UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

SSC0143
Programação Concorrente - Turma B

Projeto da Disciplina

Professor Dr. Julio Estrella

Grupo 06:	*Números USP:*
Bruno Junqueira Adami	6878762
Lucas Junqueira Adami	6792496
Lucas Lobosque	6792645

27 de junho de 2012

1 Introdução

Este projeto consiste em desenvolver uma aplicação paralela que gere palavras aleatóriamente. A cada palavra gerada, deve-se verificar se ela está no dicionário de entrada. Caso ela esteja, ela deve ser removida. A aplicação acaba quando todas as palavras são geradas. Há várias maneiras de se interpretar o que foi proposto. Portanto, a interpretação que foi adotada será explicada neste documento.

O projeto consiste de três módulos: a filtragem do dicionário, a geração das palavras aleatórias e a busca e remoção no dicionário. Palavras inteiras de tamanho 5 são geradas e depois é efetuada uma verificação da existência dela no dicionário. Caso ela exista, é removida e impressa na tela. Para palavras de tamanho menor do que 5, são comparadas as primeiras letras. Como podem existir vários prefixos de palavras iguais (Ex: abcde, abc), é escolhida a palavra de tamanho maior primeiro. Caso haja empate, a primeira que foi carregada do dicionário será escolhida. As palavras de tamanho maior do que 5 são divididas em várias palavras de tamanho 5 onde, depois de gerada uma destas partes, é verificado se todas as outras partes desta palavra composta foram geradas. Caso positivo, ela é mostrada na tela. Essa decomposição está melhor explicada nas próximas seções.

2 Filtragem do Dicionário

O dicionário de entrada fornecido no site da disciplina foi filtrado. Um programa serial que carrega as palavras eliminando as letras inválidas (vírgulas, espaços e outros símbolos) foi desenvolvido. Além disso, foi efetuada uma identificação e eliminação de palavras repetidas. Para desenvolver tal programa, uma árvore de prefixos foi implementada, conhecida como *trie*. Ela é uma árvore onde cada ramo representa uma letra de ramificação da palavra. A complexidade de inserção e busca nesta estrutura de dados é de $O(n)$, onde n é o tamanho da *string* a ser inserida. A complexidade é linear ($O(n)$ onde n é o tamanho de todas as palavras juntas) na leitura do arquivo e na manipulação da *trie*. A complexidade em memória também é de $O(n)$, onde n é o tamanho de todas as palavras juntas, pois cada nó aloca dinâmicamente os ramos da árvore.

No final da filtragem, há a produção de um arquivo com as palavras do diconário com letras de a-z apenas, uma em cada linha. No total, o arquivo tem 65813 palavras e um tamanho de $616KB$.

3 Separação do Problema em Vários *Hosts*

Um dos requisitos do projeto é o de usar os *hosts* do *cluster* da disciplina para fazer o processamento do problema. Para isso é utilizado a biblioteca *MPI*. Duas abordagens possíveis foram estudadas. A primeira é de alguma forma compartilhar todo o dicionário para todos os nós e sincronizar a remoção de palavras geradas do mesmo. Isso causa um enorme tráfego na rede e esta abordagem foi descartada. A segunda, adotada no projeto, é separar o dicionário em cada *host*. As palavras compostas devem ficar nos mesmos *hosts* para evitar a comunicação entre os nós, aumentando a eficiência da abordagem. Na separação do dicionário não há nenhum *speedup* (negativo ou positivo), pois como as palavras

são geradas aleatóriamente (uniformemente), a probabilidade a cada ciclo de execução de uma palavra ser removida do dicionário é a mesma.

Suponha que há M palavras todas de tamanho 5 no dicionário. Suponha que a probabilidade de cada palavra sair seja $1/26^5$. A probabilidade em um passo de gerar alguma palavra do dicionário é de $M/26^5$. Se o dicionário for separado entre três nós, A palavras para o primeiro, B para o segundo e C para o terceiro, a esperança (média) de palavras geradas é de $A/26^5 + B/26^5 + C/26^5$, ou $(A + B + C)/26^5$. Como $A + B + C = M$, a probabilidade para três nós é a mesma probabilidade para um nó.

Ao incluir palavras de tamanho menor, a demonstração fica um pouco mais complexa, mas o resultado é o mesmo. Na prática foi testado se essa suposição matemática esá correta, e foi comprovado que o *speedup* é de 1. Também foi testada a mesma teoria no próprio nó e o mesmo foi observado. Portanto, o speedup do problema deve ser focado em cada nó. Ou seja, onde é separado o problema no próprio *host*. Ao alcançar um *speedup* de X em cada nó, o *speedup* total será de $X * Y + T$ onde Y é a quantidade de nós e T o tempo inicial para separar as palavras em cada *host*.

4 Geração das Palavras Aleatórias

Do primeiro trabalho, na geração de números aleatórios para fazer o método de *Monte Carlo*, já era sabido que a função *rand* da *stdlib* no *C*, na implementação da *libC* no *Linux*, usa de travamentos de contexto para gerar o número aleatório. Então, um novo gerador de números aleatórios foi criado. Um gerador congruente linear foi desenvolvido, com os parâmetros personalizados para o problema:

$$M = 11881376$$
$$\text{são } 26^5 \text{ palavras possíveis de tamanho 5}$$
$$A = 53$$
$$C = 3$$

Os parâmetros A e C devem obedecer as restrições dadas em [Random]. Também, um programa para testar se realmente o gerador é uniforme foi criado. Se a geração ocorrer letra a letra, a probabilidade de se gerar uma palavra de tamanho 5 qualquer é de $1/26^5$. Ao invés de gerar letra a letra, a palavra foi representada como um número de tamanho 5 na base 26 (0-11881376). São 26^5 números e cada um tem a mesma chance de ser escolhido (geração uniforme). Assim a probabilidade é de $1/26^5$, a mesma na geração letra a letra.

5 Processamento do Problema em Cada *Host*

Agora há em cada nó o mesmo problema. Dado o dicionário, gerar palavras de tamanho 5 aleatoriamente e remover as achadas até que não exista mais palavras no dicionário. Inicialmente, a *trie* foi adotada para representar o dicionário, pois foi utilizada na filtragem do dicionário e ela é uma estrutura simples e eficiente. Então, de algum modo é preciso paralelizar as funções de busca e

remoção nesta estrutura.

Para paralelizar o problema em cada nó a *Pthreads* foi utilizada, pois esta *API* fornece uma programação bem mais baixo nível em relação a *OpenMP*. Isto não significa que o problema não pode ser paralelizado usando *OpenMP*. A adoção dessa *API* foi uma escolha de projeto.

Como a *trie* é uma árvore de prefixos, o modo mais intuitivo de fazer estas funções é com buscas em profundidade. Porém, na criação de mais de uma unidade de processamento para fazer a busca, é necessário garantir que os nós estejam sincronizados. Na prática, foi preciso inserir vários trancamentos de contextos (*mutex locks*) e o resultado não foi satisfatório. Também, uma abordagem de busca em largura nesta árvore foi pensada. Mas, antes de pô-la em prática foi decidido que a árvore não era realmente a estrutura mais fácil e eficiente disponível para resolver o problema.

Então, foi discutido que a solução ótima é a utilização de *hashing*. O número aleatório de tamanho 5 na base 26 gerado pelo gerador linear congruente gera uma *hash* para a palavra, tornando o problema muito simples:

$$funcaoHash(str) = 26^0 * (str[0] -' a') + 26^1 * (str[1] -' a') + 26^2 * (str[2] -' a') \\ +26^3 * (str[3] -' a') + 26^4 * (str[4] -' a')$$

Cada thread gera uma *hash* que representa uma palavra de tamanho 5. Se essa *hash* existe no dicionário então é retirada da estrutura. Observe que não haverá colisões neste hash. O dicionário é do que um vetor de inteiros de tamanho 26^5, onde cada posição indexa o *hash* da *string* e quantas delas existem. Em torno de $40MB$ de memória é utilizado, muito maior do que a *trie*. Porém, a complexidade de busca/remoção na execução é de $O(1)$. Para sincronizar as *threads* que geram essas *hashes* os travamentos de contexto foram utilizados e o resultado foi de um emphspeedup de 4 (testado em uma máquina com 4 *CPUs*). Para montar o dicionário no começo do programa, a complexidade em tempo é a mesma do que a *trie*, pois o cálculo da *hash* da palavra é linear no tamanho dela. O *speedup* será proporcional ao número de *hosts* vezes o número de unidades de processamento de cada *host*.

6 Palavras de Tamanho Diferente de 5

O dicionário tem uma separação por tamanho de palavras. Ou seja, ele tem uma partição para palavras de tamanho 5, outra para as de 4 e assim até as de tamanho 1. Não é possível tratar o *hash* de palavras de tamanho 5 com de outros tamanhos (Ex: $hash("aaaaa") = 0$, $hash("a") = 0$). Portanto, após gerar uma *hash* randômica, é verificado primeiro as palavras de tamanho 5, depois 4 e assim em diante.

Um outro problema é o que fazer com as palavras de tamanho maior do que 5. Na especificação do projeto foi decidido que palavras maiores do que este número devem ser quebradas em várias palavras, por causa da probabilidade de se gerar uma palavra de tamanho grande, por exemplo 7: $1/26^7 \simeq 8 * 10^9$. Esse número cresce exponencialmente à medida que o tamanho da palavra cresce.

A solução desenvolvida quebra essas palavras em palavras de tamanho 5 (ou menos caso for a última parte) e atribui à elas um identificador e qual parte da palavra toda esta pequena parte representa. As palavras de tamanho menor ou

igual a 5 também receberão um identificador e elas representam toda a palavra. Há um vetor global que guarda as informações de todas as palavras, indexado pelo identificador da palavra. Cada posição guarda qual palavra é, quantas partes tem e quais já foram sorteadas. Esse vetor nada mais é do que um complemento do dicionário como um todo. As partes das palavras recebem uma máscara que devem *setar* no vetor global de informações caso sejam sorteadas. Irá existir várias palavras com o mesmo prefixo, foi interpretado que deve se gerar o número de vezes que este prefixo aparece, e não apenas uma vez, já que a própria versão sequencial neste caso levava menos do que 2 segundos para executar.

Por exemplo a palavra "aaaaaba". Ela deve ser separada em "aaaaa"e "ba". Portanto há uma máscara de 2 partes, ou seja, 2 bits. A primeira parte ("aaaaa") deve *setar* o primeiro bit (1 em decimal ou 01 em binário). Isto é, realizar a operação *or* com a máscara no vetor de informações. A segunda parte ("ba") deve *setar* o segundo bit (2 em decimal ou 10 em binário). Ela também realiza o *or* com a máscara no vetor de informações. Quando o número de bits *setados* for igual ao número de partes, significa que a palavra foi totalmente gerada.

vetor global das palavras → id: 0 palavra: "aaaaaba" partes: 2 máscara: 0
dicionário indexado pelo hash → hash: 0 id_palavra: 0 máscara: 1
hash: 1 id_palavra: 0 máscara: 2

Podem existir mais de uma palavra de tamanho 5 (ou outros tamanhos), representando partes de palavras diferentes. Será escolhido o que primeiro foi adicionado no dicionário. No final, o tempo total de geração das palavras será o mesmo.

7 Resultados e Conclusões

Uma execução em uma máquina local foi feita para comprovar que a suposição sobre a separação do problema em vários *hosts* iria ser proporcional ao *speedup* em cada nó. Para isso o arquivo do dicionário foi separado em duas, quatro e oito partes e foi executado a versão com uma *thread* em cada um destes pedaços do dicionário. Portanto, o *speedup* é da ordem de 1:

- 1 parte: 4m5.889s
- 2 partes: 2m17.614s + 1m50.646s = 4m8.260s
- 4 partes: 1m16.884s + 0m51.535s + 0m43.798s + 1m31.629s = 4m23.846s
- 8 partes: 0m49.536s + 0m39.686s + 0m34.536s + 0m19.279s + 0m24.148s + 0m22.988s + 0m39.004s + 0m35.433s = 4m24.610s

É possível observar uma pequena diferença entre os tempos pelo tempo de inicialização do programa. Porém, o *speedup* observado foi realmente de 1.

Depois foi executada a versão final (paralela) no *cluster* da disciplina com 4 *threads* em cada host, tirando uma média entre 10 repetições. O resultado se encontra na Figura 1.

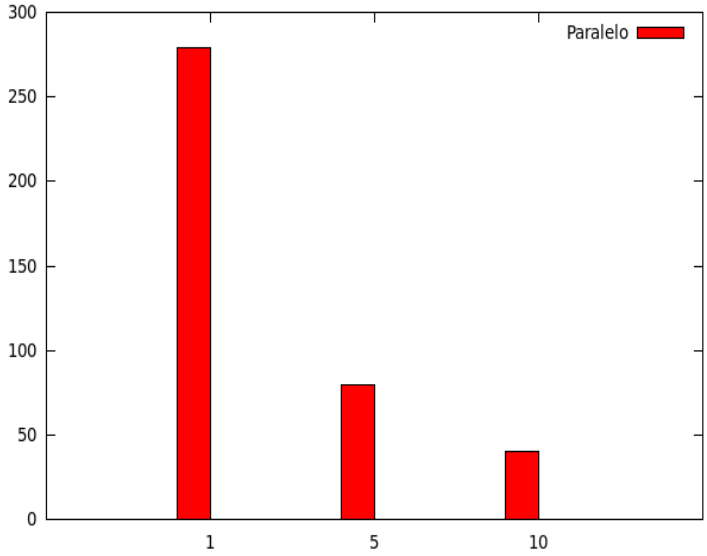

Figura 1: Execuções no *cluster*

Agora, fica comprovado que toda a teoria desenvolvida anteriormente sobre o problema está correta, já que o *speedup* foi proporcional ao número de hosts.

É possível concluir que muitas vezes a melhor solução para o problema sequencial não é totalmente aplicável e/ou pode se tornar fácilmente paralela. É necessário analisar os vários pontos de vista de um problema. A solução pode ser mais fácil do que parece e, neste caso, estava já implementada de uma maneira indireta na geração de palavras aleatórias.

Referências

[Probabilidades] http://mathworld.wolfram.com/DiscreteUniformDistribution.html

[Random] http://pubs.opengroup.org/onlinepubs/009695399/functions/rand.html

[Random] http://en.wikipedia.org/wiki/Linear_congruential_generator

[GLibC] http://www.gnu.org/software/libc/download.html

[PThreads] http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library

[PThreads] http://www.icir.org/gregor/tools/pthread-scheduling.html

[PThreads] http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

[Trie] http://www.cs.bu.edu/teaching/c/tree/trie/

[Hashing] http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-7-hashing-hash-functions/

[Hashing] http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-8-universal-hashing-perfect-hashing/

[Hashing] http://research.cs.vt.edu/AVresearch/hashing/strings.php

[MPI] http://www.slac.stanford.edu/comp/unix/farm/mpi.html

[MPI] http://www.eecis.udel.edu/~saunders/courses/372/01f/manual/manual.html

[Gnuplot] http://www.duke.edu/ hpgavin/gnuplot.html