Trabalho de Implementação

Computação Concorrente (ICP117) – 2021/2

Implementação Concorrente do Algoritmo QuickSort

Diego Mattos Marinho Malta 119044957

Felipe Dias de Melo 119093752

1. Descrição do problema:

- Do que se trata e como deve funcionar
  - O algoritmo QuickSort é um método de ordenação de vetores bastante rápido e eficiente, criado por Charles Antony Richard Hoare em 1960. Ele criou o algoritmo para tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que possam ser resolvidos de forma mais fácil e rápida. Portanto, trata-se de um algoritmo que adota a estratégia de divisão e conquista utilizando um elemento como pivô, rearranjando os valores de modo que os elementos menores precedam os maiores. Em seguida, os dois subconjuntos (de valores menores que o pivô e de valores maiores que ele) são ordenados recursivamente até que o vetor completo esteja ordenado.

- Quais são os dados de entrada e qual a saída que deve ser gerada
  - Dados de entrada:
    - Vetor que desejamos ordenar;
    - Posição inicial do vetor;
    - Tamanho do vetor.
    - Tipo de vetor (aleatório, decrescente ou ordenado)
  - Saída esperada:
    - O mesmo vetor, ordenado;
    - Teste de verificação da ordenação;
    - Tempo de execução da solução sequencial;
    - Tempo de execução da solução concorrente.

- Apontar que o problema pode se beneficiar de uma solução concorrente e suas justificativas
  - Tal como a multiplicação de matrizes, a tarefa de ordenar um vetor também pode ser feita de forma concorrente. Pela própria natureza do algoritmo, são formados subconjuntos, os quais podemos destinar às threads, que serão responsáveis por ordená-los individualmente.

2. Projeto e implementação da solução concorrente:

- Listar as estratégias que podem ser usadas para dividir a tarefa principal entre fluxos de execução independentes
    - Uma estratégia possível é dividir o vetor em subvetores que serão ordenados pelas threads com base em seus identificadores;
    - Também é possível criar uma nova thread a cada chamada recursiva da função. No entanto, essa estratégia não permite que o número de threads seja definido pelo usuário.

- Apontar qual estratégia foi escolhida e por qual motivo
    - Portanto, foi escolhida a primeira estratégia, de modo que o número de threads pode ser alterado a cada execução;
    - O tamanho do vetor (`N`) será dividido pelo número de threads (`n_threads`) para definir o tamanho dos blocos (`tam_bloco`) que cada thread será encarregada de ordenar. Isso garante o balanceamento de carga em casos que `N` é divisível por `n_threads`;
    - Nos casos que `N` não for múltiplo de `n_threads`, a última thread será responsável por um bloco de tamanho adicional no máximo de $tam\_bloco + n\_threads - 1$. Como o número de threads, em geral, não deve passar de dois dígitos, não há desbalanceamento significativo;
    - Ao final, teremos o vetor ordenado por blocos, que precisam ser fundidos. Isso será feito de forma sequencial pela função `merge()`, localizada logo após da chamada de `pthread_join()`.

- Descrever as principais decisões de implementação adotadas e suas justificativas
    - Se o número de threads passado for 0 ou 1, o algoritmo será executado de forma sequencial;
    - Se o tipo de vetor passado por linha de comando for diferente de 1, 2 e 3, o padrão é preencher o vetor de maneira aleatória;
    - Foi criada uma estrutura de dados para permitir a passagem de múltiplos argumentos para as threads;
    - A principal decisão está relacionada à função `merge()`. Como o vetor está ordenado apenas em blocos, foi necessária uma forma de fundir cada subvetor enquanto ordena. Apesar de essa função ser chamada de maneira sequencial, ainda há ganho significativo em relação ao algoritmo original;
    - A função `merge()` também utiliza um vetor auxiliar que servirá para receber de forma ordenada os valores dos subvetores e copiá-los para o vetor original.

3. Casos de Teste:

- Descrever o conjunto de casos de teste usados para avaliação da corretude da solução proposta
    - Podem ser realizados testes com 3 tipos de vetores:
        - Vetor aleatório;
        - Vetor com elementos em ordem decrescente;
        - Vetor ordenado.

- Há uma função chamada `verifica()`, responsável por checar se os vetores foram ordenados de maneira correta.

- Descrever o conjunto de casos de testes usados para a avaliação de desempenho
  - Para a avaliação de desempenho, é contado o tempo de execução de cada trecho, com o auxílio da biblioteca `timer.h`;
  - Um script em Python foi utilizado para testar as execuções do programa.

4. Avaliação de desempenho:

- Descrever a configuração da máquina onde os testes foram realizados
  - Processador Intel Core i5-7400:
    - 4 núcleos;
    - 4 threads.
  - Sistema Operacional: Ubuntu 20.04 LTS (Windows Subsystem for Linux);
  - 16 GB de memória;
  - Foram utilizados testes com vetores preenchidos de maneira aleatória;
  - Os vetores dos testes possuem dimensões (em milhões): 1, 2, 4, 8, 16, 32;
  - O teste de cada dimensão foi realizado com 2, 3 e 4 threads;
  - Cada execução do script de teste possui 18 testes e foram tomados os melhores tempos a partir de 5 execuções distintas;
  - Ganho do algoritmo concorrente em relação ao sequencial:

Dimensão do vetor	2 threads	3 threads	4 threads
$1\ x\ 10^6$	1.62027	2.08328	2.18871
$2\ x\ 10^6$	1.94779	2.17031	2.54256
$4\ x\ 10^6$	1.72203	2.19918	2.38078
$8\ x\ 10^6$	1.74961	2.35093	2.43704
$16\ x\ 10^6$	1.74069	2.17548	2.42177
$32\ x\ 10^6$	1.74804	2.28922	2.37897

  - Taxa do `merge()` em relação ao tempo total da execução concorrente:

Dimensão do vetor	2 threads	3 threads	4 threads
$1\ x\ 10^6$	13.34%	22.78%	30.08%
$2\ x\ 10^6$	13.82%	23.84%	31.28%
$4\ x\ 10^6$	13.35%	22.89%	30.15%
$8\ x\ 10^6$	13.14%	22.61%	29.36%
$16\ x\ 10^6$	12.57%	22.83%	31.47%
$32\ x\ 10^6$	13.53%	22.66%	30.16%

  - Como esperado, quanto mais threads são utilizadas, maior é a taxa das chamadas sequenciais da função `merge()` em relação ao tempo total que a solução concorrente leva. Ainda assim, utilizando 4 threads, essa taxa fica em torno de 30% (um valor razoável).

5. Discussão:

- Discutir se o ganho de desempenho alcançado foi o esperado ou não e por quais motivos:
  - O ganho obtido foi o esperado, visto que há o custo da execução sequencial das chamadas da função `merge()`;
  - Os testes de corretude são verificados no script em Python. É realizada uma varredura em cada teste e é feita uma verificação se o código de retorno foi maior que zero, indicando um possível erro.

- Descrever dificuldades encontradas para a realização do trabalho:
  - A dificuldade principal se deu justamente após o término de execução das threads. Precisávamos pensar em maneiras de fundir os subvetores no vetor original, ordenando seus elementos;
  - A partir disso, utilizamos a função `merge()`.

6. Referências bibliográficas:

- Michael J. Quinn – Parallel Programming with MPI and OpenMP;
- Selim G. Akl – Parallel Sorting Algorithms;
- Guy E. Blelloch & Bruce M. Maggs – Parallel Algorithms;
- Notas de aula sobre os vídeos e materiais complementares disponibilizados pela professora.

Códigos disponíveis aqui.