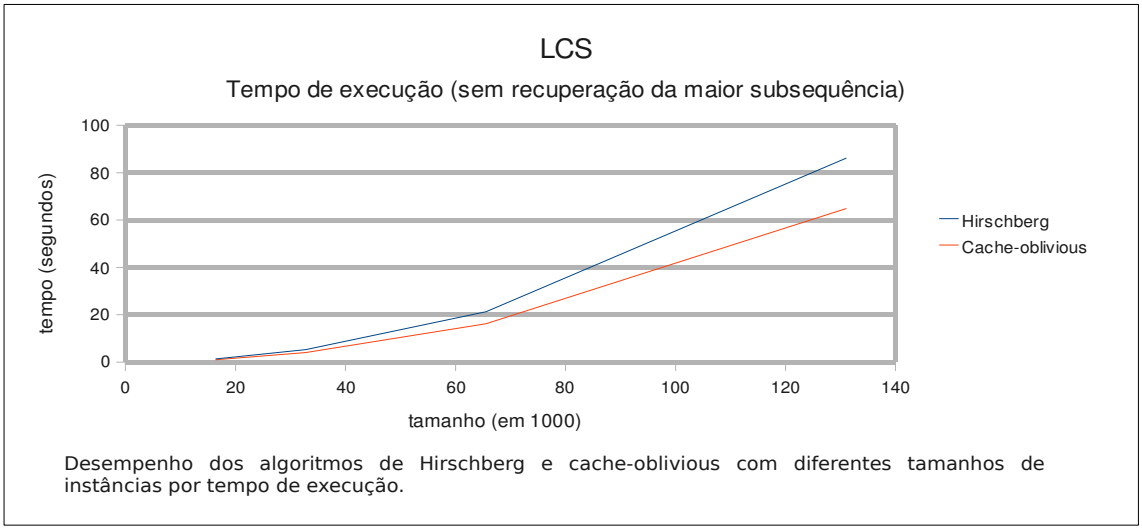
O algoritmo cache-oblivious para o problema LCS age dividindo recursivamente o problema em quadrantes até chegar ao caso base (ponto de parada), onde resolve o problema de forma tradicional. Variando o tamanho desse caso base podemos notar na tabela acima que o tempo de execução varia consideravelmente. Isso ocorre porque a cada chamada recursiva existe um custo de divisão do problema e de consistência dos vetores de dados, no entanto esse benefício de utilizar casos base maiores perde desempenho quando essa instância não cabe completamente na cache e porque a resolução tradicional é eficiente apenas em casos pequenos.

Tabela 4. Desempenho do algoritmo de Hirschberg para o problema da LCS (sem recuperação da maior subsequência).

n	tempo (segundos)	no. de cache misses (Valgrind)
16.384	1,37	11.317
32.768	5,35	14.398
65.536	21,33	20.541
131.072	86,11	32.824

Tabela 5. Comparação dos algoritmos de Hirschberg e cache-oblivious (parada em 512) para o problema da LCS (sem recuperação da maior subsequência) com aceleração do cache-oblivious em relação ao de Hirschberg.

n	tempo (min)		aceleração
	Hirschberg	cache-oblivious	
524.288	27,02	12,49	2,16
1.048.576	91,44	51,21	1,79
2.097.152	440,55	205,10	2,15



Desempenho dos algoritmos de Hirschberg e cache-oblivious com diferentes tamanhos de instâncias por tempo de execução.

Tabela 6. Desempenho do algoritmo de Hirschberg para o problema da LCS (com recuperação da maior subsequência comum).

n	tempo (segundos)	no. de cache misses (PerfSuite)
8.192	4,49	5.929
16.384	17,97	23.085
32.768	71,19	66.601
65.536	283,92	371.208
131.072	1357,06	3.009.978

Tabela 7. Desempenho do algoritmo cache-oblivious (parada em 1) para o problema da LCS (com recuperação da maior subsequência comum).

n	tempo (segundos)	no. de cache misses (PerfSuite)
8.192	5,47	10.449
16.384	22,19	23.588
32.768	84,34	107.051
65.536	337,59	198.836
131.072	1336,44	1.916.257

Nas tabelas 6 e 7 acima podemos notar que para casos pequenos o algoritmo de Hirschberg possui menos cache misses e executa em menos tempo que o algoritmo cache-oblivious. Isso ocorre porque o segundo algoritmo possui maior complexidade

associada à divisão em quadrantes e à união dos resultados dos casos base. Dessa forma, o algoritmo cache-oblivious passa a ser mais eficiente para casos grandes, com n igual a 65536 ele passa a ter um número menor de cache-misses e com n igual a 131072 o seu tempo de execução passa a ser mais vantajoso. Dessa forma, esses resultados parciais demonstram que para instâncias maiores a versão cache-oblivious é ligeiramente mais rápida. Comparando com a versão sem recuperação, podemos esperar uma aceleração significativa para bases maiores, como podemos notar no gráfico acima.

5. Conclusões

A implementação de algoritmos de programação dinâmica com divisão e conquista favorece a localidade espacial e temporal dos acessos à memória. Isso diminui significativamente a quantidade de cache misses do algoritmo, aumentando a sua velocidade, como podemos ver nas tabelas e gráficos de testes acima.

O algoritmo cache-oblivious para o problema do gap obteve até 60 vezes menos cache misses do que o tradicional e foi até 25% mais rápido. A diferença de cache misses não acelerou com a mesma proporção o tempo de execução devido às características do algoritmo, que ocupa muito espaço em memória e por isso não pode ser testado para instâncias maiores onde a diferença de cache misses seria mais sensível à velocidade de execução.

O algoritmo cache-oblivious para o problema da maior subsequência comum obteve até 30% menos cache misses do que o algoritmo de Hirschberg nos casos menores, em que foi possível medir esse número. Em casos de teste grandes, foi até 120% mais rápido, porque quanto maior o tempo de execução da instância, maior é a influência da localidade dos dados no seu desempenho.

6. Trabalhos futuros

Na próxima etapa do projeto de algoritmos de programação dinâmica com utilização eficiente da cache pretendemos realizar medidas com tamanhos maiores de parada da recursão nas implementações cache-oblivious e tradicional para os problemas da maior subsequência comum e do gap. Posteriormente iremos projetar e analisar soluções cache eficientes para outros problemas relevantes à bioinformática.

Referências

Chowdhury, R. A. (2005) “Experimental evaluation of an efficient cache-oblivious LCS algorithm”, In Technical Report TR-05-43, UTCS.

Chowdhury, R. A. and Ramachandran, V. (2006) “Cache-oblivious dynamic programing”, In SODA ’06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, pages 591–600, New York, NY, USA, ACM.

Frigo, W., Leiserson, C. E., Prokop, H. and Ramachandran, V. (1999) “Cache-oblivious algorithms”, In Proc. 40th IEE Sympos. Found. Comp. Sci., pages 285–297.

Prokop, H. (1999) “Cache-oblivious algorithms”, PhD thesis, MIT.

Hennessy, J. L. and Patterson, D. A. (2003) “Computer architecture: A quantitative approach”, In Morgan Kauffmann Publishers Inc..