Trabalho Bubble Sort Concorrente

Marcus Vinícius Torres de Oliveira. DRE:118142223

Computação Concorrente (ICP-117) — 2022/1

1. Descrição do problema

A proposta deste trabalho é a de implementar o algoritmo de ordenação Bubble Sort de forma concorrente para ordenar vetores com dimensões de até **10^5**. Neste programa o usuário deve informar como entrada a **dimensão** do vetor e a **quantidade N de threads** que quer utilizar, a saída é o **vetor ordenado**. Dois vetores com os mesmos números aleatórios entre 0 e 999 são gerados pelo programa, um deles será ordenado de forma sequencial e o outro de forma concorrente.

Como o problema se trata de uma forma de ordenar vetores, entendemos que o mesmo pode ser feito de forma concorrente, pois o vetor pode ser dividido em várias partes e a ordenação dessas partes pode ser feita de forma independente. Portanto, dada essa natureza do problema, o usuário se beneficia porque o programa ordenará essas partes do vetor de forma paralela.

2. Projeto e implementação da solução concorrente

Estratégias pensadas para a resolução do trabalho:

1. Dividir o vetor em blocos iguais e cada thread ordinária o seu bloco

2. Dividir o vetor em blocos de acordo com o maior elemento do vetor e cada thread ordinária o seu bloco

Percebi que a primeira implementação não gera um resultado válido, pois ao juntar as partes do vetor, elas não estariam ordenadas entre si. Portanto, a segunda implementação foi escolhida.

Dada a implementação escolhida, **três escolhas principais** foram feitas. A **primeira** é a **busca do maior elemento do vetor**. Esse elemento é utilizado para separar os blocos do vetor de acordo com uma faixa de valores para cada thread, assim, na hora de juntar os blocos, eles estariam ordenados entre si. Essa função foi feita de forma sequencial e é chamada pela main. Isso deve-se ao fato de trabalharmos com **dimensões pequenas** nesse tipo de ordenação, portanto, não há necessidade de implementar uma função concorrente para tal tarefa, já que não haveria ganho de desempenho, como vemos no Lab 3.

Já a **segunda** escolha principal foi a **construção dos blocos globais do vetor** (chamados de sublistas na implementação do código). Elas são os vetores que cada thread vai organizar. A função que gera esses vetores é implementada de forma sequencial e é chamada pela main. Sua implementação também deve-se ao fato de que não haveria ganho de desempenho significativo se fosse feita da forma concorrente para a faixa de dimensões dos vetores que estamos lidando.

Por fim, a **terceira** escolha principal foi a decisão de **armazenar o tamanho de cada bloco em um vetor global separado**. Esta solução foi a única ideia que me surgiu para lidar com o armazenamento do tamanho dos blocos. **Provavelmente há soluções mais eficientes**.

3. Casos de teste

O programa é testado comparando o resultado sequencial com o concorrente através de uma função. Esta função é responsável por verificar a igualdade dos dois vetores(o sequencial e o concorrente). Caso um apresente um elemento em uma determinada posição diferente do outro, ela gera uma mensagem informando que ambos não

são iguais. No caso em que ambos são iguais, ela informa o usuário que está tudo certo e que os vetores são iguais.

Ademais, o programa possui uma outra função para checar se o vetor está ordenado corretamente. Como foi decidido que a ordenação seria crescente, essa função checa o elemento(i) e compara com o elemento(i+1). Caso o elemento(i+1) seja menor do que o elemento(i), o vetor não está ordenado corretamente. Isso faz com que o programa gere uma mensagem informando o usuário. Já no caso em que essa verificação segue sem nenhum erro, o programa informa o usuário que o vetor está ordenado corretamente.

Observação: Para verificarmos a veracidade da informação gerada pelas funções descritas acima, foi utilizado prints e vetores com dimensões pequenas. Desta forma, percebi que tanto o método sequencial quanto o concorrente estavam realmente ordenando os vetores como esperado.

4. Avaliação de desempenho

Para verificar o ganho de desempenho, o programa utiliza a macro “timer.h” apresentada no Lab 2, para medir o tempo que a função sequencial e a concorrente levam para terminar as suas execuções. De forma que o ganho de desempenho com a versão concorrente é medido da forma: Tempo Sequencial / Tempo Concorrente. Como é mostrado no log gerado pelo programa ao fim de sua execução.

Foram gerados 5 testes para vetores com dimensões de tamanho: 10^3, 10^4 e 10^5. Com número de threads: 1, 2, 4. O log gerado se encontra no arquivo Resultado_dos_testes.md do github que será linkado com a entrega do trabalho. Dos testes gerados aqui apresentarei os resultados com o maior ganho de desempenho para cada caso. Segue abaixo:

Dimensão = 10^3 ; Número de threads = 1:

Tempo sequencial: 0.004729

Tempo concorrente: 0.003940

Ganho de desempenho: 1.200399

Dimensão = 10^3 ; Número de threads = 2:

Tempo sequencial: 0.003666

Tempo concorrente: 0.001360

Ganho de desempenho: 2.695362

Dimensão = 10^3 ; Número de threads = 4:

Tempo sequencial: 0.005235

Tempo concorrente: 0.001662

Ganho de desempenho: 3.149756

Dimensão = 10^4 ; Número de threads = 1:

Tempo sequencial: 0.289748

Tempo concorrente: 0.274201

Ganho de desempenho: 1.056701

Dimensão = 10^4 ; Número de threads = 2:

Tempo sequencial: 0.290823

Tempo concorrente: 0.064017

Ganho de desempenho: 4.542904

Dimensão = 10^4 ; Número de threads = 4:

Tempo sequencial: 0.288865

Tempo concorrente: 0.064590

Ganho de desempenho: 4.472316

Dimensão = 10^5 ; Número de threads = 1:

Tempo sequencial: 31.697388

Tempo concorrente: 30.153166

Ganho de desempenho: 1.051213

Dimensão = 10^5 ; Número de threads = 2:

Tempo sequencial: 31.645968

Tempo concorrente: 7.746057

Ganho de desempenho: 4.085429

Dimensão = 10^5 ; Número de threads = 4:

Tempo sequencial: 32.268335

Tempo concorrente: 7.870958

Ganho de desempenho: 4.099671

5. Discussão:

Os resultados obtidos estão de acordo com o esperado, já que a implementação concorrente está mais rápida do que a sequencial conforme a dimensão do vetor e o número de threads aumentam.

Como já citado, há três possíveis melhorias. A busca do maior elemento poderia ser implementada de forma concorrente, assim como a criação dos blocos de vetores. Além disso, a implementação do armazenamento do tamanho de cada bloco.

Quanto às dificuldades encontradas, posso citar a lógica da divisão do vetor e o armazenamento do tamanho de cada bloco. Como vimos no tópico 2 deste relatório, a primeira ideia era separar o vetor por blocos de tamanhos iguais, sem se importar com a lógica da divisão dos elementos dentro de cada bloco.

6. Referências:

Site OpenGenus IQ: https://iq.opengenus.org/parallel-bubble-sort/