Segundo Trabalho - Relatório Final

Universidade Federal do Rio de Janeiro

Disciplina: Computação Concorrente (MAB-117)

Período: 2021/1 Remoto

Título do Trabalho: Problema Produtor-Consumidor

Nomes dos Membros: Gabriel Almeida Mendes, Luiz Rodrigo Lacé Rodrigues

DRE dos Membros: 117204959, 118049873

1. Projeto final e implementação da solução concorrente

Primeiramente, precisamos implementar uma solução com o padrão produtores/consumidores. Onde haveria apenas uma thread produtora que leria um bloco por vez do arquivo de entrada e colocaria no buffer para que 1 ou mais threads consumidoras pudessem ordenar cada slot desse buffer e escrevê-lo no arquivo de saída.

No inicio da nossa aplicação, nós vamos passar pelo terminal a quantidade de threads consumidoras, o tamanho do bloco que será ordenado, o nome do arquivo de entrada e o nome do arquivo de saída.

Fazemos todo o tratamento necessário dessas informações passadas. Como passar a quantidade de números e o tamanho do bloco para variáveis globais e também abrir o arquivo de entrada para a leitura e o arquivo de saída para a escrita (ambos em escopo global para que possam ser acessados pelas threads)

Iniciamos a tomada de tempo e criamos a nossa única threads produtora e as threads consumidoras, baseado no que recebemos do terminal.

A nossa thread produtora, dentro de um loop infinito, vai chamar a função insere, que não possui parâmetros, essa função é envolta por um mutex. Precisamos fazer a exclusão mútua visto que estamos trabalhando com variáveis globais, como a variável slotsDisponíveis, que vai indicar quantos slots estão disponíveis para que as threads consumidoras possam retirar do buffer e ordená-los. Caso essa variável seja igual a 10 (tamanho máximo do buffer decidido pelo enunciado do trabalho, 10 slots de tamanho baseado no bloco passando pelo terminal), a thread vai entrar em espera até a thread consumidora decrementar essa variável e a liberá-la.

Caso contrário, vamos ler o arquivo de entrada, iterando buffer, que é um ponteiro de inteiros como se fosse uma matriz, onde o index é incrementado posteriormente com um raciocínio em que, quando chega no final, ele volta para o início, dessa forma não precisamos nos preocupar em “estourar” o buffer. Após a inserção de um slot, decrementamos a variável quantValores (valor da primeira linha do arquivo) em um bloco, assim podemos ter noção quantos valores ainda precisam ser lidos e incrementamos a variável slotsDisponiveis e liberamos uma thread para consumi-lo.

Após a saída da função insere() a thread produtora verifica se a quantidade de valores a serem lidos é igual a 0, caso seja, ela não tem mais serventia e é finalizada, sendo recebida pela pthread_join.

Para as threads consumidoras, começamos com um ponteiro de inteiros, que vai receber um slot pela função “retira”, também sem parâmetros.
Essa função, envolta por um mutex, visto que vamos acessar novamente a variável slotsDisponiveis, que caso seja igual a 0, a thread vai verificar se ainda possui algum número a ser escrito, caso não tenha, a thread é finalizada. Caso contrário, ela fica apenas esperando que a thread produtora insira um slot no buffer e a libere para o tratamento.

O que fazemos em seguida é passar o conteúdo do buffer, naquele slot disponível para um ponteiro alocado nessa função,utilizando um index de saída parecido com o de entrada, para que não corrermos o risco de retirar o mesmo slot mais de uma vez, decrementamos então a variável slotsDisponíveis para que a thread produtora possa vir a inserir mais um, caso necessario, e liberamos a thread produtora, caso esteja esperando.

Finalmente retornamos o ponteiro “vetor” para o ponteiro “item” na thread.

Esse ponteiro “item”, agora com o slot retirado, é passado para a função ordenaEscreve como parâmetro, dentro dele ele é ordenado com o algoritmo bubblesort. Após a ordenação, passamos por um teste para verificar se o slot atual a ser escrito está realmente ordenado, caso encontremos algum erro, retornamos uma mensagem de erro de corretude e fechamos a aplicação. Caso esteja tudo certo, estamos prontos para a escrita no arquivo. A escrita é feita com um mutex exclusivo para ela, assim podemos ordenar e escrever de forma concorrente entre as threads, mas sempre escrevendo de forma única. Utilizamos o mutex_escrita para essa exclusão mútua da escrita. Após a escrita, incrementamos a variável quantValoresEscritos em um tamanho de bloco, dessa forma temos o controle de quantos números foram escritos no arquivo.

Saindo do mutex_escrita e entrando no nosso mutex normal, avaliamos se quantValores é igual a 0, caso seja, damos um pthread_mutex_broadcast, para que todas as threads que estiverem esperando na função “insere” sejam liberadas e finalizadas.

Após a finalização da função ordenaEscreve,na thread, temos a avaliação, caso slotsDisponiveis e quantValores sejam iguais a zero a última thread consumidora é finalizada.

Recebemos tudo na main pela pthread_join e finalizamos a tomada de tempo e o printamos.

Antes de finalizarmos a aplicação, temos ainda a corretude da quantidade de valores escritos. Caso quantValoresEscritos, que foi incrementada sempre que um valor era escrito, seja diferente de quantValoresTotal, que recebeu também a primeira linha do arquivo, então retornamos uma mensagem de erro e finalizamos a aplicação.

Caso passe pelo teste de corretude com sucesso, retornaremos a mensagem “Aplicação finalizada com sucesso!”. Liberamos o buffer e os arquivos de entrada e saída e o programa finalmente acaba como deveria.

2. Testes de corretude:

Para a correção da nossa aplicação, fizemos dois testes:

- O primeiro teste se trata de uma checagem no slots após a sua ordenação e logo antes de ser escrito. Caso o slots não esteja ordenado de forma crescente, a aplicação para antes da escrita e retorna uma mensagem de erro de corretude.
- O segundo teste foi implementado perto do final do programa, após recebermos todas as threads pela função pthread_join. O que fazemos agora é avaliar se a quantidade de valores escritos no arquivo de saída é exatamente igual a quantidade de valores lidos na primeira linha do arquivo de entrada. Caso os números não sejam exatamente iguais, uma mensagem de erro é retornada e o programa é finalizado.

Para termos certeza que a aplicação terminou da forma que queríamos, caso nenhuma dessas mensagens de erro sejam retornadas, colocamos uma mensagem de sucesso: “Aplicação finalizada com sucesso!”

Casos de teste para a corretude

Para arquivos de dimensões 195000, 3072000 e 49152000 do arquivo de entrada temos os casos abaixo:

Caso 1 - Dimensão = 195000; Tamanho do bloco = 100; nThreads = 1 ;
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 2 - Dimensão = 195000; Tamanho do bloco = 100; nThreads = 2
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 3 - Dimensão = 195000; Tamanho do bloco = 100; nThreads = 4
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 4 - Dimensão = 195000; Tamanho do bloco = 1000; nThreads = 1
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 5 - Dimensão = 195000; Tamanho do bloco = 1000; nThreads = 2
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 6 - Dimensão = 195000; Tamanho do bloco = 1000; nThreads = 4
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 7 - Dimensão = 3072000; Tamanho do bloco = 100 ; nThreads = 1
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 8 - Dimensão = 3072000; Tamanho do bloco = 100 ; nThreads = 2
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 9 - Dimensão = 3072000; Tamanho do bloco = 100 ; nThreads = 4
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 10 - Dimensão = 3072000; Tamanho do bloco = 1000 ; nThreads = 1
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 11 - Dimensão = 3072000; Tamanho do bloco = 1000 ; nThreads = 2
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 12 - Dimensão = 3072000; Tamanho do bloco = 1000 ; nThreads = 3
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 13 - Dimensão = 49152000; Tamanho do bloco = 100 ; nThreads = 1
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 14 - Dimensão = 49152000; Tamanho do bloco = 100; nThreads = 2
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 15 - Dimensão = 49152000; Tamanho do bloco = 100; nThreads = 4
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 16 - Dimensão = 49152000; Tamanho do bloco = 1000 ; nThreads = 1
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 17 - Dimensão = 49152000; Tamanho do bloco = 1000; nThreads = 2
Avaliação de Corretude: Aplicação finalizada com sucesso!

Caso 18 - Dimensão = 49152000; Tamanho do bloco = 1000; nThreads = 4
Avaliação de Corretude: Aplicação finalizada com sucesso!

3. Avaliação de desempenho

Para a avaliação de desempenho, utilizamos a macro “timer.h” que foi disponibilizada pela professora Silvana durante as aulas do curso. Essa macro nos permitiu salvar o tempo inicial e final em variáveis diferentes e subtraí-los para fazermos o cálculo do tempo de execução. Tomamos o tempo inicial a partir do momento que as threads são criadas com as funções pthread_create e o tempo final no momento em que as recebemos de volta com a função pthread_join. Dessa forma podemos tomar o tempo apenas do que nos interessa para a avaliação da aceleração.

Para visualizar da melhor forma possível, escolhemos rodar a nossa aplicação com 1, 2 e 4 threads com arquivos de 3 tamanhos diferentes:

- O primeiro arquivo possui 195000 números no total e pesa 429,0 kB.
- O segundo arquivo possui 3072000 números no total e pesa 6,8 MB.
- O terceiro e último arquivo possui 49152000 números no total e pesa 108,1 MB

A dimensão dos arquivos foi pensada de forma que pudéssemos testar a aplicação com pelo menos 2 tamanhos diferentes de blocos para a tomada e ordenação.

Para não termos riscos de termos que lidar com números quebrados, utilizamos primeiramente blocos de 100 números e em seguida 1000 números.

Aqui está a configuração da máquina onde os testes foram realizados;

Sistema Operacional:	"20.04.3 LTS (Focal Fossa)", Ubuntu, Linux.
Arquitetura:	x86_64
Modo(s) operacional da CPU:	32-bit, 64-bit
Ordem dos bytes:	Little Endian
Address sizes:	43 bits physical, 48 bits virtual
CPU(s):	8
Lista de CPU(s) on-line:	0-7
Thread(s) per núcleo:	2
Núcleo(s) por soquete:	4

Soquete(s):	1
Nó(s) de NUMA:	1
ID de fornecedor:	AuthenticAMD
Família da CPU:	23
Modelo:	24
Nome do modelo:	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx

Casos de teste para a avaliação de desempenho

Para arquivos de dimensões 195000, 3072000 e 49152000 do arquivo de entrada temos os casos abaixo:

Caso 1 - Dimensão = 195000; Tamanho do bloco = 100; nThreads = 1 ;
Vezes que este caso foi executado: 5
Menor Tempo de Desempenho do caso: **0.166910**

Caso 2 - Dimensão = 195000; Tamanho do bloco = 100; nThreads = 2
Vezes que este caso foi executado: 5
Menor Tempo de Desempenho do caso: **0.092069**

Caso 3 - Dimensão = 195000; Tamanho do bloco = 100; nThreads = 4
Vezes que este caso foi executado: 5
Menor Tempo de Desempenho do caso: **0.078976**

Caso 4 - Dimensão = 195000; Tamanho do bloco = 1000; nThreads = 1
Vezes que este caso foi executado: 5
Menor Tempo de Desempenho do caso: **1.107352**

Caso 5 - Dimensão = 195000; Tamanho do bloco = 1000; nThreads = 2
Vezes que este caso foi executado: 5
Menor Tempo de Desempenho do caso: **0.552241**

Caso 6 - Dimensão = 195000; Tamanho do bloco = 1000; nThreads = 4
Vezes que este caso foi executado: 5
Menor Tempo de Desempenho do caso: **0.344018**

Caso 7 - Dimensão = 3072000; Tamanho do bloco = 100 ; nThreads = 1
Vezes que este caso foi executado: 5
Menor Tempo de Desempenho do caso: **2.034677**

Caso 8 - Dimensão = 3072000; Tamanho do bloco = 100 ; nThreads = 2
Vezes que este caso foi executado: 5
Menor Tempo de Desempenho do caso: **1.830811**

Caso 9 - Dimensão = 3072000; Tamanho do bloco = 100 ; nThreads = 4
Vezes que este caso foi executado: 5
Menor Tempo de Desempenho do caso: **1.397613**

Caso 10 - Dimensão = 3072000; Tamanho do bloco = 1000 ; nThreads = 1
Vezes que este caso foi executado: 4
Menor Tempo de Desempenho do caso: **13.045372**

Caso 11 - Dimensão = 3072000; Tamanho do bloco = 1000 ; nThreads = 2
Vezes que este caso foi executado: 4
Menor Tempo de Desempenho do caso: **7.786465**

Caso 12 - Dimensão = 3072000; Tamanho do bloco = 1000 ; nThreads = 4
Vezes que este caso foi executado: 4
Menor Tempo de Desempenho do caso: **5.107540**

Caso 13 - Dimensão = 49152000; Tamanho do bloco = 100 ; nThreads = 1
Vezes que este caso foi executado: 4
Menor Tempo de Desempenho do caso: **34.305953**

Caso 14 - Dimensão = 49152000; Tamanho do bloco = 100; nThreads = 2
Vezes que este caso foi executado: 4
Menor Tempo de Desempenho do caso: **29.313104**

Caso 15 - Dimensão = 49152000; Tamanho do bloco = 100; nThreads = 4
Vezes que este caso foi executado: 4
Menor Tempo de Desempenho do caso: **20.793163**

Caso 16 - Dimensão = 49152000; Tamanho do bloco = 1000 ; nThreads = 1
Vezes que este caso foi executado: 3
Menor Tempo de Desempenho do caso: **207.820072**

Caso 17 - Dimensão = 49152000; Tamanho do bloco = 1000; nThreads = 2
Vezes que este caso foi executado: 3
Menor Tempo de Desempenho do caso: **129.887116**

Caso 18 - Dimensão = 49152000; Tamanho do bloco = 1000; nThreads = 4
Vezes que este caso foi executado: 3
Menor Tempo de Desempenho do caso: **89.754118**

Cálculos da aceleração:

Utilizamos a lei de Amdahl para estimar os ganhos de desempenho, onde o ganho de velocidade é calculado por Tsequencial/ts + tp(P). Temos:

- ts: tempo da parte sequencial do programa (que não será dividida entre threads)
- tp(P): tempo da parte paralela do programa usando P processadores
- T(P): tempo total de execução previsto do programa usando P processadores (ts +tp(P)).

Como estamos trabalhando com apenas um processador e a nossa aplicação não possui implementação sequencial, vamos comparar os casos que possuímos mais de uma thread consumidora em relação a aos casos que só temos uma consumidora. Os resultados da acelerações são os seguintes:

Dimensão = 195000; Tamanho do bloco = 100; nThreads = 2
0.166910 / 0.092069 = 1,812879471

Dimensão = 195000; Tamanho do bloco = 100; nThreads = 4
0.166910 /0.078976 = 2,113426864

Dimensão = 195000; Tamanho do bloco = 1000; nThreads = 2
1.107352 / 0.552241 = 2,005197006

Dimensão = 195000; Tamanho do bloco = 1000; nThreads = 4
1.107352 / 0.344018 = 3,218878082

Dimensão = 3072000; Tamanho do bloco = 100 ; nThreads = 2
2.034677 / 1.830811 = 1,111352838

Dimensão = 3072000; Tamanho do bloco = 100 ; nThreads = 4
2.034677 / 1.397613 = 1,455822892

Dimensão = 3072000; Tamanho do bloco = 1000 ; nThreads = 2
13.045372 / 7.786465 = 1,675390822

Dimensão = 3072000; Tamanho do bloco = 1000 ; nThreads = 4
13.045372 / 5.107540 = 2,554139958

Dimensão = 49152000; Tamanho do bloco = 100; nThreads = 2
34.305953 / 29.313104 = 1,170328226

Dimensão = 49152000; Tamanho do bloco = 100; nThreads = 4
34.305953 / 20.793163 = 1,649866978

Dimensão = 49152000; Tamanho do bloco = 1000; nThreads = 2
207.820072 / 129.887116 = 1,600005285

Dimensão = 49152000; Tamanho do bloco = 1000; nThreads = 4
207.820072 / 89.754118 = 2,315437738

Em resumo temos essas tabelas resumindo os resultados das acelerações em relação a apenas uma thread consumidora, com apenas 2 casas decimais e arredondamentos:

Dimensão = 195000	Threads consumidoras: 2	Threads consumidoras: 4
Tamanho bloco: 100	1,81	2,11
Tamanho bloco: 1000	2,00	3,20

Dimensão = 3072000	Threads consumidoras: 2	Threads consumidoras: 4
Tamanho bloco: 100	1,11	1,45
Tamanho bloco: 1000	1,67	2,55

Dimensão = 49152000	Threads consumidoras: 2	Threads consumidoras: 4
Tamanho bloco: 100	1,17	1,65
Tamanho bloco: 1000	1,60	2,31

Podemos perceber que em todos os casos, obtivemos alguma aceleração, visto que o resultado é sempre maior que 1.

4. Discussão

Já esperávamos previamente que os resultados e análises dos ganhos de desempenho fossem mais expressivos e evidentes para arquivos com uma quantidade muito grande de valores em comparação a arquivos com quantidade menores. Então, quando vimos os resultados dos testes que realizamos, vimos que os resultados já eram os esperados previamente.

Para arquivos muito pequenos, como o exemplo do enunciado que só possuía 20 números, os ganhos de desempenho eram inexpressivos, visto que poucas thread consumidoras teriam trabalho efetivo e o tamanho dos blocos era limitado. Após aumentarmos os arquivos de entrada de forma consideravel, conseguimos perceber os ganhos da utilização de mais de uma thread consumidora.

O que percebemos também, é que a escrita no arquivo é rápida e o que é mais demorado é a ordenação dos blocos, dessa forma escrever no arquivo blocos ordenados de 100 é mais rápido do que blocos ordenados de 1000, o que comprova a nossa teoria de que quando temos pequenos valores de entrada, não existe muita diferença utilizar mais de uma thread. Na

verdade podemos ter até uma gasto de memória desnecessário, o que fará com que a nossa aplicação demore mais com mais threads.

O que poderíamos melhorar no programa é a iteração que fazemos do slot do buffer para o vetor alocado na função “retira”, onde poderíamos passar a referência de memória diretamente, sem necessidade de iteração. Mas por falta de tempo não conseguimos implementar.

As dificuldades encontradas foram a de interpretar o enunciado de primeira. Adaptar o padrão produtores e consumidores para o problema tratado. Relembrar os conceitos de tratamento de arquivo e implementar um raciocinio para os casos para que a aplicação terminasse.

5. Referências bibliográficas

Aulas de computação concorrente da UFRJ/2021.1 remoto

6. Repositório para o código da implementação:

https://gitlab.com/luizrodrigolace/laboratiorios-computacao-concorrente/-/tree/master/trabalhos/trabalho%202