Introdução à Programação Multithread com PThreads

OBJETIVOS

- Introduzir o conceito de programação concorrente com múltiplas threads e seções críticas a serem tratadas.
- Analisar comparativamente o desempenho do algoritmo com diferentes números de threads.

GRUPOS

Deverão ser formados grupos com 3 alunos, a serem escolhidos livremente. No caso do número de alunos da turma não ser múltiplo de 3, a decisão deverá ficar a critério do professor.

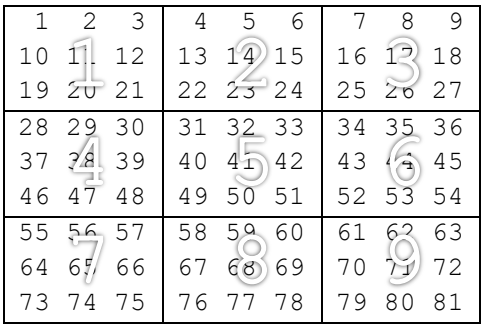
PONTUAÇÃO

O referido trabalho será avaliado de 0 a 100 e corresponderá a 20% da nota semestral.

IMPLEMENTAÇÃO

O objetivo final do algoritmo é: dada uma matriz de números naturais aleatórios (intervalo 0 a 29999) contabilizar quantos números primos existem e o tempo necessário para isso. No entanto, isso será feito de duas formas:

- De modo serial, ou seja, a contagem dos primos será feita um a um, um após o outro. Esse será o seu tempo de referência.
- De modo paralelo. Para tanto, o trabalho de verificar cada número e se for primo contabilizá-lo consistirá na subdivisão da matriz em "macroblocos" (submatrizes), sem qualquer cópia, baseando-se apenas nos índices. Como no exemplo abaixo:



Ou seja, a matriz acima é 9 x 9 e cada macrobloco é composto por 9 elementos. O macrobloco 1 vai da coluna 0 a 2 e da linha 0 a 2, e assim sucessivamente. Os macroblocos serão as unidades de trabalho de cada thread. Atenção: **Nem a matriz nem os macroblocos deverão ser obrigatoriamente quadradas**. A única exigência é que todos os macroblocos tenham o mesmo tamanho. Além disso, você deve encontrar alguma forma de PARAMETRIZAR essa divisão (usando a diretiva #define, por exemplo) a fim de poder efetuar os testes para diferentes tamanhos de macroblocos. **Os macroblocos terão tamanhos que podem variar de desde um <u>único elemento</u> até a <u>matriz toda</u>** (equivalente ao caso serial).

Dito qual é o objetivo final da implementação, segue um passo a passo das diretrizes básicas a serem seguidas pelo algoritmo:

1. Geração de uma matriz de números naturais aleatórios (intervalo 0 a 29999) usando uma semente pré-definida no código, a fim de sempre ter a mesma "matriz aleatória" para todos os testes. A geração de números aleatórios em C se dá com o uso das funções srand() e rand(). **Essa matriz e a variável que contabiliza o número de números primos encontrados na matriz deverão ser globais (logo, compartilhadas) e únicas.**
    a. O tamanho da matriz deverá ser consideravelmente grande a fim de que possam ser efetuadas medidas de desempenho consistentes. Para efeito de comparação, num desktop com 8 GB de RAM e CPU core i5 com 4

núcleos reais, a verificação serial de uma matriz de 20000 x 20000 levou aproximadamente 42 segundos. **Atenção**: Muito cuidado na escolha desse tamanho! Nos testes efetuados pelo professor, o processo que efetuou o teste com a matriz desse tamanho ocupou aproximadamente 1,5 GB de RAM. No entanto, visto ter o computador uma quantidade de RAM expressivamente maior que isso, pode-se concluir com alguma certeza que a memória secundaria (HD ou SSD, que é muito mais lento que a RAM, obviamente) não fora usada. A escolha das dimensões da matriz deverá levar em conta a configuração do computador onde os testes serão executados a fim de evitar medidas incorretas devido ao uso da memória virtual. Faça testes de olho no consumo de memória ao rodar o executável (use o Gerenciador de Tarefas, se estiver no Windows) a fim de definir um tamanho razoável para a matriz. Um tamanho razoável é algum que leve a um tempo de busca para o caso serial superior a 20 segundos.

2. **Busca serial** pela quantidade de números primos da matriz gerada acima. Exiba a quantidade de números primos encontrados e o tempo decorrido nessa busca.

3. **Busca paralela** pela quantidade de números primos na **mesma matriz** usada na busca serial. Cabem, aqui, várias observações:

    a. A escolha do número de threads para o teste principal deverá levar em conta o **número de núcleos reais** da CPU que equipa o computador. Se a CPU tiver N ≥ 2 núcleos reais, crie N threads. Se não souber dessa informação, procure saber de antemão.

    b. Caso a CPU possua uma quantidade de núcleos virtuais maior que a de núcleos reais, como é o caso daquelas com o *HyperThreading* da Intel, teste também para essa quantidade de núcleos virtuais (ex: CPU com 2 núcleos reais e 4 virtuais = crie 4 threads). Faça uma análise dessa comparação: Quantidade de Threads igual ao número de **processadores reais** x Quantidade de Threads igual ao número de **processadores virtuais**.

    c. A atribuição de qual macrobloco será processado em cada momento deverá ser da seguinte forma: Suponha que foram criadas 4 threads. A thread 1 deverá começar a busca no macrobloco 1, a thread 2 no macrobloco 2, a thread 3 no macrobloco 3 e a thread 4 no macrobloco 4. Devido à natureza aleatória dos números (consequentemente do tempo de verificação se o número é primo ou não depender da magnitude do número) e do escalonador do sistema operacional, nada se pode afirmar sobre que thread terminará sua busca primeiro. No entanto, digamos que a thread 2 termine sua busca primeiro. Ela deverá: somar o número de primos encontrado à variável global que esteja contabilizando o número total de primos da matriz (ou seja, a thread usará uma variável temporária para contar o número de primos e, após terminada a busca no macrobloco, somará esse valor à variável global) e reiniciar a busca no próximo macrobloco que ainda não foi atribuído a nenhuma thread. **A variável que controla quais macroblocos estão livres/alocados deverá ser global (compartilhada)**.

        i. Exemplo: Thread 2 termina. Logo, ela verificará se o macrobloco 5 já foi atribuído a alguma thread. Se não, ela "marca-o" como já atribuído e reinicia seus trabalhos nele. Caso contrário, busca pelo próximo macrobloco "livre", até que por fim se esgote os macroblocos a serem buscados. Neste ponto, a thread termina e a thread principal fica esperando que as demais threads terminem.

    d. **ATENÇÃO**: É proibido criar um vetor para armazenar o número de primos encontrados em cada macrobloco e depois somá-los. **Como já mencionado, as variáveis que armazenam o número de primos total, a que controla a alocação do macroblocos e matriz principal deverão ser globais e o acesso compartilhado deverá ser controlado**.

    e. Uma vez criadas as N threads, outras não deverão ser criadas. As mesmas threads deverão "buscar trabalho" em outros macroblocos livres, conforme mencionado acima.

    f. Faça testes com macroblocos de tamanhos diferentes, entre os extremos: um único elemento e a matriz toda. Anote esses valores, pois serão usados no relatório.

    g. **Não fuja das seções críticas**. Trate-as!

    h. Forneça uma forma prática e fácil de testar o código em *single-thread*, *multi-thread* ou ambos. O professor precisará fazer todos esses testes ao corrigir seu trabalho.

RELATÓRIO

Seja criativo! Monte tabelas, gráficos, etc. Avalie/critique o máximo possível os resultados dos testes. Faça conjecturas e verifique, com testes, se elas estão certas.

Elabore gráficos relativo ao **tempo de processamento** versus diferentes **números de threads** e **tamanhos de macroblocos**, conforme citado anteriormente. Você encontrará resultados bem interessantes quando o tamanho dos macroblocos for muito grande ou muito pequenos! ☺ Igualmente, verifique se há algum ganho/perda quando o número de threads for maior que o número de núcleos da CPU.

Testes e Análises obrigatórias

- Mensure o *speedup* ao rodar em múltiplas threads. Analise se esse resultado está coerente com o que a Lei de Amdahl prevê. Mas atenção a diferença entre núcleos reais e núcleos virtuais (ou lógicos)! Tanto processadores da AMD quanto da Intel podem implementar técnicas de SMT (Simultaneos Multi-Threading). No caso da Intel, esse recurso é denominado *HyperThreading*. Nestes casos, mensure o quanto esse recurso mudou ou não a velocidade dos cálculos da maneira mais isolada possível (alguns computadores permitem desabilitar esse recurso na BIOS/UEFI).
- Teste em dois computadores bem diferentes, tal qual feito em AOC. Compare o IPC de ambos.
- Aumente **muito** (algumas centenas ou mais) o número de threads a fim de que o overhead possa realmente ficar crítico e analise os resultados.
- Remova os mutexes que protegem as RC's. Isso mesmo... remova as proteções temporariamente, rode o programa e observe os resultados.

CONCLUSÃO

Pronto, chegou a hora de você responder da maneira mais abrangente possível a simples questão:

O que você pode aprender com esse trabalho?

Capriche na elaboração da resposta.

ENTREGA

A data limite para entrega do trabalho, via email do professor, é o dia **30/10 (quarta-feira)** até as **23:59**.

O trabalho deverá ser entregue comprimido em zip/rar (Relatório em PDF + Arquivo fonte único em .c), com a nomenclatura **[SO 2019-2] Trabalho 1 - Aluno1, Aluno2, Aluno3** tanto para o arquivo como para o assunto do email.

Siga estritamente as regras acima, ou seu trabalho poderá ser desclassificado. Principalmente as normas de nomenclatura. O objetivo é facilitar a correção/análise por parte do professor.

Bom Trabalho!