Universidade Federal
do Espírito Santo

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Sistemas Distribuídos – 2023/1

Prof. Rodolfo da Silva Villaça – rodolfo.villaca@ufes.br
Monitor: Eduardo M. Moraes Sarmento – eduardo.sarmento@ufes.br

Laboratório I – Paralelismo de Processos e *Threads*

Objetivo:
Experimentar o paralelismo por meio de processos e *threads* na ordenação de um vetor de grandes dimensões. Comparar o tempo de execução da tarefa de ordenação de vetores com diferentes quantidades de processos ou *threads*.

Instruções:

Recentemente, você aprendeu sobre paralelismo de processos e *threads*, que permitem que um usuário configure e execute paralelamente várias instâncias de um único processo. Neste laboratório você programará um algoritmo básico de ordenação *mergesort* com múltiplos processos (ou threads).

Como exemplo, considere o vetor a seguir:

```
10 49 9 34 32 37 37 48 45 19 5 31 20 19 29 22 45 30 40 31 35 8 36 39 14
```

O seu programa de ordenação paralela receberá dois argumentos de linha de comando: o número de vezes que os dados devem ser divididos como parte da ordenação *mergesort* (k) em sua primeira iteração, e o tamanho do vetor de entrada a ser ordenado (n). Por exemplo, um k igual a 5 indica que os dados devem ser divididos em 5 segmentos de tamanho aproximadamente iguais. Se considerarmos executar este exemplo com um k igual a 5, os dados serão divididos da seguinte maneira:

Split #1	Split #2	Split #3	Split #4	Split #5
10 49 9 34 32	37 37 48 45 19	5 31 20 19 29	22 45 30 40 31	35 8 36 39 14

A primeira parte do *mergesort* paralelo exige que você faça a ordenação de cada segmento individual, em paralelo. Em nosso exemplo, isso significa que você lançará cinco *threads* (ou processos) – uma para cada uma das divisões – e cada processo (ou *thread*) deve ordenar apenas seu segmento. Depois desta primeira iteração, todos os processos (ou *threads*) de ordenação foram concluídos e os segmentos ficarão parecidos com:

Split #1	Split #2	Split #3	Split #4	Split #5
9 10 32 34 49	19 37 37 45 48	5 19 20 29 31	22 30 31 40 45	8 14 35 36 39

Em seguida, o seu programa pegará pares adjacentes, por exemplo o par #1 e #2, 3# e #4 e assim por diante, e os mesclará (*merge*), **também em paralelo**. Isso será feito em rodadas, onde a cada rodada um número de fusões (*merge*) paralelas ocorrem até que reste apenas uma única lista ordenada. Lembre-se, mesclar é uma operação O(n) NÃO O(n*log(n)) e por isso você **não deve** chamar usar bibliotecas de ordenação neste passo do algoritmo.

Em nosso exemplo, a primeira rodada consiste em duas fusões paralelas, começando as fusões com as splits “mais à esquerda”: Split #1 e Split #2; Split #3 e Split #4.

Split #(1+2)	Split #(3+4)	Split #5
9 10 19 32 34 37 37 45 48 49	5 19 20 22 29 30 31 31 40 45	8 14 35 36 39

A segunda rodada consiste em apenas um *merge*:

Split #(1+2+3+4)	Split #5
5 9 10 19 19 20 22 29 30 31 31 32 34 37 37 40 45 45 48 49	8 14 35 36 39

A última rodada também consiste em apenas um *merge*:

Split #(1+2+3+4+5)
5 8 9 10 14 19 19 20 22 29 30 31 31 32 34 35 36 37 37 39 40 45 45 48 49

Requisitos:

Neste laboratório você deve sempre começar com a divisão “mais à esquerda” e mesclá-la com sua vizinha. Como tal, a divisão “mais à direita” geralmente será a última a ser mesclada. Para nos informar sobre o andamento será necessário imprimir, na tela, algumas informações sobre os *merges* e as rodadas.

Divida a entrada em k segmentos de forma que cada segmento tenha aproximadamente o mesmo tamanho, garantindo que os segmentos tenham o mesmo tamanho e apenas o último segmento seja menor que os demais em 1 unidade.

Inicie uma *thread* (ou processo) por segmento para ordenar cada segmento usando algum algoritmo de ordenação (implementação própria ou biblioteca). Seu programa deve bloquear até que todos as *threads* (ou processo) terminem de ordenar. Cada *thread* (ou processo) deve imprimir na tela uma linha de *status* indicando o número de elementos ordenados (tamanho do segmento) ao final do seu processamento.

Inicie várias rodadas para mesclar os segmentos. Cada rodada deve bloquear até que todas as *threads* (ou processos) na rodada atual terminem antes de avançar. Os encadeamentos devem ser mesclados em uma ordem específica, onde o segmento no início do conjunto de dados é mesclado com o próximo segmento no conjunto de dados. Ou seja, o segmento[0] deve ser mesclado com o segmento[1]. Esse padrão deve seguir tal que segmento[i] seja mesclado com segmento[i + 1] para todos os valores pares de i. Se houver um número ímpar de segmentos, você não deve mesclar o segmento restante na rodada atual.

Após cada rodada, você terá (x/2) ou ((x/2) + 1) segmentos restantes, onde x é o número de segmentos que você teve na rodada anterior. Você deve continuar o processo de mesclagem até restar apenas 1 segmento. Lembre-se, mesclar é uma operação O(n) NÃO O(n*log(n)) e por isso você **não deve** chamar usar bibliotecas de ordenação neste passo do algoritmo.

Um exemplo para o uso de threads e processos, escrito na linguagem Python, na resolução de um problema, o de checar se um número é primo ou não, está disponível no Github[1]:

Graduação:
1. O paralelismo pode ser implementado por meio de processos OU *threads*, a critério do grupo;
2. O trabalho pode ser feito em grupos de 2 ou 3 alunos. Não serão aceitos trabalhos individuais ou em grupos de mais de 3 alunos. Se for necessário, o professor reserva-se no direito de ter que subdividir grupos já existentes;
3. Os grupos poderão implementar os trabalhos usando as linguagens de programação C ou Python;

Pós-Graduação:
1. O paralelismo pode ser implementado por meio de processos E *threads*;
2. O trabalho pode ser feito em grupos de 2 ou 3 alunos. Não serão aceitos trabalhos individuais ou em grupos de mais de 3 alunos. Se for necessário, o professor reserva-se no direito de ter que subdividir grupos já existentes;
3. Os grupos poderão implementar os trabalhos usando as linguagens de programação C ou Python;

1 https://github.com/nerds-ufes/sistdist/tree/main/lab1

Já que estamos usando paralelismo, devemos ver uma aceleração entre usar um único segmento (onde todo o trabalho é feito por uma única *thread*) e usar muitos segmentos. Isso deve ficar mais claro em casos de teste muito grandes.
Como resultado final cada grupo deverá executar o trabalho para pelo menos 5 tamanhos diferentes de lista (use valores grandes para n, faça testes para definir suas escolhas) e use k=1 (sem paralelismo), 2, 4, 8, 16. Plote o resultado do tempo de execução nessas diferentes combinações de k e n e avalie os resultados encontrados.

Entrega:
1. Por meio da Sala de Aula Virtual da disciplina no *Google Classroom*, na atividade correspondente ao Laboratório I. 1 (uma) submissão por grupo é suficiente;
2. Deve-se submeter apenas o *link* para o repositório virtual da atividade (Github, Bitbucket, *Google Colaboratory* ou similares) contendo: i) códigos-fonte; ii) instruções para compilação e execução; iii) relatório técnico (*.pdf* ou *readme*); e iv) vídeo curto (máx 3 min) mostrando 1 execução, resultado e análise;
3. O relatório técnico deverá conter: a metodologia de implementação e testes usada, resultados apresentados sob a forma gráfica, e análise e avaliação dos resultados (Ex: o resultado esperado foi alcançado? Comente!);
4. Esta atividade será executada parcialmente em laboratório no dia 28/03, porém com entrega até o dia 07/04.

Bom trabalho!

Bibliografia:
[1] Python multiprocessing — Process-based parallelism
https://docs.python.org/3/library/multiprocessing.html
[2] Python threading — Thread-based parallelism
https://docs.python.org/3/library/threading.html
[3] POSIX thread (pthread) libraries
https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
[4] c_pthreads [Programação Concorrente]
http://cocic.cm.utfpr.edu.br/progconcorrente/doku.php?id=c_pthreads
[6] ALGORITMOS CONCORRENTES E SUA IMPLEMENTAÇÃO
https://www.prp.unicamp.br/pibic/congressos/xxcongresso/paineis/101918.pdf