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. Uma empresa de segurança pediu para você escrever um programa decifrador de senhas, utilizando a técnica da busca por força bruta. Essa busca é feita testando exaustivamente todas as possíveis soluções, até que a senha seja encontrada. Você deve usar T threads, na qual cada thread irá testar um caractere da senha, e caso o comprimento da senha seja maior que T, algumas threads deverão testar mais de um caractere. Quando todos os caracteres forem encontrados, a senha correta deverá ser impressa. A senha é uma string de 10 caracteres, que pode ser inicializada estaticamente.

OBS.: Selecione arbitrariamente algumas senhas, e para cada uma faça comparações entre os tempos de execução para T=1 e para T=5 ou T=10. Adicionalmente, todas threads deverão ser criadas no início do programa e, durante a execução, nenhuma outra thread poderá ser executada.

2. Como visto em sala de aula, um *deadlock* pode ser detectado usando um grafo de alocação de recursos. Neste caso, uma busca em profundidade é realizada a partir de cada nó (recurso ou processo), a fim de encontrar um ciclo (*deadlock*). Implemente o algoritmo apresentado em sala de aula usando múltiplas *threads*.

Dica: coloque cada thread para realizar a busca a partir de um nó inicial diferente.

3. Um sistema gerenciamento de banco de dados (SGBD) comumente precisa lidar com várias operações de leituras e escritas concorrentes. Neste contexto, podemos classificar as threads como leitoras e escritoras. Assuma que enquanto o banco de dados está sendo atualizado devido a uma operação de escrita (uma escritora), as threads leitoras precisam ser proibidas em realizar leitura no banco de dados. Isso é necessário para evitar que uma leitora interrompa uma modificação em progresso ou leia um dado inconsistente ou inválido.

Você deverá implementar um programa usando pthreads, considerando **N** threads leitoras e **M** threads escritoras. A base de dados compartilhada (região crítica) deverá ser um array, e threads escritoras deverão continuamente (em um laço infinito) escrever no array em qualquer posição. Similarmente, as threads leitoras deverão ler dados (de forma contínua) de qualquer posição do array. As seguintes restrições deverão ser implementadas:

1. As threads leitoras podem simultaneamente acessar a região crítica (array). Ou seja, uma thread leitora não bloqueia outra thread leitora;
2. Threads escritoras precisam ter acesso exclusivo à região crítica. Ou seja, a manipulação deve ser feita usando exclusão mútua. Ao entrar na região crítica, uma

thread escritora deverá bloquear todas as outras threads escritoras e threads leitoras que desejarem acessar o recurso compartilhado.

Dica: *Você deverá usar mutex e variáveis de condição.*

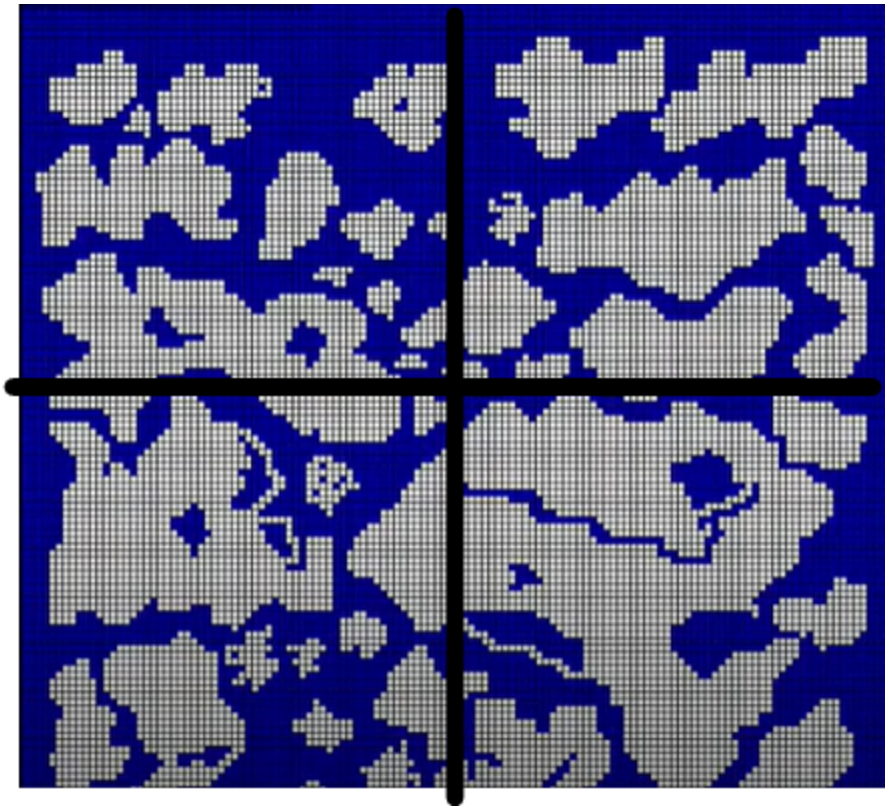
4. A marinha recebeu um mapa em formato de matriz com 0 e 1, de tal forma que:
0: água
1: terra

Sua equipe deverá implementar um programa em *pthreads* para identificar a quantidade de ilhas no mapa. De forma geral, se duas posições de terra são adjacentes na vertical, na horizontal, ou na diagonal, elas são da mesma ilha.

Uma forma de implementação é utilizando a estrutura *Disjoint Sets* (veja capítulo 21 do livro Algoritmos: Teoria e Prática de Cormen et al). Percorrendo a matriz, toda vez que encontrar uma entrada de terra (1), faça *Union* da posição com todas as posições de terra adjacentes. Ao final da execução do algoritmo, conta-se a quantidade de conjuntos disjuntos. Este é o número de ilhas.

O algoritmo deve receber uma matriz como entrada e um inteiro **N** que representa o número de *threads* a serem criadas e utilizadas para resolver o problema..

Dica: (I) divida as threads por regiões disjuntas do mapa, de tal forma que uma *thread* nunca olhe a região da outra. No final, faça o *Union das* interseções dos territórios das threads, ou seja, analisando as fronteiras (linha preta na figura) e a existência de território.. Outra opção é não se importar com a possibilidade das *threads* acessarem uma mesma região. Entretanto, o programa deverá controlar a condição de disputa na criação dos conjuntos disjuntos.



5.Implemente um programa em pthreads que encontre todos os número primos menores que um natural N, usando o crivo de Eratóstenes (versão simplificada):

1. Crie um array de bool com tamanho **N** com todos os elementos sendo *true* (ou 1)
2. Ignore a posição 0 e 1 do array (não são considerados primos) : array[0] = array[1] = false (ou 0);
3. Comece com o primeiro primo: 2;
4. Elimine todos os múltiplos do primo escolhido que estão no array: array[múltiplo] = False;
5. Pegue o primeiro número não eliminado que é maior que o anterior (ex:3) . Ele também é primo. Em seguida. volte para o passo anterior;
6. se não existir esse próximo primo termine o algoritmo

O usuário deverá informar pelo teclado o número *T* de threads a serem criadas e o número *N*. As *T* threads só poderão ser criadas no começo do programa e só poderão ser encerradas depois que todos os números primos já tenham sido encontrados (imprima eles no terminal). Cada thread deverá fazer a eliminação de múltiplos de primos diferentes (as threads não devem pegar o mesmo primo para eliminar em nenhum momento). Assim que uma thread terminar de eliminar os múltiplos de um número, ela imediatamente deverá testar um outro número.

Ex: Entrada: Número T: 3
Número N: 11

Saída: 2, 3, 5, 7

OBS: caso haja problema com duas threads eliminando um número simultaneamente (ex: thread 2 elimina o 6 e thread 3 elimina o 6 ao mesmo tempo), implemente um mecanismo baseado em mutex que garanta exclusão mútua na hora de escolher o próximo número. Ademais, cuidado com a condição de disputa ao array de booleanos..

6. OMP (Open MP) é uma API para C muito usada para concorrente, permitindo implementações sucintas, mas poderosas. Um simples `#pragma omp parallel` antes de chaves faz com que o respectivo escopo seja repetidos **N** vezes por **N** threads (**N** definido por variavel de ambiente). Adicionalmente, um `#pragma omp parallel for` antes de um *for* faz com que esse laço seja executado concorrentemente, de tal forma que cada iteração só ocorra uma vez, mas que todas ocorram até o final do for em qualquer ordem. Suas implementações são dependentes da máquina, compilador, sistema operacional e outros componentes. Às vezes, OMP é até implementado usando Pthreads. **Sua missão é implementar uma função em C que simule um `#pragma omp parallel for` em Pthread.**

OMP é uma API "inteligente", pois não cria threads desnecessárias, nem destrói e cria outras threads sem motivo. Dentre suas variáveis de ambiente, há uma variavel chamada OMP_NUM_THREADS que será aqui emulada por uma macro de mesmo nome. Usualmente, essa variável tem a mesma quantidade de núcleos que o sistema possui, a fim de usar completamente os recursos da máquina, mas sem gerar *overheads* desnecessários.

Quando um *for* aparece no código, OMP entrega a cada uma das **N** threads uma parcela do trabalho daquele *for*, e espera que todas as **N** threads acabem seu trabalho. Como exemplo temos :

```
#pragma omp for parallel
for(int i = 0 ; i < 10 ; i++ )
{
     CODIGO...
}
```

Se o número de threads a ser criada é 4, OMP distribuiria o trabalho possivelmente assim:

Thread 0: `for( int i = 0; i < 3 ; i++ ) CODIGO...`

Thread 1: `for( int i = 3; i < 6  ; i++ ) CODIGO...`
Thread 2: `for( int i = 6; i < 8  ; i++ ) CODIGO...`
Thread 3: `for( int i = 8; i < 10 ; i++ ) CODIGO...`
Espera as threads acabarem.

Repare que nenhum trabalho é realizado mais de uma vez, e que nenhuma *thread* pegou mais trabalho que a outra de forma considerável. Adicionalmente, nenhum trabalho não solicitado não foi distribuído (como uma implementação errada poderia fazer as Threads 2 e 3 terem também 3 iterações, o que transformaria o for entre 0 e 12! Um absurdo).

No final OMP esperaria que todas as 4 threads acabassem para continuar.

A forma de divisão de trabalho pela OMP não é tão rigido. Sabemos que `#omp for parallel` possui uma cláusula chamada `schedule` (definida por padrão quando omitida) que recebe dois argumentos: o tipo de escalonamento das threads e o tamanho do *chunk_size*. Os tipos de escalonamento são: *static, dynamic, guided, runtime,auto*. O *chunk_size* é um valor entre 1 e o número total de iterações.

Em outras palavras, o *chunk_size* é o quanto as iterações devem estar juntas. Por exemplo, com *chunk_size* igual a 3, as iterações são passadas de 3 em 3 para as *threads*, respeitando o final das iterações. No primeiro exemplo desta questão, o *chunk_size* era igual a 3, mesmo assim as ultimas threads pegaram 2 iterações cada (o mais próximo de 3 possível).

Segue agora as definições dos schedules:

static : Antes de todas as threads rodarem, já é definido quais iterações cada thread vai executar, e não há nenhuma comunicação entre a thread pai e as filhas até elas terminarem. As iterações são passadas em *round-robi*n a cada *chunk_size*. Exemplo : Um *for* entre 0 e 100(exclusivo) com *chunk_size* igual a 10 e 4 threads seria dividido (deterministicamente):

Thread 0: 0-9 , 40-49, 80-84
Thread 1: 10-19, 50-59, 85-89
Thread 2: 20-29, 60-69, 90-94
Thread 3: 30-39, 70-79, 95-99

dynamic: As threads são inicializadas sem iterações definidas e conforme vão necessitando elas pedem à thread pai mais iterações. A thread pai deve passar tantas iterações quanto sejam definidas em *chunk_size* na ordem original desde que não ultrapasse o límite definido no *for*. Caso não haja mais trabalho a thread deve ser fechada. Reparem que condições de corrida são implícitas nesse modelo. Exemplo:

Um *for* entre 0 e 15(exclusivo) com *chunk_size* igual a 2 e 4 threads será executado possivelmente assim:

Thread 0 a 3 são criadas.
Thread 0 pede iterações: Recebe iterações 0-1
Thread 2 pede iterações: Rebece iterações 2-3
Thread 3 pede iterações: Recebe iterações 4-5
Thread 1 pede iterações: Rebece iterações 6-7
Thread 3 pede iterações: Rebece iterações 8-9
Thread 3 pede iterações: Rebece iterações 10-11
Thread 0 pede iterações: Recebe iterações 12-13
Thread 2 pede iterações: Rebece iterações 14
Thread 1 pede iterações: É fechada.
Thread 2 pede iterações: É fechada.
Thread 3 pede iterações: É fechada.
Thread 0 pede iterações: É fechada.

guided: É o mesmo que *dynamic,* mas com tamanho de iterações passadas igualmente para cada *thread*, diminuindo a cada etapa e nunca menor que o *chunk_size* passado como parâmetro. A cada momento que novas iterações sejam executadas pelas threads, o número de iterações a serem repassadas deve ser: (iterações restantes)/(número de threads), mas respeitando o mínimo relativo ao *chunk_size*. Exemplo: Um *for* entre 0 e 15(exclusivo) com chunk_size igual a 2 e 4 threads será executado possivelmente assim:

Thread 0 a 3 são criadas.
Thread 0 pede iterações: Recebe iterações 0-3 (teto de 15/4)
Thread 2 pede iterações: Rebece iterações 4-6 (teto de 11/4)
Thread 3 pede iterações: Recebe iterações 7-8 (teto de 8/4)
Thread 1 pede iterações: Rebece iterações 9-10 (teto de 6/4<chunk_size = 2)
Thread 3 pede iterações: Rebece iterações 11-12 (teto de 4/4<chunk_size = 2)
Thread 3 pede iterações: Rebece iterações 13-14 (teto de 2/4<chunk_size = 2)
Thread 1 pede iterações: É fechada.
Thread 2 pede iterações: É fechada.
Thread 3 pede iterações: É fechada.
Thread 0 pede iterações: É fechada.

runtime: Em tempo de execução OMP vê em suas variaveis de ambiente quais das 3 outras politicas de escalonamento ele deve executar neste *for*. **Não precisa ser implementado nessa questão.**

auto: O mesmo que runtime, mas definido pelo compilador. **Igualmente não precisa ser implementado nessa questão.**

Implemente a função em C com a seguinte assinatura para simular a API OMP:

```
void omp_for( int inicio , int passo , int final , int schedule , int chunk_size , void (*f)(int) );
```

A qual simularia um :

```
#pragma omp for
for(int i = inicio ; i < final ; i += passo )
{
     f(i);
}
```

Repare que `f` é um ponteiro de função e `schedule` deve receber um valor entre vários tipos enumerados ou macros para cada uma das 3 politicas de escalonamento.