LABORATÓRIO DE PROGRAMAÇÃO AVANÇADA
DÉCIMO QUARTO TRABALHO PRÁTICO
- - PTHREADS - -

Neste trabalho vamos exercitar novamente **múltiplas threads**. Vamos utilizar o Posix Threads (**Pthreads**) **necessariamente na linguagem de programação C**. Este trabalho será composto por quatro exercícios. Cada exercício requer a execução da mesma implementação sob diferentes configurações de quantidade de threads e de tamanho dos dados. Estes resultados deverão ser colocados em uma tabela. Com o intuito de minimizar efeitos locais, é exigido que os dados dos relatórios contenham os valores médios de **pelo menos três** execuções em cada configuração.

Este trabalho visa medir o tempo de execução para diversas configurações. Para o cálculo do tempo de execução pode ser usado tanto o **`gettimeofday`** ou **`clock_gettime`**.

A data de entrega máxima será no dia 01 de setembro, até meia-noite. É obrigatório entregar os relatórios e os códigos em um arquivo zipado, que deve ser colocado tanto no `Dropbox` quanto enviado por email. A quantidade de threads e o tamanho do vetor devem ser passados como argumento de linha de comando (argc e argv). Por isso, é importante que seja enviado também um **script** que automatize as várias execuções.

O relatório deve também conter a **arquitetura da máquina** em que você executou os experimentos. No Linux esta informação está no `/proc/cpuinfo`.

EXERCÍCIO 1:

Implemente um programa para **somar** um conjunto de **N** números armazenados em um vetor gerado **aleatoriamente**, utilizando múltiplas threads. As threads vão somar os valores em **uma única variável global protegida por um semáforo MUTEX**.

Antes de criar as threads, você vai "dividir" **o vetor em k partes** e passar os índices de início e fim de cada parte para a respectiva thread. Por exemplo, suponha que N = 2000 e k = 4, nesse caso, cada thread vai ficar responsável por somar 500 valores. A thread[0] vai receber os valores 0 (início) e 499 (fim); thread[1] vai receber os valores 500 (início) e 999 (fim); thread[2] vai receber os valores 1000 (início) e 1499 (fim); e thread[3] vai receber os valores 1500 (início) e 1999 (fim).

Escreva uma **relatório** que avalie o tempo de execução para os seguintes valores de N: 1024, 2048, 4096, 8192, 16384, 32768, 65536; e para os seguintes valores de k: 1, 2, 4, 8, 16, 32 e 64. Quer dizer, avalie N=1024 e k=1, depois avalie N=1024 e k=2, depois avalie N=1024 e k=4; e assim por diante. O relatório deve ter mais ou menos o seguinte formato:

	1024	2048	4096	8192	16384	32768	65536
1							
2							
4							
8							
16							

32							
64							

A estratégia que usa k=1 é simplesmente criar uma única thread. A razão é ser mais "justo" na comparação dos tempos com as outras quantidades de threads.

Plote sete **gráficos em linhas**, um para cada tamanho do dado (1024, 2048..., 65536), onde a abscissa é a quantidade de threads (1, 2, 4, ..., 64) e a ordenada é o tempo de execução. **Avalie se é possível colocar os sete gráficos em um único gráfico** (às vezes deve-se usar a escala **logarítmica**). Se conseguir, não esqueça de incluir a legenda.

EXERCÍCIO 2:

Da mesma forma que o exercício anterior, implemente um programa para **somar** um conjunto de **N** números armazenados em um vetor gerado **aleatoriamente**, utilizando múltiplas threads. Dessa vez, parecido com o que foi feito no TP13, as threads vão somar os valores e **armazenar diretamente em um outro vetor** `soma_parcial` de **dimensão k**, que sera usado para armazenar os k valores parciais da soma.

A função main deve somar os valores do vetor `soma_parcial`. No cálculo do tempo de execução, inclua o tempo de cálculo do valor da soma final sobre o vetor `soma_parcial`.

As configurações da quantidade de dados e número de threads são das mesmas do exercício 1, isto é, o relatório deve estar no mesmo formato de tabela.

Plote sete **gráficos em linhas**, um para cada tamanho do dado (1024, 2048..., 65536), onde a abscissa é a quantidade de threads (1, 2, 4, ..., 64) e a ordenada é o tempo de execução. **Avalie se é possível colocar os sete gráficos em um único gráfico** (às vezes deve-se usar a escala **logarítmica**). Se conseguir, não esqueça de incluir a legenda.

EXERCÍCIO 3:

Altere a implementação do exercício 2 no sentido de que, agora, os valores são somados em **uma variável local** e só após o loop é que esta variável local deverá ser movida para o vetor `soma_parcial` (dimensão k).

Da mesma forma que o exercício 2, a função main deve somar os valores do vetor `soma_parcial`. No cálculo do tempo de execução, inclua o tempo de cálculo do valor da soma final sobre o vetor `soma_parcial`.

As configurações da quantidade de dados e número de threads são das mesmas do exercício 1, isto é, o relatório deve estar no mesmo formato de tabela.

Plote sete **gráficos em linhas**, um para cada tamanho do dado (1024, 2048..., 65536), onde a abscissa é a quantidade de threads (1, 2, 4, ..., 64) e a ordenada é o tempo de execução. **Avalie se é possível colocar os sete gráficos em um único gráfico** (às vezes deve-se usar a escala **logarítmica**). Se conseguir, não esqueça de incluir a legenda.

Adicione no relatório uma comparação/comentários sobre os resultados dos três exercícios.

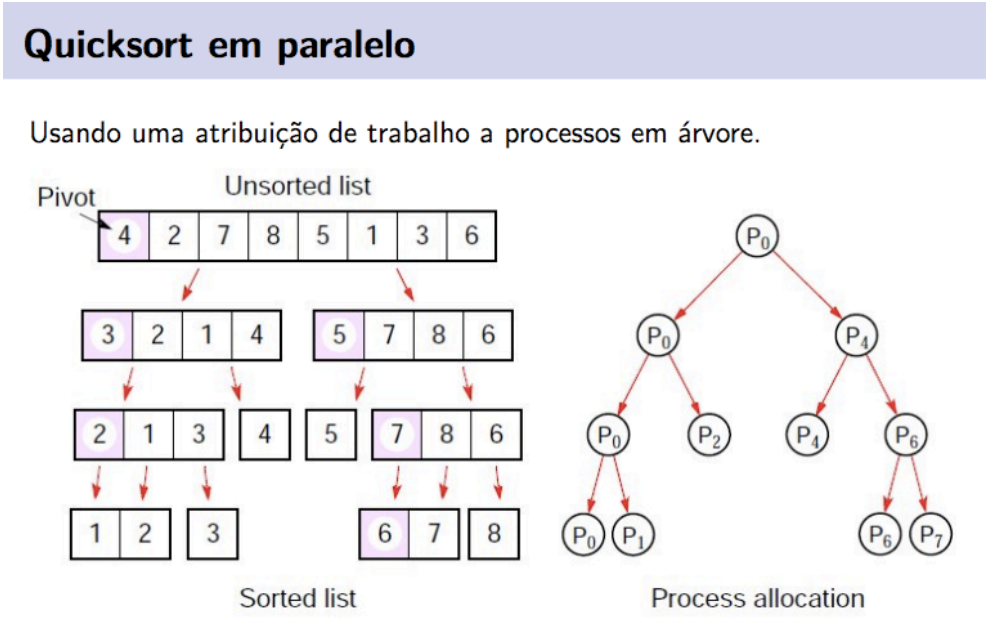
EXERCÍCIO 4:

Implemente um programa para **ordenar** um conjunto de **N** números armazenados em um vetor gerado **aleatoriamente**, usando a estratégia **Quicksort Sequencial**. Escreva uma **relatório** que avalie o tempo de execução para os seguintes valores de N: 512, 1024, 2048, 4096, 8192, 16384.

EXERCÍCIO 5:

Implemente um programa para **ordenar** um conjunto de **N** números armazenados em um vetor gerado **aleatoriamente**, usando a estratégia **Quicksort Paralelo**. Apesar de haver estratégias mais eficientes, esta implementação será simples (*naive*).

Uma maneira de paralelizar o quicksort é executá-lo inicialmente em uma única thread. Quando o algoritmo vai fazer as chamadas recursivas, deve-se atribuir SOMENTE um dos subproblemas para outra thread. O algoritmo termina quando o vetor não pode ser mais particionado. Esta formulação paralela do quicksort **usa n threads para ordenar n elementos**. **Veja na figura abaixo**.



Um problema dessa implementação é que o particionamento continua sendo feito de forma sequencial. O tratamento desta situação está fora do escopo deste exercício.

Escreva uma **relatório** que avalie o tempo de execução para os seguintes valores de N: 512, 1024, 2048, 4096, 8192, 16384, usando a implementação do quicksort paralelo.

Plote um **gráfico** (pode ser **em linhas**) que compare os tempos de execução do quicksort sequencial com o quicksort paralelo. Neste caso, a abscissa é a tamanho do dado (512, 1024, 2048, 4096, 8192, 16384) e a ordenada é o tempo de execução. Portanto, o gráfico deve conter duas séries: quicksort sequencial; e quicksort paralelo. Às vezes pode ser necessário usar a escala **logarítmica**. Adicione no relatório uma **comparação/comentários** com os resultados do exercício 4.