INE5645 – PROGRAMAÇAO PARALELA E DISTRIBUÍDA – PROVA 1 – 18/09/2017
ALUNO: _____Prof. Bosco_____

1. **a. (Verdade/Falso)** Threads ou processos são programas sequenciais. Programação concorrente é aquela onde diversos processos/threads cooperam (se cooperam, então se comunicam e precisam se sincronizar) entre si para executar, "simultaneamente, num pseudo-paralelismo", agendado pelo Schedule. (0,25)

   b. (Verdade/Falso) Na programação concorrente, a interação, a comunicação e sincronização correta entre as diferentes tarefas, além da coordenação do acesso concorrente aos recursos computacionais, são as principais questões discutidas durante o desenvolvimento de sistemas concorrentes. (0,25)

2. **(Verdade/Falso)** Para que um programa possa ser executado paralelamente é preciso que não hajam dependências entre os dados a serem processados. Uma instrução B depende de uma instrução A, quando ela necessita do resultado da computação de A para a execução de B. O trecho de código abaixo ilustra uma função onde há dependência de dados. Veja uma função onde há dependência de dados.

   ```
   1. function Dep(a, b) {
   2.     c := a * b;
   3.     d := 2 * c;
   4. }
   ```

   Conforme pode ser observado no trecho de código acima, a instrução da linha 3 depende do conteúdo da variável **c**, o qual só estará disponível após a computação da instrução da linha 2. Portanto, a instrução da linha 3 não pode ser executada enquanto a instrução da linha 2 não for concluída. Desta forma, não é possível executar as instruções das linhas 2, 3, paralelamente. (0,5)

Para que um programa possa ser executado paralelamente é preciso que nao hajam dependencias entre os dados a serem processados. Uma instrução B depende da instrucao A quando ela necessita do resultado da computacao de A para a sua execucao. O trecho de codigo acima ilustra uma funcao onde ha dependencia de dados. Conforme pode ser observado no trecho de codigo acima, a instrucao da linha 3 depende do conteudo da variavel C, o qual so estará disponivel apos a computacao da instrucao da linha 2. Portanto, a instrucao da linha 3 nao pode ser executada enquanto a instrucao da linha 2 nao for concluida. Desta forma, não é possível executar as instruções das linhas 2, 3, paralelamente.

3. **(Verdade/Falso)** O próximo trecho de código ilustra uma função onde não há dependência de dados. Veja uma função onde não há dependência de dados:

```
1. function  NoDep(a, b)  {
2.     c := a * b;
3.     d := 2 * b;
4.     e := a + b;
5. }
```

Neste código as instruções das linhas 2, 3 e 4 são totalmente independentes, pois elas utilizam apenas os valores das variáveis **a** e **b**, os quais estão disponíveis por terem sido passados como argumentos na chamada da função **NoDep**. Desta forma, é possível executar as instruções das linhas 2, 3 e 4 paralelamente. (0,5)

O próximo trecho de codigo acima ilustra uma funcao onde nao ha dependencia de dados. Neste codigo as instrucoes das linhas 2, 3 e 4 sao totalmente independentes, pois elas utilizam apenas os valores das variaveis A e B, os quais estao disponiveis por terem sido passados como argumentos na chamada da funcao *NoDep*. Desta forma, é possível executar as instrucoes das linhas 2, 3 e 4 paralelamente.

4. **(Formas de Escalonamento em Java)** No código seguinte,

```
1. ExecutorService executar_leitor Executors.newFixedThreadPool(4);
2. ScheduledExecutorService executar_escritor =
                                Executors.newScheduledThreadPool(1);

      ..............................
      try
      {
3. executar_leitor(new Leitor( sharedLocation ));
4. executar_escritor.scheduleAtFixedRate(new
                Escritor(sharedLocation),0,5,TimeUnit.MILLISECONDS)
      }
      ..................................
```

Qual a diferença entre **ExecutorService** e **ScheduledExecutorService** (0,25)

Com **ExecutorService** trabalha-se com o tempo default para o esalonamento. Com **ScheduledExecutorService** pode-se alterar este tempo de escalonamento.

5. **(Formas de Escalonamento em Java)** Comente os comandos para escalonamento de threads em Java, conforme o que cada um funciona: (0,75)

// Define um pool de threads de tamanho 5 e escolhe-se a forma de escalonamento ScheduledThreadPoolExecutor.
O código abaixo mostra como você pode obter o executor para o agendamento/escalonamento para um pool de threads de tamanho 5.

ScheduledThreadPoolExecutor fornece métodos para agendar tarefas periódicas. Com o método scheduleWithFixedRate(), Ele tem os mesmos parâmetros que o método scheduledAtFixedRate(), mas há uma diferença que vale a pena notar. No método scheduledAtFixedRate(), o terceiro parâmetro determina o período de tempo entre o início de duas execuções. No método scheduledWithFixedRate(), o parâmetro determina o período de tempo entre o final de uma execução da tarefa e o início da próxima execução. O método **scheduleWithFixedDelay()** executa ações periódicas que se tornam habilitadas, primeiro, após o atraso inicial dado e, posteriormente, com o atraso entre o término de uma execução e o início da próxima.

```
ScheduledThreadPoolExecutor sch = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(5);
```

// Cria threads para agendamento prevendo um atraso no processamento das threads.

```
Runnable delayTask = new Runnable(){
   @Override public void run() {.....; Thread.sleep(10 * 1000);.....;}
```

// Cria e executa uma ação periódica que se torna habilitada, primeiro, após o atraso inicial dado e, posteriormente, com o atraso entre o término de uma execução e o início do próximo.

```
ScheduledFuture<?> delayFuture = sch.scheduleWithFixedDelay( delayTask, 5, 5, TimeUnit.SECONDS);
```

6. Segue exemplo de como paralelizar o cálculo do produto escalar em OpenMP. ---------- (0,5)

```
1. #include
2. #define N 5
3.
4. double x[N], y[N];
5. double res = 0;
6.
```

```
7. #pragma omp parallel for reduction(res:+)
8.    for (i=0;i<N;i++) {
9.     res += x[i]* y[i];
10.  }
11.
12. printf("A soma eh:%lf\n",res);
```

Conforme pode ser observado no trecho de código acima, a linha 7 apresenta as definições em OpenMP para a paralelização do laço compreendido pelas linhas 8, 9 e 10. Ainda, ela define uma restrição com relação à variável **res**, determinando que cada thread terá uma cópia da variável **res**, e que, ao final da execução, ela sofrerá o processo de redução para a operação de soma. No programa acima, será criada uma thread para cada núcleo existente na arquitetura alvo. Supondo que X=[1, 2, 3, 4, 5] e Y=[6, 7, 8, 9, 10], qual será o valor da variável **res**, ao final da operação de reduction?
res = _**130**_____

7. Comente nas linhas de comentário, o que significa cada #pragama omp ... (0,5)

Visão completa do código acima:

```
1. int X = 10;
2. int Y = 0;
3. int Z = 0;
4. omp_set_num_threads(5);
5. #pragma omp parallel for private(Y) shared(X, Z) schedule(static, 10)
6. for (i=0; i<100; i++) {
7. Y += X;
8. #pragma omp master
9. {
10. printf("Somente a thread principal... Y = %d\n", Y);
11. }
12. #pragma omp critical
13. {
14. Z += Y;
15. }
16. Y += X;
17. #pragma omp barrier
18. printf("Agora eu posso continuar...\n");
19. }
```

Explicando:

```
1. int X = 10;
2. int Y = 0;
3. int Z= 0;
```

// A linha 4 define que serao criadas 5 *threads* no momento da execução deste codigo, independentemente do numero de *núcleos* da arquitetura onde será executado o código.

```
4. omp_set_num_threads(5);
```

// A linha 5 define a paralelizacão do laço das linhas 6 a 19, O código define que a variavel *Y* e privada a cada *thread*, que as variaveis *X* e *Z* sao compartilhadas por todas as *threads* e que cada *thread* executa uma carga de 10 iteracoes do laco por vez.

```
5. #pragma omp parallel for private(y) shared(X, Z) schedule(static, 10)
6. for (i=0; i<100;i++) {
7. y += x;
```

//_ O trecho de código compreendido pelas linhas 9 a 11 são executadas apenas pela *thread* principal.

```
8. #pragma omp master
9.  {
10.   printf("Somente a thread principal... Y = %d\n", y);
11. }
```

// Adicionalmente, o trecho de codigo definido pelas linhas 13 a 15 é executado sequencialmente pelas *threads*, uma de cada vez.

```
12. #pragma omp critical
13. {
14.   z += y;
15. }
16. y += x;
```

// Por fim, a linha 17 define uma barreira de sincronizacão a partir do qual a execucão só continua quando todas as *threads* chegarem neste ponto.

17. **#pragma omp barrier**

18. **printf("Agora eu posso continuar...\n");**

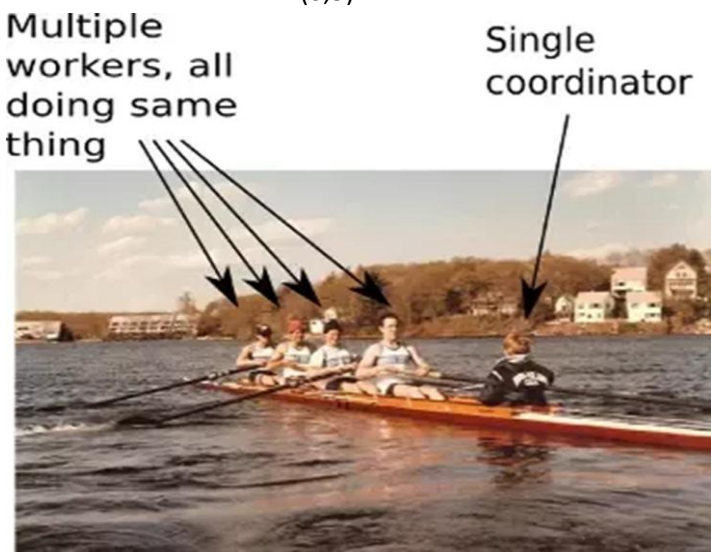
// Aqui o for é fechado.

19. **}**

Texto completo:

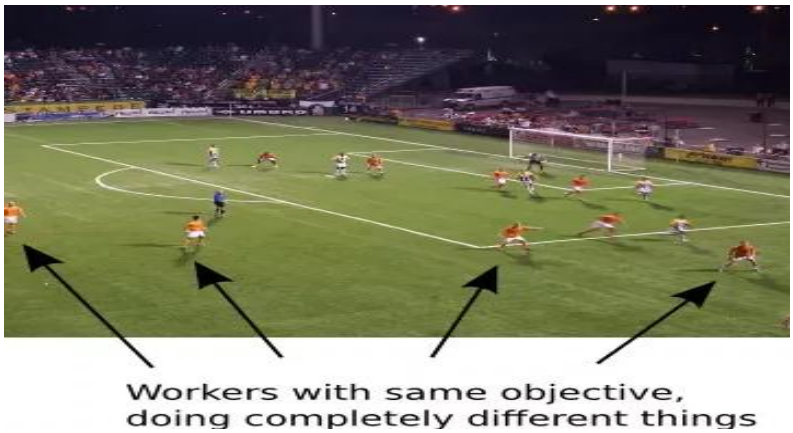
Conforme pode ser observado no codigo acima, a linha 4 define que serão criadas 5 *threads* no momento da execucao deste codigo, independentemente do numero de *núcleos* da arquitetura onde se executa o código. A linha 5 define a paralelizacão do laço das linhas 6 a 19, O código define que a variavel *Y* e privada a cada *thread*, que as variaveis *X* e *Z* sao compartilhadas por todas as *threads* e que cada *thread* executa uma carga de 10 iteracoes do laco por vez. O trecho de codigo compreendido pelas linhas 9 a 11 são executadas apenas pela *thread* principal. Adicionalmente, o trecho de codigo definido pelas linhas 13 a 15 e executado sequencialmente pelas *threads*, uma de cada vez. Por fim, a linha 17 define uma barreira de sincronizacao a partir do qual a execucao só continua quando todas as *threads* chegarem neste ponto.

8. Indique (Paralelismo de Tarefas/Paralelismo de Dados): Observe, em analogia, a figura seguinte: (0,5)



Todas as unidades paralelas compartilham a mesma instrução, mas a realizam em diferentes elementos de dados. Por exemplo, a idéia é que podemos adicionar os arrays A=[1,2,3,4] e B=[5,6,7,8], componente-a-componente, para obter o array S=[6, 8, 10, 12]. Para este exemplo, devemos ter __4___ unidades aritméticas no trabalho paralelo, com todas as somas de componentes podendo compartilhar a mesma instrução (neste caso, a operação "*soma*"). Assim, podemos utilizar esta técnica de programação que pode dividir uma grande quantidade de dados em partes menores que podem ser operadas em paralelo.

9. Observando a figura abaixo, indique o tipo de paralelismo (**Dados**/**Tarefas**), conforme descrito na legenda da figura: (0,5



As unidades paralelas têm instruções distintas, então cada uma delas pode fazer algo diferente em um dado momento.

10. (Verdade/Falso) – A região paralela indicada na figura abaixo representa uma região paralela com seções paralelas concorrentes (????). Explique. (0,5)