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.
Atenção: Usar somente pthreads e Linux.

1. Um sistema de restaurante robotizado contém um dispositivo de atendimento personalizado em cada mesa. Quando um cliente solicita atendimento, um dos robôs se dirige até a mesa, faz o atendimento, e retorna para a base. Assuma a existência de **N** garçons robôs que irão atender **M** mesas, sendo **N <= M**. Os robôs e o dispositivo da mesa são representados por threads. Ao realizar um pedido, o dispositivo da mesa adiciona o pedido em uma fila de tamanho máximo **Q.** Os robôs disponíveis ficam checando se há pedidos na fila. Caso haja, remove os pedidos da fila. Assuma que o dispositivo da mesa executa em um laço infinito, adicionando itens na fila. Similarmente, assuma que os robôs executam em um laço infinito removendo itens na fila.

Dica: o problema possui condição de disputa na fila. Faça o controle adequado para o acesso à estrutura de dado.

2. Em um sistema de controle de falhas, deseja-se fazer o levantamento da quantidade de defeitos de diferentes equipamentos adquiridos pelos clientes de uma empresa. O registro de cada produto vendido encontra-se em diferentes arquivos, os quais foram obtidos dos clientes. Faça um programa que receba um número ***N*** de arquivos, um número ***T <= N*** de *threads* utilizadas para fazer a contagem, e um número ***P*** de produtos. Em seguida, o programa deverá abrir os ***N*** arquivos nomeados ***"x.in"*** no qual ***1 <= x <= N***. Cada arquivo terá 1 produto por linha que será um número ***y | 0 <= y <= P*** em que 1 significa o produto 1 e assim sucessivamente. Cada *thread* deverá pegar um arquivo. Quando uma *thread* concluir a leitura de um arquivo, e houver um arquivo ainda não lido, a *thread* deverá ler algum arquivo pendente. Ao final imprima na tela o total de produtos lidos, e o percentual de cada tipo de produto vendido.

Assumindo o conhecimento prévido da quantidade de threads e arquivos, pode-se definir no início do programa quais arquivos a serem tratados por cada thread. Uma outra alternativa ler os arquivos sob demanda, a partir do momento que uma thread termina a leitura de um arquivo, pega qualquer outro não lido dinamicamente.

Ademais, deve-se garantir a exclusão múltua ao alterar o array que guardará a quantidade de cada produto. ***Porém, você deverá assumir uma implementação refinada.*** *Uma implementação refinada garante a exclusão mútua separada para cada posição do array. Mais especificamente, enquanto um produto está sendo contabilizado para um candidato* ***x*** *e*

*modificando o array na respectiva posição, uma outra thread pode modificar o array em uma posição **y** que representação outro produto. Ou seja, se o array de produtos possui tamanho 10, haverá um outro array de 10 mutex, um para cada posição do vetor de produtos. Ao ler um arquivo e detectar um venda para o produto y, a thread trava o mutex relativo à posição y, incrementa a quantidade desse tipo de produto, e destrava o mutex na posição y. Obviamente, se mais de uma thread quiser modificar a mesma posição do array de produtos simultaneamente, somente 1 terá acesso, e as outras estarão bloqueadas. O mutex garantirá a exclusão mútua na posição.*

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 : $A\mathbf{x} = \mathrm{b}$, 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

$x_1^{(k+1)} = 1/2 * (11 - x_2^{(k)})$
$x_2^{(k+1)} = 1/7 * (13 - 5x_1^{(k)})$
k = k+1;
end

Exemplo de execução
k=0
$x_1^{(1)} = 1/2 * (11 - x_2^{(0)}) = 1/2 *(11-1) = 5$
$x_2^{(1)} = 1/7 * (13 - 5x_1^{(0)}) = 1/7 *(13-5 * 1) = 1.1428$

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 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. Um quadrado mágico é uma matriz quadrada em que todas as linhas, colunas e diagonais têm a mesma soma. Faça um programa usando pthreads para verificar se a matriz quadrada é um quadrado mágico. Crie uma constante para que o programa assuma **N** para a verificação do quadrado mágico.

5. Um dos algoritmos mais conhecidos na computação é o Merge Sort, o qual é um algoritmo de ordenação por comparação do tipo dividir-para-conquistar.

https://pt.wikipedia.org/wiki/Merge_sort

```
sort(l, r){
        printf("Ordenando [%d, %d]\n", l, r);
        if(l == r) return;
        m = (l+r)/2
        sort(l, m)
        sort(m+1, r)
        merge(l, m, r)
        printf("Ordenado [%d, %d]\n", l, r);
}
```

Uma coisa a se notar é que enquanto o problema do lado esquerdo não for resolvido, o algoritmo **não** começará a resolver o problema do lado direito.

Como o problema da esquerda e o da direita operam em intervalos disjuntos, uma boa otimização seria resolvê-los em paralelo.

Implemente esta otimização usando pthreads, imprimido o vetor final ordenado.

6.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:

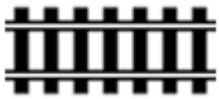
- Multiplicação de uma matriz esparsa por um vetor denso (vetor comum de C)
- Multiplicação de uma matriz esparsa por outra matriz esparsa
- Multiplicação de uma matriz esparsa por uma matriz densa (matriz comum de C)

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.

7. Você deverá implementar um sistema computacional de controle de ferrovias usando C e *pThreads*. A ferrovia é composta por 5 interseções, as quais só permitem **2 trens simultaneamente**. Todavia, um trem passando em uma interseção não deve afetar (bloquear) o andamento dos outros trens em outras interseções. Uma interseção é representada por uma variável inteira. Um trem passando pela interseção (com menos de 2 trens) deverá modificar a variável contador da interseção indicando a quantidade de trens nesta, e esperar 500 milissegundos para indicar o término de sua passagem. Ao concluir a passagem na interseção, deverá decrementar o respectivo contador em uma unidade. O trem, que liberar uma interseção, somente deverá notificar algum trem aguardando a liberação desta interseção.

Por exemplo: Trem 1 e Trem 2 estão na interseção 5, e Trem 3 está aguardando a liberação desta interseção (depois de ter passado pela interseção 4). Trem 1, ao sair da interseção 5, disponibilizará um trilho na interseção, mas só deverá notificar sua saída para o Trem 3 (pois é o único trem aguardando a interseção 5). Os trens deverão ser implementados usando threads, as quais precisam acessar as interseções na sequência 1,2,3,4,5. Ao concluir o percurso, cada trem começará a trafegar novamente a partir do início (ou seja, a partir da interseção 1). Assuma a existência de 10 trens.

1 -2 -3 - 4 - 5



==========>