O projeto valerá de 0 a 10 e será levada em consideração a organização dos códigos de cada solução. Escreva códigos legíveis e com comentários sobre cada decisão importante feita nos algoritmos. As questões deverão ser implementadas em pthreads e utilizando o sistema operacional Linux. Ademais, caso uma questão necessite de arquivos, a equipe deverá disponibilizar arquivos exemplos de entrada. O não cumprimento das regras acarretará em perdas de pontos na nota final.

1. Você deverá criar um programa usando pthreads, no qual *n* threads deverão incrementar um contador global até o número 1.000.000. A thread que alcançar este valor deverá imprimir que o valor foi alcançado e todas as threads deverão finalizar a execução.

2 .Para uma nova política de armazenamento de dados, uma empresa solicitou a eliminação de seus usuários considerados inativos do banco de dados de seu sistema. Para isso, as informações de cada usuário estão inclusas em arquivos de tamanhos indeterminados. Faça um programa que receba, respectivamente, um número ***N*** de usuários, um número ***A*** ≥ 2 de arquivos e um número ***T ≤ ⌊A/2⌋*** de *threads* para o acesso aos arquivos. Todos os arquivos devem ser analisados e cada um é nomeado "bancoX.txt", ***onde 1 ≤ X ≤ A.*** Cada linha do arquivo deverá conter os dados de apenas um usuário, que consiste em, nesta ordem:
nome: Nome do usuário
id: Identificação exclusiva do usuário
ultimo_Acesso: A quantidade de dias no qual o usuário está inativo
pontuacao: A pontuação de atividade do usuário. 0 ≤pontuacao ≤1

Após a leitura de todos os arquivos e o armazenamento dos dados em arrays, deve-se calcular a média do grau de inatividade do banco, cuja média igual a:

$$\frac{\sum_{i=1}^{N}\left(\frac{ultimo\_Acesso(i)}{pontuacao^2(i)}\right)}{N}$$

onde *i é* um usuário. Com isso, os usuários em que o seu grau de inatividade é, no mínimo, duas vezes maior que a média devem ter os seus dados apagados de seu respectivo arquivo e os seus nomes impressos no terminal (não precisa ser em uma ordem específica).

Para as leituras e a escrita dos arquivos, ***é necessário que apenas uma thread acesse um arquivo de cada vez***, com o intuito de garantir a exclusão mútua e a prevenção de deadlocks. Além disso, como o número de threads é fornecido pelo usuário, não podemos determinar qual thread acessa qual arquivo, assim, o acesso a qualquer arquivo pode ser feito por qualquer thread e, quando uma thread acabar de acessar um arquivo, se ainda existir algum arquivo não acessado, a thread deve acessar este arquivo. Ademais, cada usuário aparece pelo menos uma vez em cada arquivo e somente em um arquivo.

Clique aqui para abrir um exemplo dos arquivos antes e depois da execução do programa

3. O método de Jacobi é uma técnica representativa para solucionar sistemas de equações lineares (SEL). Um sistema de equações lineares possui o seguinte formato : ***A*x** = b, no qual

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Ex:
$2x_1 + x_2 = 11$
$5x_1 + 7x_2 = 13$

$$A = \begin{bmatrix} 2 & 1 \\ 5 & 7 \end{bmatrix}, \; b = \begin{bmatrix} 11 \\ 13 \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

O método método de Jacobi assume uma solução inicial para as incógnitas (x_i) e o resultado é refinado durante P iterações , usando o algoritmo abaixo:

```
while(k < P)
begin
```

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

```
      k = k + 1;
end
```

Por exemplo, assumindo o SEL apresentado anteriormente, P=10, e $x1^{(0)}$=1 e $x2^{(0)}$=1:
while(k < 10)

begin

$x_1^{(k+1)}$ =1/2 * (11 – $x_2^{(k)}$)

$x_2^{(k+1)}$ = 1/7 * (13 – 5$x_1^{(k)}$)

k = k+1;

end

Exemplo de execução

k=0

$x_1^{(1)}$ =1/2 * (11 – $x_2^{(0)}$) = 1/2 *(11-1) = 5

$x_2^{(1)}$ = 1/7 * (13 – 5$x_1^{(0)}$) = 1/7 *(13-5 * 1) = 1.1428

k=1

$x_1^{(2)}$ =1/2 * (11 –1.1428)

$x_2^{(2)}$ = 1/7 * (13 – 5 * 5)

...

Nesta questão, o objetivo é quebrar a execução seqüencial em threads, na qual o valor de cada incógnita x_i pode ser calculado de forma concorrente em relação às demais incógnitas (Ex: $x_1^{(k+1)}$ pode ser calculada ao mesmo tempo que $x_2^{(k+1)}$). A quantidade de threads a serem criadas vai depender de um parâmetro ***N*** passado pelo usuário durante a execução do programa, e ***N*** deverá ser equivalente à quantidade de processadores (ou núcleos) que a máquina possuir. No início do programa, as ***N*** threads deverão ser criadas, ***I*** incógnitas igualmente associadas para thread, e nenhuma *thread* poderá ser instanciada durante a execução do algoritmo. Dependendo do número ***N*** de threads, alguma thread poderá ficar com menos incógnitas assoicadas à ela.

Para facilitar a construção do programa e a entrada de dados, as matrizes não precisam ser lidas do teclado, ou seja, podem ser inicializadas diretamente dentro do programa (ex: inicialização estática de vetores). Ademais, **os valores iniciais de $x_i^{(0)}$ deverão ser iguais a 1,** e adote um mecanismo (ex: *barriers)* para sincronizar as threads depois de cada iteração.
Faça a experimentação executando o programa em uma máquina com 4 processadores/núcleos, demostrando a melhoria da execução do programa com 1, 2 e 4 threads.

ATENÇÃO: apesar de $x_1^{(k+1)}$ pode ser calculada ao mesmo tempo que $x_2^{(k+1)}$, $x_i^{(k+2)}$ só poderão ser calculadas quando todas incógnitas $x_i^{(k+1)}$ forem calculadas. *Barriers são uma excelente ferramenta para essa questão.*

4. Dado um intervalo de inteiros positivos, seu programa deve, operando com 4 threads, informar:

a) Quantos números do intervalo são primos.
b) Qual a maior quantia de divisores.
c) Qual o número do intervalo que contém mais divisores.

Em caso de empate na quantia de divisores, deve-se escolher o menor número entre os empatados.

EXEMPLO

Entrada: 1 10

Saída: 4 4 6

5.Uma matriz esparsa é uma matriz em que a maioria de seus elementos tem valor zero. Matrizes desse tipo aparecem frequentemente em aplicações científicas. Ao contrário de uma matriz densa, em que é necessário levar em consideração todos os valores da matriz, em uma matriz esparsa podemos nos aproveitar da estrutura da matriz para acelerar a computação de resultados. Muitas vezes, faz-se mesmo necessário aproveitar essa estrutura, ao se lidar com matrizes esparsas de dimensões imensas (da ordem de 10^6 x 10^6) para que a computação termine em um tempo razoável, visto que a multiplicação de matrizes tradicional tem complexidade $O(n^3)$. Nesta questão, você deverá implementar algoritmos para realizar algumas operações comuns sobre matrizes esparsas de números de ponto-flutuante.

Para auxiliar na definição de uma matriz esparsa, definiremos primeiramente um vetor esparso: Um vetor esparso será dado por um vetor de pares ***(índice,valor)***, no qual o primeiro elemento do par indica o índice de um elemento não-zero do vetor, e o segundo elemento indica seu valor. Portanto, o vetor `{0,0,0,1.0,0,2.0}`, em forma de vetor esparso, será representado por `{Par (3,1.0),Par (5,2.0)}` A implementação dos vetores esparsos fica a cargo do aluno. Uma matriz esparsa será dada por um vetor de vetores esparsos.
Portanto, a matriz esparsa a seguir:

```
2.0  -1.0 0.0  0.0
-1.0 2.0  -1.0 0.0
0.0  -1.0 2.0  -1.0
0.0  0.0  -1.0 2.0
```

Seria representada como:

```
{{(0,2.0),(1,-1.0)},
{(0,-1.0),(1,2.0),(2,-1.0)},
{(1,-1.0),(2,2.0),(3,-1.0)},
{(2,-1.0),(3,2.0)}}
```

O aluno deve implementar as seguintes operações sobre matrizes esparsas:

- Multiplicação de uma matriz esparsa por um vetor denso (vetor comum de C)
- Multiplicação de uma matriz esparsa por outra matriz esparsa
- Multiplicação de uma matriz esparsa por uma matriz densa (matriz comum de C)

Todas essas funções devem usar paralelismo. Ex: na multiplicação, cada linha da matriz deve ser gerenciada por uma thread. Portanto, deve-se ter uma thread para cada linha da matriz.
Em sala de aula, foi visto uma abordagem para múltiplicação de matrizes usando múltiplas threads. Você pode adaptá-la para considerar matrizes esparsas.

6. Para facilitar e gerenciar os recursos de um sistema computacional com múltiplos processadores (ou núcleos), você deverá desenvolver uma **API** para tratar requisições de chamadas de funções em threads diferentes. A **API** deverá possuir:

- Uma constante ***N*** que representa a quantidade de processadores ou núcleos do sistema computacional. Consequentemente, ***N*** representará a quantidade máximas de threads em execução;

- Um ***buffer*** que representará uma fila das execuções pendentes de funções;;

- Função **agendarExecucao.** Terá como parâmetros a função a ser executada e os parâmetros desta função em uma *struct*. Para facilitar a explicação, a função a ser executada será chamada de ***funexec***. Assuma que ***funexec*** possui o mesmo formato daquelas para criação de uma thread: um único parâmetro. Isso facilitará a implementação, e o struct deverá ser passado como argumento para ***funexec*** durante a criação da *thread*. A função **agendarExecucao** é não bloqueante, no sentido que o usuário ao chamar esta funcionalidade, a requisição será colocada no ***buffer***, e um ***id*** será passado para o usuário. O ***id*** será utilizado para pegar o resultado após a execução de ***funexec*** e pode ser um número sequencial;

- Thread **despachante.** Esta deverá pegar as requisições do ***buffer***, e gerenciar a execução de ***N*** *threads* responsáveis em executar as funções ***funexecs*****.** Se não tiver requisição no buffer, a *thread* **despachante** dorme. Pelo menos um item no ***buffer,*** faz com que o despachante acorde e coloque a ***funexec*** pra executar. Se por um acaso ***N*** *threads* estejam executando e existem requisições no buffer, somente quando uma thread concluir a execução, uma nova ***funexec*** será executada em uma nova thread. Quando ***funexec*** concluir a execução, seu resultado deverá ser salvo em uma área temporária de armazenamento (ex: um buffer de resultados). O resultado de uma ***funexec*** deverá estar associada ao ***id*** retornado pela função **agendarExecucao. Atenção: esta *thread* é interna da API e escondida do usuário.**

- Função **pegarResultadoExecucao.** Terá como parâmetro o ***id*** retornado pela função **agendarExecucao.** Caso a execução de ***funexec*** não tenha sido concluída ou

executada, o usuário ficará bloqueado até que a execução seja concluída. Caso a execução já tenha terminado, será retornado o resultado da função. Dependendo da velocidade da execução, em muitos casos, os resultados já estarão na área temporária.

A implementação não poderá ter espera ocupada, e os valores a serem retornados pelas funções ***funexec*** podem ser todas do mesmo tipo (ex: números inteiros ou algum outro tipo simples ou composto definido pela equipe). ***funexec*** é um nome utilizado para facilitar a explicação, e diferentes nomes poderão ser utilizados para definir as funções que serão executadas de forma concorrente.
Você deverá utilizar variáveis de condição para evitar a espera ocupada. Lembre-se que essas variáveis precisam ser utilizadas em conjunto com mutexes. Mutexs deverão ser utilizados de forma refinada, no sentido que um recurso não deverá travar outro recurso independente.