

ALUNO \_\_\_\_\_

**1. Sockets - Indicar (Verdade/Falso):**

- (a) (Verdade/Falso) A comunicação entre processos consiste em transmitir uma mensagem entre o *socket* de um processo e o *socket* de outro processo.
- (b) (Verdade/Falso) *Sockets* são abstrações utilizadas nos protocolos de comunicação UDP e TCP, que implementam pontos de interação para comunicação entre processos concorrentes.  
Comunicação em rede entre processos concorrentes ? Não. Comunicação em rede entre processos paralelos em máquinas diferentes.
- (c) (Verdade/Falso) *Sockets* é um conceito de redes, implementados com um par (IP, porta).
- (d) (Verdade/Falso) Para que um processo receba mensagens, seu *socket* deve estar vinculado ao IP de um computador e a uma porta local desse computador em que é executado.
- (e) (Verdade/Falso) As mensagens enviadas para um endereço IP e uma porta específica, só podem ser recebidas por um processo cujo *socket* esteja associada a esse IP e a esse número de porta.
- (f) (Verdade/Falso) Processos não podem usar o mesmo *socket* para enviar e receber mensagens.  
Podem, sim. Veja o exemplo do Socket UDP.  
(Verdade/Falso) Um processo pode fazer uso de uma porta para receber mensagens.
- (g) (Verdade/Falso) Um processo pode compartilhar portas com outros processos no mesmo computador. A frase certa é: Um processo não pode compartilhar ... Para não haver conflito entre aplicações. Cada aplicação tem sua porta estabelecida como um ponto de interação entre a aplicação e a camada de transporte. Por exemplo, a porta padrão para FTP é 21, a do SSH, porta 22, ...
- (h) (Verdade/Falso) Os processos que usam *IP Multicast* são uma exceção, pois esses não compartilham portas.  
Um *DatagramSocket* pode ser usado para enviar e receber pacotes IP Multicast. Exemplo, um *Multicast Server* que usa a porta 5000 para enviar e receber pacotes *IP Multicast*.
- (i) (Verdade/Falso) Qualquer número de processos podem enviar mensagens para a mesma porta. Isto não ocorre em *IP Multicast*.  
Isto ocorre, como no caso dos membros em um *IP Multicast*.

**2. (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, 1, TimeUnit.MILLISECONDS)
}
.....
```

- (a) Qual a diferença entre `ExecutorService` e `ScheduledExecutorService`?  
O `ExecutorService` utiliza o tempo default do processador para escalonar. No `ScheduledExecutorService`, o programador pode estabelecer um tempo apropriado ...
- (b) O que se pode afirmar sobre o **estado das threads** `Leitor` e `Escritor`, quando os trechos das linhas 3 e 4 for executado ?  
Ao serem criadas, as threads leitoras e escritoras passam do estado `NEW THREAD` para o estado de `PRONTAS` para executar.

### 3. Indique (Verdade/Falso)

(a) (Verdade/Falso) Thread é uma divisão de um processo em linhas de execução para processamento paralelo. As linhas de instruções dos processos adquiriram características únicas, que possibilitaram separá-las para execuções em diferentes threads. Essas são executadas em núcleos. Todavia, nem todos os processos são divididos em múltiplas threads, assim como nem todos os processadores são capazes de trabalhar com uma enormidade de threads.

(b) (Verdade/Falso) Os mais recentes processadores vêm com especificações quanto aos núcleos e às threads. Um processo sendo executado dividido nos núcleos. Vamos tomar como exemplo um processador Intel Core i7. Um determinado modelo vem com **quatro núcleos e tem suporte para trabalhar com até oito threads**. Isso quer dizer que esse processador pode trabalhar com quatro threads (códigos indivisíveis) simultaneamente (um em cada núcleo físico) ou com até oito **núcleos lógicos** que executam duas threads em cada núcleo físico, e esses agrupamentos de threads podem ou não ser de um mesmo processo.

#### 4. Componentes de processos e threads

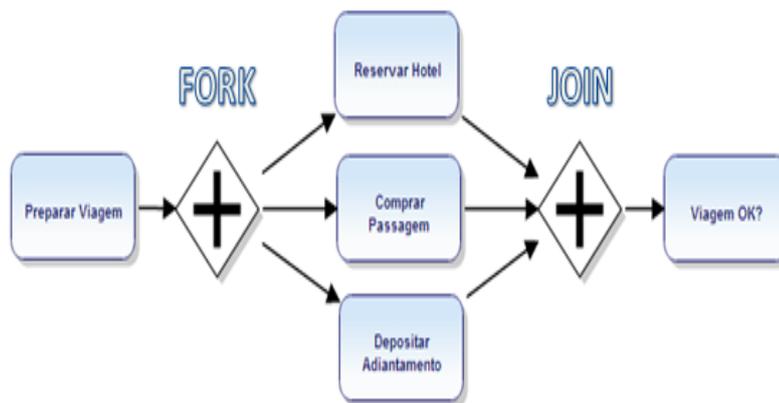
(Verdade/**Falso**) Itens propriedade de **threads** são: (a) Espaço de endereçamento, (b) Variáveis globais, (c) Contador de programa lógico, (d) Registradores, (e) Pilha, (f) Estado, (g) Recursos.

(a) Variáveis globais, (b) Contador de programa lógico, (c) Registradores, (d) Pilha, (e) Estado. Threads usam os recursos do processo onde elas executam.

(Verdade/**Falso**) Itens propriedade de **processos** são: (a) Variáveis globais, (b) Contador de programa lógico, (c) Registradores, (d) Pilha, (e) Estado.

(a) Espaço de endereçamento, (b) Variáveis globais, (c) Contador de programa lógico, (d) Registradores, (e) Pilha, (f) Estado, (g) Recursos. Processos tem espaço de endereçamento. Threads num processo usam o espaço de endereçamento do processo onde elas são executadas.

#### 5. OpenMP (Modelo de Programação)



- (a) (Verdade/**Falso**) – A região paralela indicada representa uma região paralela com seções **paralelas concorrentes**. Neste caso, a região paralela conterá threads de códigos diferentes, implementadas neste caso, em seções paralelas dentro da região paralela, mas que são executadas em paralelo, em núcleos diferentes.

## 6. OpenMp -

### (a) Sobre o código seguinte pode afirmar (SIM/NÃO)

```
#define N 10000;
int i;
#pragma omp parallel
    #pragma omp for
        for (i=0 ; i < 10000 ; i++) {
            calculo();
        }
printf("Terminado")
```

- (1) O **#pragma omp for** faz com que as iterações sejam distribuídas entre threads? **SIM**
- (2) Ao fim do **#pragma omp parallel**, não existe uma barreira implícita de sincronização entre threads no final do loop? **NÃO**, existe.
- (3) **#pragma omp for** pode ser complementado pela *schedule* para especificar como fazer a distribuição da carga do *for* (i=0 ; i < 10000 ; i++), de maneira **static** ou **dynamic**. **SIM**.

## 7. OpenMP

Vimos que o OpenMP irá particionar automaticamente as iterações de um loop *for*. Como podemos otimizar a forma como as iterações de loop são divididas. No código abaixo, indique se o tipo de *schedule* é apropriado. **NÃO É APROPRIADO, o correto é dynamic** porque cada *sleep(i)* tem um tempo de execução diferente dentro do loop.

```
#define THREADS 4
#define N 16
int main ( ) {
    int i;

    #pragma omp parallel for schedule(static) num_threads(THREADS)
    for (i = 0; i < N; i++) {
        /* wait for i seconds */
        sleep(i);

        printf("Thread %d has completed iteration %d.\n",
                omp_get_thread_num( ), i);
    }
    /* all threads done */
    printf("All done!\n");
    return 0;
}
```

8. Indique (Correto/Errado)

( C ) (**Paralelismo de Dados**) Uma técnica de programação que divide uma grande quantidade de dados em partes menores que podem ser operadas em paralelo. A mesma operação é executada simultaneamente em processadores paralelos.

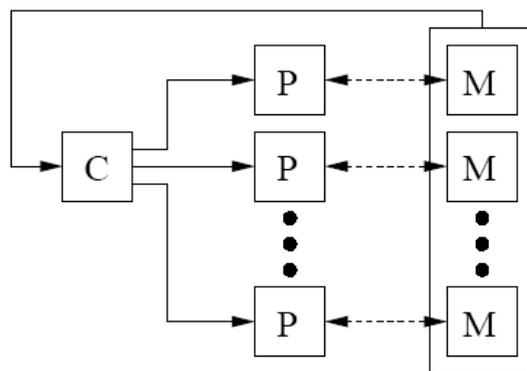
( C ) As **arquiteturas** apropriadas para *Paralelismo de dados* é "SIMD" e para *Paralelismo de tarefas* é "MIMD".

( E ) **SIMD** significa que as unidades paralelas têm instruções distintas, então cada uma delas pode fazer algo diferente em um dado momento.

( C ) **SIMD** significa que todas as unidades paralelas compartilham a mesma instrução, mas a realizam em diferentes elementos de dados.

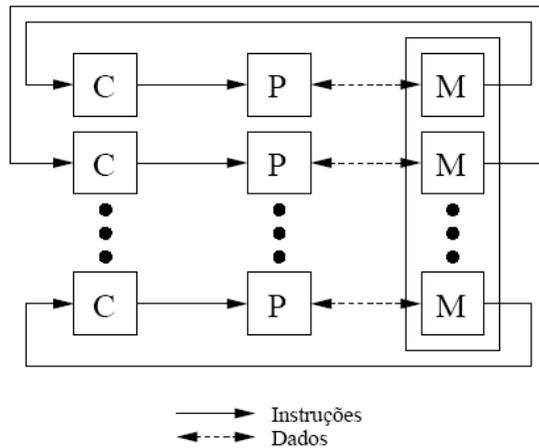
( C ) No **modelo de paralelismo de tarefas**, para cada operação a ser executada, deve ser definido um *kernel* para executar uma determinada operação na arquitetura MIMD.

9. Dadas as figuras abaixo, o que significam, em termos de arquiteturas paralelas, cada uma delas ?



—> Instruções  
->>> Dados

**SIMD (Single Instruction Multiple Data)**



### MIMD (Multiple Instruction Multiple Data)

10. **OpenCL** - Um objeto de programa encapsula o *código-fonte de um kernel*, sendo este identificado no código-fonte por meio da palavra-chave `__kernel`. O que significa, o seguinte código, executado em OpenCL ?

```
__kernel void ArrayDiff (
    __global const int* a,
    __global const int* b,
    __global int* c )
{
    int id = get_global_id(0);
    c[id] = a[id] - b[id];
}
```

Na verdade, no caso em tarefas estão executando em um único núcleo de uma CPU, essas não podem ser executados em paralelo. Neste caso, o aplicativo ou o sistema operacional (*schedule*) devem alternar entre as tarefas, permitindo tanto tempo para executar no núcleo (*pseudo-paralelismo*), existindo o que se chama de concorrência. O paralelismo diz respeito à execução de duas ou mais atividades em paralelo (paralelismo verdadeiro) com o objetivo explícito de aumentar o desempenho no processamento. É o que acontece com os *kernels* do OpenCL.

**O código kernel em representa o código que é paralelizado.**

Veja a explicação. A unidade de execução simultânea no OpenCL C é um item de trabalho (*work-item*). Tal como acontece com o exemplo acima, cada item de trabalho executa o corpo da função do kernel. Em vez de se executar o loop, muitas vezes, num

work-item, mapeia-se uma única iteração do loop para um item de trabalho. Conceitualmente, isso é muito parecido com o paralelismo inerente a uma operação do paralelismo de dados para um loop em um modelo como o OpenMP. Quando um dispositivo OpenCL começa a executar um kernel, ele fornece funções intrínsecas que permitem que um item de trabalho se identifique. No código, a chamada para *get\_global\_id (0)*, permite ao programador fazer uso da posição do work-item corrente, no caso para recuperar o contador do loop.