Computação de Alto Desempenho 2013/14
1º Teste – 10/4/2014
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 multiprocessador de memória partilhada em que cada CPU tem uma cache L1; a coerência das caches é assegurada por hardware.

a) Explique porque é que é necessário assegurar a coerência das caches.

b) Explique o que é o false sharing e porque é que ele existe no seguinte código OpenMP

```
main {
    int x,y;
     ...
    #pragma omp parallel (shared x,y)
    {
        tid = omp_get_thread_num();
        if (tid == 0) x++;
        if (tid == 1) y++;
     }
}
```

c) Explique como poderia evitar o "false sharing".

2. A aceleração ou *speedup* de um programa executado numa arquitectura com n processadores é o quociente entre o tempo de execução da versão sequencial e o tempo de execução da versão paralela com esses n processadores. De acordo com a lei de Amdalh qual é a máxima aceleração que pode ser conseguida se 80 % da computação de um programa puder ser executada em paralelo de forma embaraçosamente paralela? Explique claramente a forma de obter o resultado – terá uma cotação reduzida se se limitar a escrever uma fórmula e substituir valores.

3. Escreva um programa na linguagem C que soma duas matrizes A e B com NxN elementos obtendo-se a matriz S também com NxN elementos. Cada elemento S[i,j] da matriz S é obtido pela fórmula

$$S[i,j] = A[i,j] + B[i,j]$$

a) Aplique a metodologia de Foster ao desenvolvimento de uma solução. Na fase de "mapping" assuma que faz 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. Escreva na sua folha de teste as partes em falta no código seguinte:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *w (void *arg) {
  // código em falta
}
int main(int argc, char *argv[]){
  pthread_t p1, p2, p3;

  // código em falta

  pthread_create(&p1, NULL, w, ?);       // parâmetro em falta
  pthread_create(&p2, NULL, w, ?);       // parâmetro em falta
  pthread_join(p1, NULL); pthread_join(p2, NULL);
  return 0;
}
```

4. Pretende-se calcular a evolução ao longo do tempo da temperatura de uma chapa metálica com NxN pontos (admita por exemplo N=5000). A chapa está inicialmente a 20 graus centígrados. No instante T=0 é aplicado um *aquecimento constante* que coloca todas as extremidades à temperatura de 200ºC.
Pretende-se obter a distribuição de temperatura na chapa ao longo de 20 unidades de tempo. Num dado instante de tempo T, a nova temperatura do ponto da chapa C[i,j] é dada pelo seguinte algoritmo em pseudo-código C:

```
if ( i== 0) || (j== 0) || ( i==N-1) || (j==N-1) ) {
      temperatura do ponto igual à temperatura no passo anterior (200°C)
}
else {
      temperatura do ponto é igual à média aritmética dos quatro pontos
          vizinhos [i-1,j], [i+1,i], [i,j+1], [i,j-1], que é por sua vez
          multiplicada pela constante CT=0.335
}
```

Implemente o algoritmo acima descrito usando OpenMP. Sugestão: em cada passo da simulação use duas matrizes NxN de números reais, em que uma contém a temperatura no instante *t-1* e a outra guarda os novos valores no instante *t*. Escreva na sua folha de teste as partes em falta no código seguinte:

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

#define END 20
#define N  5000
float t_1[N][N]; // matriz com o estado da chapa em t-1
float t[N][N];   // idem em t

int main (int argc, char *argv[])
{
    int nthreads, tid, t;
    FILE *f;
```

```
    // Partes em falta (por exemplo, ciclo de inicialização da matriz t_1)

    for(t=0; t < END; t++)

    // Partes em falta

    f= fopen("simulacao.bin", "w");
    fwrite(t, sizeof(float), N*N, f);

    return 0;
}
```

5. Considere o problema *N-body* de simulação das posições e velocidades de um conjunto de planetas apresentado nas aulas[1]. O pseudo-código de um programa sequencial que simula o comportamento desse conjunto de planetas pode ser descrito como:

```
Get input data;
for each timestep {
   if (timestep output) Print positions and velocities of particles;
   for each particle q
      Compute total force on q;
   for each particle q
      Compute position and velocity of q;
}
Print positions and velocities of particles;
```

a) Considere o pseudo-código do *algoritmo básico* dado no *anexo A* e que calcula, para cada *timestep*, o total das forças sobre os vários planetas (partículas) *q*, bem como o cálculo das novas posições e velocidades de todos os planetas (partículas) *q*, como resultado das forças calculadas.

No contexto deste algoritmo básico, altere o pseudo-código apresentado em cima usando as funções e directivas do OpenMP disponibilizadas, de modo a paralelizar essa versão sequencial. Justifique as opções que tomar.

b) Considere agora o pseudo-código do *algoritmo reduzido* para cálculo do valor das forças sobre os vários planetas, também apresentado no *anexo B*. Diga se a solução de paralelização que apresentou na alínea a) é adequada no contexto deste algoritmo reduzido, e caso não seja, indique que alteração (ou alterações) teriam de ser realizadas. Justifique a sua resposta.

Anexo

N-body solver para simulação do comportamento de um conjunto de planetas[1]

A. *Algoritmo básic*o

a) Cálculo do total das forças sobre os vários planetas (partículas) *q*:

[1] Descrição em "An Introduction to Parallel Programming", capítulo 6, Peter Pacheco.

```
for each particle q {
   for each particle k != q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
      forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
   }
}
```

b) Cálculo das novas posições e velocidades de todos os planetas (partículas) *q*, como resultado das forças calculadas:

```
for each particle q {
   pos[q][X] += delta_t*vel[q][X];
   pos[q][Y] += delta_t*vel[q][Y];
   vel[q][X] += delta_t/masses[q]*forces[q][X];
   vel[q][Y] += delta_t/masses[q]*forces[q][Y];
}
```

B. *Algoritmo reduzido*.

a) Neste caso, o cálculo das forças sobre os vários planetas (partículas) *q* é optimizado uma vez que, pela terceira lei do movimento de Newton, a acção que um corpo k exerce sobre um corpo q é simétrica da acção que o corpo q exerce sobre o corpo k (mesma dimensão e sinal contrário).

```
for each particle q
   forces[q] = 0;
for each particle q {
   for each particle k > q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
      force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff

      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] -= force_qk[X];
      forces[k][Y] -= force_qk[Y];
   }
}
```

C. 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)`