Computação de Alto Desempenho 2014/15
1º Teste – 17/4/2015
Duração: 2h

O teste é sem consulta e em caso de dúvida na interpretação do enunciado, deve explicitar todos os pressupostos na sua elaboração das suas respostas.

1. Considere um programa sequencial que cifra um ficheiro com L bytes. Esse programa demora 10s a ser executado. Nesse programa há uma parte relacionada com a geração de números aleatórios que não pode ser paralelizada e demora 2s. A restante actividade do programa é embaraçosamente paralela e pode ser executada em N processadores que fazem a leitura do ficheiro em claro e da escrita do ficheiro de cifra em paralelo; os gastos de tempo no lançamento e destruição de processos nas N máquinas consideram-se desprezáveis. Qual é a máxima aceleração (speedup) conseguida? Justifique.

2. Considere o algoritmo de ordenação "odd-even transposition sort" estudado nas aulas, cujo código base é o seguinte:

```
void Odd_even_sort(
        int  a[]  /* in/out */,
        int  n    /* in     */) {
    int phase, i, temp;

    for (phase = 0; phase < n; phase++)
        if (phase % 2 == 0) { /* Even phase */
            for (i = 1; i < n; i += 2)
                if (a[i-1] > a[i]) {
                    temp = a[i];
                    a[i] = a[i-1];
                    a[i-1] = temp;
                }
        } else { /* Odd phase */
            for (i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                    temp = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                }
        }
}  /* Odd_even_sort */
```

a) Aplique a metodologia de Foster ao desenvolvimento de uma solução, considerando o mapeamento para um multiprocessador de memória partilhada com 2 CPUs. Preste especial atenção à minimização dos conflitos nas caches.

b) Implemente o código para a API Pthreads, usando apenas 2 threads.

3. Considere o código seguinte que, dado um conjunto de valores, implementa o cálculo do seu histograma:

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

#define MAXC 256
#define MAXV 2048

int main()
{
       int i, vec[MAXV], h[MAXC];
       time_t tlock;

       srand(time(&tlock));

       for(i=0; i< MAXV; i++)
              vec[i] = rand()%MAXC;

       for(i=0; i<MAXV; i++)
              h[vec[i]] +=1;

       // outro codigo...

       return 0;
}
```

Paralelize este código usando a API C/OpenMP, e do modo mais eficiente possível. Justifique as opções tomadas.

4. Considere o seguinte fragmento de um programa C/OpenMP:

```
int count = 0;
int i;
#pragma omp parallel schedule(dynamic, 100) private(i)
reduction(+:count)
{
    for (i = 0; i < 1000000; i++)
        if( function_with_variable_execution_time(i) )
              count++;
}
printf("count = %d\n", count);
```

Explique em detalhe o efeito da cláusula omp parallel, em especial sobre o tempo de execução deste troço do programa.

5. Diga qual ou quais os problemas que podem surgir nos seguintes códigos em C/OpenMP:

a)

```
# pragm omp parallel
       . . .
# pragma omp atomic
    x += f(y);
# pragma omp critical
    x = g(x);
```

b)

```
# pragm omp parallel
while(1) {
    . . .
    # pragma omp critical
    x = g(my rank);
    . . .
 }
```

6. Pretende-se conhecer a temperatura ao longo de um fio que se supõe, para o efeito, dividido em *N* segmentos. Conhecem-se as temperaturas nas extremidades que são *TA* e *TB* que se mantêm constantes ao longo do tempo. Usa-se um método iterativo para calcular a temperatura no segmento *i* no instante de tempo *j* ($T_{i,t}$) que pode ser resumido da seguinte forma:

$$T_{i,t} = (T_{i-1,t-1} + T_{i+1,t-1})/2.0$$

Admite-se que no instante de tempo inicial (*i=0*):

$T_{0,0}$ = TA

$T_{0,N+1}$ = TB

$T_{k,0}$ = 0

(para todo o *k* diferente de *0* e *N+1*)

Suponha que a temperatura corrente em cada segmento está representada num vector *Temp* com *N+2* posições. *Temp[0]* é sempre igual a *TA* e *Temp[N+1]* é sempre igual a *TB.*

Apresente um programa que, usando o OpenMP, imprime a temperatura em cada segmento a cada *Nit* iterações e para um total de *NT* iterações (*Nit* < *NT*).

Anexo

A. Algumas funções da biblioteca de Pthreads

`int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)`
`int pthread_join (pthread_t thread, void **retval)`
`int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)`
`int pthread_mutex_lock (pthread_mutex_t *mutex)`
`int pthread_mutex_unlock (pthread_mutex_t *mutex)`
`int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)`
`int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
`int pthread_cond_signal(pthread_cond_t *cond)`