Análises Experimentais de uma Implementação Eficiente de um Algoritmo Paralelo para a Geração de Arranjos de Sufixos *

M.A.P. Cristo
J.P. Kitajima
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Caixa Postal 702
30123-970 Belo Horizonte - MG
{marco, kitajima}@dcc.ufmg.br

Resumo - Este artigo apresenta experimentos feitos com uma implementação de um algoritmo paralelo para a geração de Arranjos de Sufixos. O algoritmo analisado é uma variação de um algoritmo baseado no *hyperquicksort*. Os experimentos mostram que pequenas modificações na idéia originalmente apresentada conduzem a uma versão bastante eficiente que permite, por exemplo, com 24 processadores, a geração de arranjos para textos de 400 MB em menos de dez minutos, ou seja, trinta e seis vezes mais rápido que a versão seqüencial.

Abstract - This paper presents experiments performed with an implementation of a parallel algorithm for generation of suffix arrays. This algorithm is a variation of a hyperquicksort based algorithm. The experiments show that small changes on the original algorithm leads to an efficient version that allows, by instance, with 24 processors, the generation of a suffix array for a 400 MB text in less than ten minutes, that is, thirty-six times faster than the sequential version.

Palavras chave - Processamento Paralelo, Geração de Índices, Arranjos de Sufixos, Passagem de Mensagens.

1 Introdução

A necessidade de se fazer buscas na Internet exige um grande esforço de coleta, indexação e localização de palavras nos textos que constituem as *homepages*. Para tais aplicações a criação de índices é uma operação crítica. Em [6], [7] e [12] são apresentados algoritmos paralelos para a construção das duas principais estruturas de indexação de textos, as listas invertidas e os arranjos de sufixos. Estes algoritmos foram propostos no contexto de redes de estações de trabalho [1]. Neste artigo apresentamos uma versão do algoritmo proposto em [12] e experimentos feitos com uma implementação deste novo algoritmo.

*Os experimentos apresentados neste trabalho foram feitos usando os computadores IBM SP disponíveis nos laboratóris LMC/IMAG, Grenoble, França e CENAPAD/MG, Belo Horizonte, Brasil.

2 Trabalhos Relacionados

O arranjo de sufixos é uma sofisticada estrutura de indexação que permite consultas complexas por palavras e frases ocupando espaço próximo ao tamanho do texto indexado [3]. Esta estrutura foi inicialmente proposta por [4] e [9] e consiste de uma lista de ponteiros para todos os sufixos no texto em ordem lexicográfica dos respectivos sufixos. A construção do arranjo de sufixos é, então, uma ordenação indireta dos ponteiros, cuja complexidade é da ordem de $O(n \log n)$ onde n é o tamanho do texto. A grande dificuldade associada a esta estrutura ocorre quando o texto a ser indexado é maior que a memória disponível e é necessário utilizar memória secundária. Para este caso, o melhor algoritmo seqüencial conhecido para gerar arranjos de sufixos é da ordem de $O(n^2 \log \frac{n}{m})$, onde m é o tamanho da memória primária [3].

Na literatura encontramos dois algoritmos paralelos para a construção de arranjos de sufixos. Ambos partem da idéia de utilizar a memória agregada de uma máquina paralela para evitar o uso da memória secundária. O primeiro, baseado em *merge-sort*, é apresentado por [6]. O algoritmo mostrou pobre escalabilidade, o que levou [12] a propor o segundo algoritmo, este baseado em *hyperquicksort*. A complexidade deste algoritmo é da ordem de $O(\frac{n}{r} \log n) + O(\frac{n}{r})$, onde n é o tamanho do texto e r é o número de processadores. Em [8] são apresentados experimentos feitos com uma implementação deste algoritmo. Seus resultados mostram que, na prática, à medida que se aumenta o número de processadores, o algoritmo precisa utilizar uma quantidade cada vez maior de arquivos durante a fase final de intercalação. Assim, o desempenho cai em função do aumento na quantidade de *seeks*.

3 O Algoritmo Paralelo Proposto

Como visto na seção anterior, o principal problema observado por [8] no algoritmo de [12] (que a partir de agora chamamos de RZK) foi a necessidade de operações de I/O nas fases finais. Para contornar este problema, propomos um novo algoritmo onde as porções de texto trocadas pelo RZK são suficientemente pequenas de forma a caber na memória primária. Para conseguir isto, em nosso algoritmo (que a partir de agora chamamos de RZK II) os processadores colaboram para gerar uma escala a ser usada na fase final. Esta fase é constituída por tantas intercalações quanto for o número de rodadas na escala.

Para gerar esta escala todos os p processadores trocam entre si as porções comprimidas dos arranjos locais reparticionados em função dos percentis globais. Ao contrário do algoritmo RZK, os blocos trocados não incluem ponteiros mas somente sufixos de l caracteres. Cada processador intercala os blocos recebidos e então verifica quantos sufixos devem ser solicitados dos demais processadores de acordo com o montante de memória livre na fase final. O valor de l deve ser escolhido de forma que o maior bloco de sufixos iguais de l caracteres seja menor que a memória local e os sufixos sejam tão pequenos quanto possível para minimizar a informação a ser transmitida na rede e alcançar o máximo de compressão por eliminação de sufixos repetidos.

O fato das palavras em textos em linguagem natural apresentarem distribuição

de Zipf [2] nos garante que o sufixo mais freqüente de l caracteres deve ter baixa freqüência em relação ao texto como um todo. Essa freqüência é menor à medida que é maior o valor de l. Isso também implica que há mais repetições de sufixos com valores menores de l. Na implementação usada nos experimentos deste artigo usamos l igual a cinco caracteres. Na prática a fase de criação da escala tem um custo pequeno em relação às outras fases do algoritmo e a escala gerada proporciona um bom balanceamento de carga entre os processadores. A complexidade apresentada por RZK II é da ordem de $O(\frac{n}{r}\log n) + O(\frac{n}{r})$.

4 Análise Experimental

Os experimentos aqui apresentados foram realizados no CENAPAD/MG no Brasil e no LMC na França. Ambos os computadores são IBM SP, o primeiro com 40 e o segundo com 48 nós. No CENAPAD, os processadores são interconectados por um *switch* de 155 Mbps de largura de banda por processador e compartilham um sistema de disco de 160 MB/s. No LMC o *switch* suporta 40 Mbps e cada nó é provido de um disco de 40 MB/s. Para evitar que as medidas de tempo fossem influenciadas pelo sistema de gerência de memória virtual, em nenhum experimento foi utilizado mais que 70% da memória disponível. A implementação do RZK II foi feita em ANSI C usando a biblioteca de passagem de mensagens MPI [11]. Os textos usados nos experimentos foram extraídos do *Wall Street Journal* da coleção TREC-3 [5]. Em todos os experimentos, o texto usado não sofreu qualquer tipo de filtragem nem eliminação de *stopwords* [10].

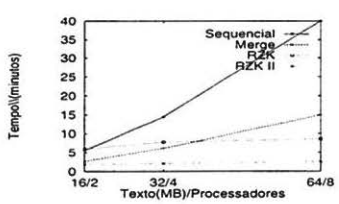

Figura 1: Comparação entre algoritmos para Geração de Arranjos de Sufixos

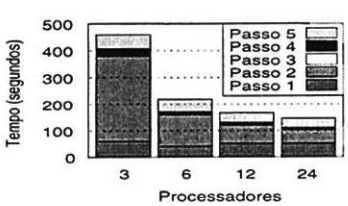

Figura 2: Caracterização do Tempo de Criação do arranjo para texto de 100 MB

Na figura 1 podemos observar uma comparação entre os vários algoritmos para a criação de arranjos de sufixos em experimento feito no LMC. O RZK II mostra boa escalabilidade e o melhor desempenho entre todos os algoritmos apresentados. Em experimento similar feito no CENAPAD com 24 processadores, o RZK II gerou um arranjo de sufixos para um texto de 400 MB em menos de 10 minutos, ou seja, trinta e seis vezes mais rápido que o algoritmo seqüencial que precisou de seis horas. Apesar disso, foi observada uma queda na eficiência do *speedup* à medida que se aumentou o número de processadores.

A figura 5 apresenta a caracterização do tempo gasto por RZK II para gerar arranjos de sufixos. Nesta figura o passo 1 corresponde à leitura e distribuição do texto a ser indexado, o passo 2 corresponde à criação dos arranjos locais, o passo 3 corresponde à fase em que os processadores colaboram para decidir como serão divididos os arranjos locais, o passo 4 corresponde à criação da escala e, finalmente, o passo 5 corresponde à fase final de intercalação. Como pode ser observado, a queda no tempo total de criação do arranjo se deve quase exclusivamente à queda no tempo de criação dos arranjos locais (passo 2). Também podemos observar que o tempo gasto com a criação da escala (passo 4) é pouco relevante em relação ao tempo total. O passo 1 consome mais tempo à medida que se acrescentam mais processadores em função da necessidade maior de comunicação. O mesmo ocorre com o passo 5, pois a filosofia de particionamento de mensagems do *quicksort* num sistema de passagem de mensagens faz com que o total de mensagens trocadas cresça à medida que se aumenta o número de processadores (com 3 processadores, 2/3 dos sufixos são comunicados; com 6, 5/6 dos sufixos são comunicados). Na prática, o ganho de memória nessa fase compensa parcialmente a maior carga de comunicação mas não é capaz de fazer o tempo cair. Isto também explica a queda na eficiência do *speedup* alcançado, pois ao dobrarmos o tamanho do texto e o número de processadores, o custo de comunicação deve aumentar mais que duas vezes.

5 Conclusões

Neste artigo mostramos que com pequenas variações no algoritmo RZK é possível conseguir um novo algoritmo que apresenta boa escalabilidade e bom desempenho. Atualmente nós estamos em processo de validação de um modelo analítico para o algoritmo proposto. Com este modelo esperamos determinar o comportamento do algoritmo diante de novos cenários sem as limitações impostas pelos ambientes de experimentação que nós temos usado. Nós devemos continuar os nossos estudos em três direções. Primeiro, pretendemos avaliar novas otimizações que possam tornar o algoritmo mais eficiente. Segundo, vamos modificar o algoritmo para operar em disco e lidar com situações onde a memória agregada é menor que o texto a indexar. Por último, verificaremos que modificações devem ser feitas para que o nosso algoritmo seja capaz de lidar com cadeias de DNA.

Referências

[1] T. Anderson, D. Culler, and D. Paterson. A case for NOW (Network of Workstations). *IEEE Micro*, 15(1):54-64, February 1995.

[2] M. D. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In R. Baeza-Yates, editor, *Proceedings Fourth South American Workshop on String Processing*. Carleton University Press International Informatics Series, Valapraiso, Chile, 1997.

[3] G. H. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. Baeza-Yates (Eds.), *Information Retrieval: Data Structures and Algorithms*, 66-82, Prentice-Hall, Englewoods Cliff, N.J., 1992.

[4] G. Gonnet. PAT 3.1: An Efficient Text Search System - User's Manual. Centre of the New Oxford English Dictionary, University of Waterloo, Canada, 1987.

[5] D. K. Harman. Overview of the Third Text REtrieval Conference. *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1-19, National Institute of Standards and Technology Special Publication 500-207, Gaithersburg, Maryland, 1995.

[6] J. P. Kitajima, M. D. Resende, B. Ribeiro-Neto, and N. Ziviani. Distributed parallel generation of indices for very large text databases. *Proceedings of the 1997 3rd International Conference on Algorithms and Architectures for Parallel Processing - ICA3PP*. World Scientific. A. Goscinski and M. Hobbs and W. Zhou. Melbourne, Australia. dec, 1997.

[7] J. P. W. Kitajima, B. A. N. Ribeiro, G. Navarro, and N. Ziviani. Parallel generation of inverted lists on a network of workstations Universidade Federal de Minas Gerais - Departamento de Ciência da Computação. RT 009-97. Belo Horizonte, Brasil. 1997.

[8] A. Macedo *et alli*. Experimental Analysis of a Parallel Quicksort-Based Algoritm for Suffix Array Generation. *3rd International Meeting on Vector and Parallel Processing*, Porto, Portugal, 1998.

[9] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searchs. *1st ACM-SIAM Symposium on Discrete Algorithms*, 319-327, San Francisco, 1990.

[10] G. A. Miller, E. B. Newman, and E. A. Friedman. Length-frequency statistics for written English. In *Information and Control*, 1:370-380, 1958.

[11] IBM. *IBM Parallel Environment for AIX: MPI Programming and Subroutine*. Version 2, Release 3, Doc Number GC23-3894-02. 1997.

[12] G. Navarro, J. P. W. Kitajima, B. A. N. Ribeiro, and N. Ziviani. Distributed parallel generation of suffix arrays. A. Apostolico and J. Hein, editors, *Lecture notes in Computer Science*, volume 1264, pages 102-115, Aarthus, Denmark, June 1997. Springer-Verlag. Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM).