Trabalho de Sistemas Operacionais

Aramis S. H. Fernandes GRR20034170
Jaime Biernaksi GRR20061836
Lucas N. Ferreira GRR20072901

20 de junho de 2010

Universidade Federal do Paraná
CI212 A Sistemas Operacionais
For: Luiz C. E. de Bona

---

1 Introdução

O objetivo do trabalho é resolver um problema simplificado relacionado à computação da terapia de tumores utilizando programação *Multi-thread*.

2 Arquitetura do Sistema

O Sistema segue as seguintes estruturas e funções de *threads*:
- *Threads* Produtoras;
- *Threads* Consumidoras;
- Buffer que armazena os dados gerados pelas threas produtoras;
- Estrutura de dados para armazenar as informações vindas das *threads* consumidoras. Cada nó da estrutura armazena os seguintes dados:
- distância (distância entre as células);
- quantidade (número de ocorrências daquela distância).

1. Carregar matriz de células do arquivo texto

2. Disparar *Threads* Produtoras

   Essas *threads* têm o objetivo de fazer os cálculos das distâncias entre as células. Feito o cálculo, o resultado é armazenado no buffer.

3. *Threads* consumidoras envaziam o buffer, e armazenam os dados na estrutura de dados.

3 Experimentos

No decorrer do desenvolvimento do trabalho foram feitas várias implentações, cada qual com uma abordagem diferente.

3.1 A questão das *Threads*:

A idéia inicial era fazer um cálculo sobre o tamanho do arquivo, afim de descobrir quantas células o arquivo possuía, e baseados nisso, obter uma certa porcentagem de células consumidoras e células produtoras. As células produtoras iriam abordar diferentes partes do arquivo texto ao mesmo tempo, afim de trabalhar em cima de todo o arquivo paralelamente. No entanto, após os primeiros experimentos e análises ficou claro que essa abordagem não era boa, pois existiriam muitas *threads* que ficariam ociosas enquanto outras poderiam ficar "atarefadas demais". Mudamos a forma de abordar o problema, e passamos a enviar o número de *threads* consumidoras e o número de *threads* produtoras como parâmetro na execução do programa.

3.2 A questão do Buffer:

As primeiras idéias eram construir um Buffer diferente, no entando criamos um Buffer simples, apenas um vetor de floats de tamanho N. Para acesso desse Buffer foram utilizados semáforos seguindo o princípio da teoria vista em sala. Cada *thread* faz acesso atômico ao buffer afim de evitar inconsistência dos dados.

3.3 A questão da Estrutura de dados:

As primeiras versões prontas tinham a implementação de uma árvore AVL como estrutura de armazenamento de dados, depois passamos para implementação da estrutura com uma árvore B. Nos testes realizados acabamos constatando que uma implementação de modo sequencial era mais rápida que uma implementação com o uso de duas *threads* (uma para produzir e outra para consumir). Isso gerou preocupação da equipe, e passamos a analisar onde poderia estar o gargalo.

Após as análises, concluímos que o grande problema era a forma como as informações vindas das *threads* produtoras eram armazenadas na estrutura de dados. A política adotada era a seguinte: Havia um semáforo para controlar o acesso das *threads* à árvore AVL. Isso mostrou-se muito custoso, pois a velocidade com que os dados eram produzidos era muito maior que a velocidade com a qual eram consumidos.

Cada *thread* consumidora tinha que esperar para acessar a estrutuda de dados, e esta, por ser uma árvore AVL, prejudicava esse tempo de espera ao fazer rotações. Vimos então que seria necessário uma forma de implementar a estrutura de dados para que o acesso à ela fosse possível por mais de uma *thread* consumidora ao mesmo tempo. Então surgiu a idéia de criar uma tabela *hash*, onde cada *thread* poderia acessar a tabela *hash* independente das outras n-1 *threads*, pois a chave da *hash* garantiria que valores diferentes pudessem ser inseridos em lugares diferentes. O que precisaríamos garantir era que o acesso a cada posição da *hash* fosse atômico. Isso foi resolvido com a aplicação de um semáforo em cada posição da *hash*.

Ainda assim, o acesso estava razoavelmente lento, pois em cada posição, quando haviam colisões, era criada uma lista ligada, contendo os elementos coincidentes na função *hash*, e por causa dos

semáforos, o acesso à lista ficava bloqueado, ocasionando perda no desempenho. Depois, vimos que uma lista ligada ainda não era o suficiente, e procuramos aplicar uma árvore AVL para cada elemento coincidente na *hash*.

3.4 A questão dos Semáforos:

Para controlar o acesso à memória, foram colocados semáforos em cada nó da árvore, garantindo que ficassem travados apenas os nós que possuíssem escritas ocorrendo. Mas foi detectado um grave problema de desempenho, que viria a acontecer quando a árvore precisasse ser balanceada. Para ocorrer tal balanceamento, todo acesso à árvore deveria ser bloqueado para as *threads* de escrita, e apenas as rotações deveriam ser executadas. O resultado disso foi uma queda brusca no desempenho.

Posteriormente, optamos por utilizar uma tabela *hash*, com listas ligadas nos elementos da *hash* onde ocorressem colisões. Um aumento significativo no desempenho foi percebido, mas enfrentamos alguns problemas para encontrar os pontos no programa onde ocorreram *dead_locks*. Esse problema foi percebido especificamente durante o acesso à *hash*, a estrutura que contém a região crítica.

Depois de implementada a árvore AVL para cada elemento da hash, foi colocado apenas um semáforo na entrada do nó (ou seja, na raiz de cada árvore), deixando os nós restantes da árvore sem semáforo algum.

É possível evitar colisões assim pois, dado que uma raiz de árvore fica travada, a única thread que percorre a árvore, excreve e logo libera a árvore. Percebemos que não é necessário colocar um semáforo para cada nó da árvore, e com isso conseguimos uma melhora considerável no desempenho.

4 Releases:

Nos testes com os releases, decreveremos a arquitetura utilizada e os resultados obtidos ao fim da execução. O Arquivo utilizado se baseia no arquivo enviado para testes "Cells.txt"com as 1000 primeiras posições. Algumas funções ficaram inalteradas em todas as releases, estas são:

No arquivo *produtores.c*:
getCelulas: le o arquivo texto, aloca um vetor e coloca as informações do arquivo no vetor.
calculaDistancia: executa a função $d(Cj, Ck) = SQRT((Xj - Xk)^2 + (Yj - Yk)^2 + (Zj - Zk)^2)$ e retorna seu resultado.
insereDist: quando o buffer não está cheio, insere um novo item no buffer.

1. Release 1:
   Threads: número máximo de threas = 50.
   Threads produtoras: Número de *threads* produtoras calculadas com base na quantidade de células contidas no arquivo.
   Threads consumidoras: Número de *threads* consumidoras igual ao número de threads produtoras.
   Buffer: 1 buffer estático de tamanho 10000.
   Estrutura de Dados: Árvore AVL.
   Semáforos: usados para o acesso do buffer e para o acesso da árvore; o semáforo trava a árvore toda quando uma *thread* vai escrever.

Resultados:
real 2m33.619s
user 2m31.248s
sys 4m44.405s

Após os testes realizados, verificamos que o cálculo do número de *threads* baseado na quantidade de células era uma solução inviável.

2. Release 2:
   Similar ao release 1.
   Threads: Número de *threads* passado por parâmetro.

   Resultados:
   produtoras 1 consumidoras 1
   real 0m0.470s
   user 0m0.670s
   sys 0m0.084s

   produtoras 1 consumidoras 2
   real 0m0.490s
   user 0m0.579s
   sys 0m0.142s

   produtoras 2 consumidoras 2
   real 0m0.600s
   user 0m0.649s
   sys 0m0.294s

3. Release 3:
   Threads: Número de *threads* passados por parâmetro.
   Threads produtoras:
   Threads consumidoras:
   Estrutura de Dados: árvore B.

   Resultados:
   produtoras 1 consumidoras 1
   real 0m0.423s
   user 0m0.679s
   sys 0m0.030s

   produtoras 1 consumidoras 2
   real 0m0.514s
   user 0m0.643s
   sys 0m0.239s

   produtoras 2 consumidoras 2
   real 0m0.573s
   user 0m0.775s
   sys 0m0.308s

4. Release 4:
   Descontinuada.

5. Release 5:
   Threads: número de *threads* passadas por parâmentro.
   Buffer: estático de tamanho 100000000.
   Estrutura de Dados: arvore AVL.

Resultados:
produtoras 1 consumidoras 1
real 0m0.226s
user 0m0.214s
sys 0m0.022s

produtoras 1 consumidoras 2
real 0m0.237s
user 0m0.218s
sys 0m0.043s

produtoras 2 consumidoras 2
real 0m0.231s
user 0m0.221s
sys 0m0.037s

6. Release 6:
   Versão sequencial.

   Resultados:
   real 0m0.160s
   user 0m0.144s
   sys 0m0.007s

   Até então foram direcionados esforços para tipos de estruturas e otimização de operações. Após analisar estes resultados partimos para outra forma de abordar o problema, vendo que a única saída para melhorar o desempenho era fazer a inserção concorrente na estrutura (este é o principal gargalo do sistema).

7. Release 7:
   Threads: número de *threads* passadas por parâmetro.
   Buffer: estático de tamanho 100000000.
   Estrutura de Dados: A principal melhoria desta abordagem é a inserção paraleda de dados numa tabela *hash*.
   Cada nó da tabela *hash* possui:
   - valor da distância calculada;
   - quantidade de ocorrências da distância;
   - semáforo para controle de acesso;
   - ponteiro para o próximo nó caso hajam colisões no cálculo da função *hash*;

8. Release Final:
   Threads: número de *threads* passadas por parâmetro.
   Buffer: estático de tamanho 1024.
   Estrutura de Dados: A principal melhoria desta abordagem é a inserção paralela de dados numa tabela *hash*. E uma leitura de uma sequência de dados passados para as *threads* produtoras.
   Cada nó da tabela *hash* possui um ponteiro para a raiz de uma árvore e um semáforo, que controla o acesso à árvore.
   Cada árvore possui em seu nó:
   - o valor da distância calculada;
   - quantidade de ocorrências da distância;

5 Conclusão

É possível atingir uma melhora significativa com o uso da programação paralela, mas existem alguns cuidados que devem ser tomados para que se consiga tal melhoria, caso contrário, o desempenho

tende a piorar, dado que quando mais de um processo tenta acessar uma região crítica do sistema, os controles de acesso do próprio sistema tendem a piorar o desempenho do conjunto de threads.