INE5645 – Prova 1 – Parte A - Paralelização em OpenMP - 12/04/2017

Nome : _____

1. Identifique os seguintes loops

Versão sequencial

```
double res[10000];
for (i=0 ; i < 10000 ; i++)
       calculo_pesado(&res[i]);
```

Versão paralela

```
double res[10000];
#pragma omp parallel for
  for (i=0 ; i < 10000 ; i++)
       calculo_pesado(&res[i]);
```

(a) Explique, brevemente, a diferença de execução existente entre estes dois exemplos de loops.

(b) Se um usuário estiver usando um processador *quad-core*, o que pode você afirmar com relação ao desempenho com a adição de apenas uma linha de código `#pragma omp parallel for` na versão paralela ?

2. Explique a diferença entre as cláusulas (atributos) `shared` e `private` quanto ao compartilhamento dos dados entre threads, usando-se numa diretiva `omp parallel`.

3. Explique como funciona as funções:

(a) `omp_set_num_threads()` e `omp_get_thread_num()`, quando se constrói uma região paralela omp parallel.

(b) Antes da execução do programa, será necessário especificar o número de threads que irão participar da execução paralela. No caso de você não usar `omp_set_num_threads()`, qual a maneira mais fácil de se obter o mesmo resultado num ambiente Linux ?

4. Definição de região paralela. Indique para os itens abaixo (Verdade/Falso).

```
OMP PARALLEL double A[10000];
omp_set_num_threads(4);
#pragma omp parallel
  { int th_id = omp_get_thread_num();
      calculo_pesado(th_id, A);
  }
  printf("Terminado");
```

(a) o início da execução das threads é sinalizado;
(b) as threads são sincronizadas;
(c) o vetor A é compartilhado;
(d) Usou-se funções OpenMP, além das diretivas.

5. Sobre o código seguinte pode afirmar (Verdade/Falso)

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

(a) As iterações são distribuídas entre as threads ?

(b) Tem uma barreira implícita de sincronização entre threads no final do loop ?

(c) `omp for` pode ser complementado pela `schedule` para especificar como fazer a distribuição da carga do `for (i=0 ; i < 10000 ; i++)` ?

6. Mais uma cláusula: `reduction(op : list)`. Usada para operações tipo "all-to-one". A cláusula `reduction` efetua uma operação de redução em uma lista de variáveis (pode ser somente uma veriável). Essas variáveis devem ser escalares, no contexto em que elas participarão.
   - exemplo com a operação de soma: op = '+'
   - cada thread terá uma cópia da(s) variável(is) definidas em 'list' com a devida inicialização;
   - ela efetuará a soma local com sua cópia;
   - ao sair da seção paralela, as somas locais serão automaticamente adicionadas na variável para proporcionar um soma total, ou seja, no final da construção paralela do loop, todas as threads irão adicionar o seu valor do resultado para thread mestre com a variável global.

```
#include <omp.h>
#define NUM_THREADS 4
...
int i, tmp, res = 0;
#pragma omp  parallel  for reduction(+:res)  private(tmp)
{
     for (i=0 ; i< 15 ; i++)
     {
       tmp = Calculo( );
       res += tmp ;
     }
   printf("O resultado vale %d´´, res) ;
}
```

Obs: Os índices de loops sempre são privados.

(a) A variável `res` deve ser `shared` ou `private` ? Explique.

(b) Veja o código acima e explique o que se passa quanto a variável `res`, indicada em `reduction(+:res).` Dê um exemplo de como você entende, para 4 threads e `i` variando de 0 a 15.

7. Distribuição de trabalho com seções paralelas. Pode-se usar `omp sections`, quando não se usam loops:

```
#pragma omp parallel
{
  #pragma omp sections
  {
    Calculo1( );
    #pragma omp section
       Calculo2( );
    #pragma omp section
       Calculo3( );
  }
}
```

(a) As seções são distribuídas entre as threads ou as thread são distribuídas nas seções paralelas ?

(b) Explique como funciona a região paralela e as seções paralelas definidas.

8. Sincronizações - Existem algumas instruções para sincronizar os acessos à memória compartilhada. Procure dar um exemplo de cada caso (a), (b) e (c).

(a) **Seção crítica**

- `#pragma omp critical {. . . }`
- Apenas uma thread pode executar a seção crítica num dado momento.

(b) **Atomicidade**

- `#pragma omp atomic`
- **`<instrução atômica>;`**
- versão “light” da seção crítica.
- funciona apenas para uma próxima instrução de acesso à memória.

(c) **Barreira**

```
#pragma omp barrier
```

- barreiras implícitas nos fins das seções paralelas! master. Pode-se provocar barreiras.

10 . Funções de biblioteca para *run-time* para locks:

São diretivas ?

`omp_lock_t , omp_init_lock, omp_destroy_lock, omp_set_lock, omp_unset_lock, omp_test_lock`

Explique o que ocorre no código acima.

```
#include <stdio.h>
#include <omp.h>

omp_lock_t my_lock;

int main() {
   omp_init_lock(&my_lock);

   #pragma omp parallel num_threads(4)
   {
```

```
      int tid = omp_get_thread_num( );
      int i, j;

      for (i = 0; i < 5; ++i) {
         omp_set_lock(&my_lock);
         printf_s("Thread %d - starting locked region\n", tid);
         printf_s("Thread %d - ending locked region\n", tid);
         omp_unset_lock(&my_lock);
      }
   }

   omp_destroy_lock(&my_lock);
}
```