Universidade Federal do Rio Grande do Norte
Departamento de Engenharia de Computacão e Automação

Programação Concorrente e Distribuida

Terceira Lista de Exercícios

Natal-RN, Brasil
[Novembro de 2017]

Professor

Prof. Samuel Xavier - DCA/UFRN

Aluno

Igor Macedo Silva - Bacharelando em Engenharia de Computação

Sumário

Lista de Figuras

Lista de Tabelas

1 Descrição

Lista da 3a unidade Descrição: Apresentar as respostas a todas as questões de exercício do capítulo 5 do livro texto, com exceção às questões: 5.1, 5.4, 5.7 e 5.11.

Instruções para resolução (LEIAM!):

- Procure responder corretamente todas as questões da lista;
- Suas respostas serão validadas de forma oral por amostragem - geralmente de 2 à 3 defesas orais;
- Se não conseguir responder alguma questão, procure esclarecer as dúvidas em tempo em sala de aula com o professor, pelo SIGAA, com um colega, ou por e-mail. Se necessário, é possível marcar um horário para tirar dúvidas na sala do professor;
- Não serão aceitas respostas "mágicas", ou seja, quando a resposta está na lista entregue mas você não sabe explicar como chegou a ela. Sua nota nesse caso será 0 (zero). Mesmo que não saiba explicar apenas parte da sua resposta;
- Procure entregar a resolução da lista de forma organizada. Isso pode favorecer a sua nota;
- Os códigos dos programas requisitados (ou as partes relevantes) deverão aparecer no corpo da resolução da questão;
- A resolução da lista deverá ser entregue em formato PDF em apenas 1 (um) arquivo;
- O envio da resolução pode ser feito inúmeras vezes. Utilize-se disso para manter sempre uma versão atualizada das suas respostas e evite problemas com o envio próximo ao prazo de submissão devido a instabilidades no SIGAA;
- A lista com o número das questões respondidas deve aparecer na primeira folha da lista. Não será aceita alteração nessa lista.
- Procure preparar sua defesa oral para cada questão. Explicações diretas e sem arrodeios favorecerão a sua nota;
- A defesa deverá ser agendada com antecedência. Para isso, indique por email (samuel@dca.ufrn.br) no mínimo 3 horários dentro dos intervalos disponíveis em pelo menos 3 turnos diferentes. Caso não tenha disponibilidade em 3 turnos diferentes, deverá apresentar uma justificativa.
- Os horários disponíveis serão disponibilizados em uma notícia na turma virtual e serão atualizados a medida que os agendamentos forem sendo fixados.
- A defesa oral leva apenas de 10 a 15 minutos em horários fixados com antecedência. Não será tolerado que o aluno chegue atrasado para a sua prova.

Período: Inicia em 09/11/2017 às 00h00 e finaliza em 24/11/2017 às 23h59

2 Questões

2.1 Questão 5.2

Download `omp_trap_1.c` from the book's website, and delete the `critical` directive. Now compile and run the program with more and more threads and larger and larger values of n. How many threads and how many trapezoids does it take before the result is incorrect?

	Tamanho de N			
threads	256	1024	16384	262144
1	3.33001853942871e+05	3.33000115871429e+05	3.33000000452623e+05	3.33000000001810e+05
16	3.33001853942871e+05	3.33000115871429e+05	3.33000000452623e+05	3.33000000001772e+05
256	3.33001853942871e+05	3.33000115871429e+05	3.33000000452623e+05	3.33000000001766e+05
1024	-	3.33000115871429e+05	3.33000000452628e+05	3.33000000001767e+05

Tabela 1: Tempo de execução (s) (média)

É preciso um N = 2^{14} para que possamos começar a perceber erros de cálculos, por volta de 1024 thread. Depois de N=2^{14}, fica cada vez mais fácil e é preciso um menor número de threads para notar esses erros. Quando N=2^{18} é preciso apenas 16 threads ou menos para notar erros de cálculo.

2.2 Questão 5.3

Modify `omp_trap_1.c` so that
a. it uses the first block of code on page 222, and
b. the time used by the `parallel` block is timed using the OpenMP function `omp_get_wtime()`. The syntax is
`double omp_get_wtime(void)`
It returns the number of seconds that have passed since some time in the past. For details on taking timings, see Section 2.6.4. Also recall that OpenMP has a barrier directive:
`# pragma omp barrier`
Now find a system with at least two cores and time the program with
c. one thread and a large value of n, and
d. two threads and the same value of n. What happens? Download `omp_trap_2.c` from the book's website. How does its performance compare? Explain your answers.

Para `omp_trap_1.c`
Para 1 thread, t = 15.747014 s
Para 2 threads, t = 15.724514 s
Para 64 threads, t = 15.990875 s
para 1024 threas, t = 16.322817 s

Percebemos que quando aumentamos o número de threads, o tempo cresce. Isso pode indicar que o próprio overhead de criação e troca de threads já é responsável pelo aumento desse tempo.

Para `omp_trap_2.c`:
Para 1 thread, t = 15.603254 s
Para 2 threads, t = 7.978304 s
Para 8 threads, t = 2.354562 s
para 64 threas, t = 2.422850 s

O resultado é melhor, pois o cálculo é feito todo de forma paralela e apenas a soma dos resultados parciais acontece em uma região crítica. E vemos a redução do tempo, conforme o esperado.

2.3 Questão 5.5

Suppose that on the amazing Bleeblon computer, variables with type float can store three decimal digits. Also suppose that the Bleeblon's floating point registers can store four decimal digits, and that after any floating point operation, the result is rounded to three decimal digits before being stored. Now suppose a C program declares an array a as follows:

```
float a [] = {4.0, 3.0, 3.0, 1000.0};
```

a. What is the output of the following block of code if it's run on the Bleeblon?

```
int i ;
float sum = 0.0;
for ( i = 0; i < 4; i ++)
    sum += a[i];
printf ("sum = \%4.1f\\n", sum );
```

b. Now consider the following code:

```
int i ;
float sum = 0.0;
# pragma omp parallel for num threads (2) \
  reduction (+: sum )
for ( i = 0; i < 4; i ++)
    sum += a[i];
printf ("sum = \%4.1f\\n", sum );
```

Suppose that the run-time system assigns iterations i = 0, 1 to thread 0 and i = 2, 3 to thread 1. What is the output of this code on the Bleeblon?

a) memory -> registers -> sum register
a[0] = 4.00e0 -> 4.000e0 -> sum = 4.000e0
a[1] = 3.00e0 -> 3.000e0 -> sum = 7.000e0
a[2] = 3.00e0 -> 3.000e0 -> sum = 1.000e1
a[3] = 1.00e3 -> 1.000e3 -> sum = 1.010e3

Print
sum = 1010.0

Todas as operações com sum são feitas no próprio registrador

b) thread 0
a[0] = 4.00e0 -> 4.000e0 -> sum = 4.000e0
a[1] = 3.00e0 -> 3.000e0 -> sum = 7.000e0
sum = 7.00e0

thread 1
a[2] = 3.00e0 -> 3.000e0 -> sum = 3.000e0
a[3] = 1.00e3 -> 1.000e3 -> sum = 1.003e3
sum = 1.00e3

Sum precisa voltar para a memória para a operação de redução

sum_0 = 7.00e0 -> 7.000e0 -> sum_total = 7.000e0
sum_1 = 1.00e3 -> 1.000e3 -> sum_total = 1.007e3
sum = 1.01e3

Print
sum = 1010.0

2.4 Questão 5.6

Write an OpenMP program that determines the default scheduling of parallel for loops. Its input should be the number of iterations, and its output should be which iterations of a parallelized for loop are executed by which thread. For example, if there are two threads and four iterations, the output might be:

```
Thread 0: Iterations 0 -- 1
Thread 1: Iterations 2 -- 3
```

```
/* File:     omp_trap1.c
 * Purpose: Print default scheduling.
 *
 * Input:   a, b, n
 * Output:  estimate of integral from a to b of f(x)
 *          using n trapezoids.
 *
 * Compile: gcc -g -Wall -fopenmp -o omp_trap1 omp_trap1.c
 * Usage:   ./omp_trap1 <number of threads>
 *
 * IPP:  Section 5.2.1 (pp. 216 and ff.)
 */

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

```
18
19 void Usage(char* prog_name);
20
21 int main(int argc, char* argv[]) {
22     int      thread_count;
23     int      iterations;
24
25     if (argc != 3) Usage(argv[0]);
26
27     thread_count = strtol(argv[1], NULL, 10);
28     iterations = strtol(argv[2], NULL, 10);
29
30 #   pragma omp parallel num_threads(thread_count)
31     {
32         int i;
33         int bottom = iterations;
34         int top = 0;
35
36 #       pragma omp for
37         for(i = 0; i < iterations; i++)
38         {
39             if(i < bottom) bottom = i;
40             if(i > top) top = i;
41         }
42
43         int my_rank = omp_get_thread_num();
44 #       pragma omp for ordered
45         for(i = 0; i < thread_count; i++)
46         {
47 #           pragma omp ordered
48             if( i == my_rank)
49             printf("Thread %d: Iterations %d -- %d\n", my_rank,
                   bottom, top);
50
51         }
52
53     }
54     return 0;
55 }  /* main */
56
57 /*
    ------------------------------------------------------------------

58  * Function:     Usage
59  * Purpose:      Print command line for function and terminate
```

```
60  * In arg:       prog_name
61  */
62 void Usage(char* prog_name) {
63
64    fprintf(stderr, "usage: %s <number of threads> <number of
         iterations>\n", prog_name);
65    fprintf(stderr, "   number of trapezoids must be evenly divisible
         by\n");
66    fprintf(stderr, "   number of threads\n");
67    exit(0);
68 }  /* Usage */
```

2.5 Questão 5.8

Consider the loop:

```
1 a[0] = 0;
2 for(i = 1; i < n; i++)
3   a[i] = a[i - 1] + i ;
```

There's clearly a loop-carried dependence, as the value of a[i] can't be computed without the value of a[i - 1]. Can you see a way to eliminate this dependence and parallelize the loop?

Analisando o resultado do loop, vemos que o vetor vai armazenar a soma de uma sequência de números, uma série aritmética que pode ser descrita como:

$$S_n = \frac{n}{2} \cdot (a_1 + a_n) \tag{1}$$

Onde a_1 é o termo inicial, a_n é o termo final e n é o número de termos a serem somados. Logo, podemos traduzir essa formula em um loop for que irá calcular cada elemento do vetor:

```
1 for(i = 0; i < n; i++)
2   a[i] = i*(i+1)/2 ;
```

2.6 Questão 5.9

Modify the trapezoidal rule program that uses a `parallel for` directive (`omp_trap_3.c`) so that the `parallel for` is modified by a `schedule(runtime)` clause. Run the program with various assignments to the environment variable `OMP_SCHEDULE` and determine which iterations are assigned to which thread. This can be done by allocating an array iterations of `n ints` and in the `Trap` function assigning `omp_get_thread_num()` to `iterations[i]` in the ith iteration of the for loop. What is the default assignment of iterations on your system? How are guided schedules determined?

Todos os exemplos foram rodados com 8 threads

Para a=10, b=100, n=100, `OMP_SCHEDULE` = default

Comportamento visualmente semelhante ao dynamic

```
(5, 4, 3, 5, 5, 5, 3, 5, 4, 5, 3, 5, 4, 3, 5, 4, 5, 5, 3, 5, 4, 5, 5, 3,
5, 4, 5, 5, 3, 5, 4, 5, 3, 5, 4, 5, 3, 5, 4, 5, 3, 5, 4, 5, 3, 5, 5, 4,
5, 3, 4, 5, 3, 5, 4, 3, 5, 4, 5, 3, 5, 5, 4, 3, 5, 4, 3, 5, 3, 5, 4, 3,
```

```
5, 3, 5, 4, 3, 5, 4, 3, 5, 4, 5, 3, 4, 5, 4, 5, 3, 4, 5, 0, 7, 5, 3, 5,
4, 7, 5)
```

Para a=10, b=100, n=100, `OMP_SCHEDULE` = static

```
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5,
5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7,
7, 7, 7)
```

Para a=10, b=100, n=100, `OMP_SCHEDULE` = static,1

```
(0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7,
0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7,
0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7,
0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7,
0, 1, 2)
```

Para a=10, b=100, n=100, `OMP_SCHEDULE` = guided

```
(5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5,
5, 5, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 5, 0, 0, 0, 0, 0, 5, 5, 5, 5,
0, 0, 0, 0, 5, 5, 5, 0, 0, 0, 5, 5, 5, 0, 0, 5, 5, 0, 0, 5, 0, 0, 5, 5,
0, 0, 5)
```

No guided schedule, o runtime inicia distribuindo chunksizes grandes para as threads e, a medida que as threads vão acabando o trabalho, o openmp runtime distribui chunksizes cada vez menores para as threads.

Para a=10, b=100, n=100, `OMP_SCHEDULE` = dynamic

```
(4, 5, 2, 5, 2, 4, 5, 4, 2, 4, 5, 5, 2, 4, 5, 2, 4, 2, 5, 2, 4, 2, 5, 4,
2, 4, 2, 5, 2, 4, 2, 5, 2, 4, 2, 5, 4, 2, 4, 2, 5, 2, 4, 5, 2, 4, 2, 5,
2, 4, 2, 5, 2, 4, 2, 5, 2, 4, 2, 5, 2, 4, 2, 5, 4, 2, 5, 2, 4, 2, 5, 2,
4, 2, 5, 2, 4, 5, 2, 4, 2, 5, 2, 4, 2, 5, 2, 4, 2, 5, 7, 2, 4, 5, 7, 7,
2, 4, 7)
```

2.7 Questão 5.10

Recall that all structured blocks modified by an unnamed critical directive form a single critical section. What happens if we have a number of atomic directives in which different variables are being modified? Are they all treated as a single critical section? We can write a small program that tries to determine this. The idea is to have all the threads simultaneously execute something like the following code

```
1 int i ;
2 double my_sum = 0.0;
3 for (i = 0; i < n; i++)
4 # pragma omp atomic
5   my_sum += sin(i);
```

We can do this by modifying the code by a parallel directive:

```
1 # pragma omp parallel num threads ( thread count )
2 {
```

```
3     int i ;
4     double my_sum = 0.0;
5     for(i = 0; i < n; i++)
6     # pragma omp atomic
7       my_sum += sin(i);
8   }
```

Note that since my sum and i are declared in the parallel block, each thread has its own private copy. Now if we time this code for large n when thread count = 1 and we also time it when thread count > 1, then as long as thread count is less than the number of available cores, the run-time for the single-threaded run should be roughly the same as the time for the multithreaded run if the different threads' executions of my sum += sin(i) are treated as different critical sections. On the other hand, if the different executions of my sum += sin(i) are all treated as a single critical section, the multithreaded run should be much slower than the single-threaded run. Write an OpenMP program that implements this test. Does your implementation of OpenMP allow simultaneous execution of updates to different variables when the updates are protected by atomic directives?

```
1
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <math.h>
5   #include <omp.h>
6
7   int main(int argc, char* argv[]) {
8       int     thread_count;
9
10      thread_count = strtol(argv[1], NULL, 10);
11
12      # pragma omp barrier
13      double start = omp_get_wtime();
14
15      # pragma omp parallel num_threads(thread_count)
16      {
17          int i ;
18          double my_sum = 0.0;
19          for ( i = 0; i < 100000 ; i ++)
20          {
21              # pragma omp atomic
22              my_sum += sin(i);
23           }
24      }
25
26      # pragma omp barrier
27      double end = omp_get_wtime();
28
29      printf("Elapsed time = %lf\n", end - start);
```

```
30     return 0;
31 }   /* main */
```

Os resultados encontrados apontam que em média, para 1 thread, o programa leva 0.004676s; para 2 threads, o programa leva 0.004968s; para 4 threads, 0.009019s; e para 8 threads, 0.018598s.

Podemos perceber que a seção atomic é tratada como se fosse a mesma seção para todas as threads. Isso faz com que o tempo de execução com várias threads demore ainda mais tempo, pois cada thread precisa executar o bloco de código isoladamente. Contudo, essa pesquisa indica que essa forma de implementar o openMP não é garantida por padrão e pode ser que mude em outras implementações.

2.8 Questão 5.12

Download the source file `omp_mat_vect_rand_split.c` from the book's web-site. Find a program that does cache profiling (e.g., Valgrind [49]) and compile the program according to the instructions in the cache profiler documentation. (For example, with Valgrind you will want a symbol table and full optimization. (With gcc use, gcc -g -O2 ...). Now run the program according to the instructions in the cache profiler documentation, using input $k \times (k \cdot 10^6)$, $(k \cdot 10^3) \times (k \cdot 10^3)$, and $(k \cdot 10^6) \times k$. Choose k so large that the number of level 2 cache misses is of the order 10^6 for at least one of the input sets of data.

Para 5×5000000

```
Elapsed time = 1.012433e+00 seconds
==9156== D1  misses: 10,004,702 (6,253,722 rd + 3,750,980 wr)
==9156== LLd misses: 9,812,671  (6,061,749 rd + 3,750,922 wr)
==9156== D1  miss rate: 1.3% (1.1% + 1.5%)
==9156== LLd miss rate: 1.2% (1.1% + 1.5%)
```

Para 5000×5000

```
Elapsed time = 9.107981e-01 seconds
==9814== D1  misses: 9,390,505  (6,263,207 rd + 3,127,298 wr)
==9814== LLd misses: 6,255,177  (3,127,971 rd + 3,127,206 wr)
==9814== D1  miss rate: 1.4% (1.3% + 1.5%)
==9814== LLd miss rate: 0.9% (0.6% + 1.5%)
```

Para 5000000×5

```
Elapsed time = 1.123883e+00 seconds
==10010== D1  misses:6,879,904 (3,128,830 rd + 3,751,074 wr)
==10010== LLd misses: 6,878,965 (3,127,987 rd + 3,750,978 wr)
==10010== D1  miss rate: 1.0% (0.6% + 1.8%)
==10010== LLd miss rate: 1.0% (0.6% + 1.8%)
```

a. How many level 1 cache write-misses occur with each of the three inputs? Ver dados

b. How many level 2 cache write-misses occur with each of the three inputs? Ver dados

c. Where do most of the write-misses occur? For which input data does the program have the most write-misses? Can you explain why? O maior número de write-misses está no último input, porém com pouca diferença em relação ao primeiro.

d. How many level 1 cache read-misses occur with each of the three inputs? Ver dados

e. How many level 2 cache read-misses occur with each of the three inputs? Ver dados

f. Where do most of the read-misses occur? For which input data does the program have the most read-misses? Can you explain why? O maior número de read-misses acontece no primeiro input. A variável x[j] é provável de causar read-misses e ela contém bem mais elementos no primeiro input, que nos outros.

g. Run the program with each of the three inputs, but without using the cache profiler. With which input is the program the fastest? With which input is the program the slowest? Can your observations about cache misses help explain the differences? How?

Para 5×5000000: Elapsed time = 2.281404e-02 seconds

Para 5000×5000: Elapsed time = 1.610494e-02 seconds

Para 5000000×5: Elapsed time = 1.966500e-02 seconds

O mais rápido acontece com o input do meio. Isso acontece pois as read-misses geralmente são mais custosas que write-misses, e vemos que no primeiro input, o número de read-misses é maior. Por outro lado, o terceiro input tem um número considerável de write-misses maior que o segundo input e, apesar de terem diferentes read-misses no L1, o L2 possui uma quantidade de read-misses equivalente e é importante considerar que o L2 é bem mais custoso que o L1 miss.

2.9 Questão 5.13

Recall the matrix-vector multiplication example with the 8000×8000 input. Suppose that thread 0 and thread 2 are assigned to different processors. If a cache line contains 64 bytes or 8 double s, is it possible for false sharing between threads 0 and 2 to occur for any part of the vector y ? Why? What about if thread 0 and thread 3 are assigned to different processors; is it possible for false sharing to occur between them for any part of y?

Para que aconteça false-sharing é preciso que elementos de uma mesma linha de cache estejam associados a threads diferentes, nesse caso. Dessa forma, consideramos que os elementos serão divididos como thread 0: 0 - 1999; thread 1: 2000 - 3999; thread 2: 4000 - 5999; e thread 3: 6000 - 7999.

Se para a thread 0, mesmo considerando o elemento 1999 dentro de uma linha de cache, o elemento final dessa linha de cache poderia ser 2006 (isso consideranso 8 doubles na linha de cache, por isso 8 elementos) e portanto não existe possibilidade de um elemento de thread 0 estar em uma mesma linha de cache de um elemento da thread 2 ou mesmo da thread 3.

2.10 Questão 5.14

Recall the matrix-vector multiplication example with an 8×8000000 matrix. Suppose that doubles use 8 bytes of memory and that a cache line is 64 bytes. Also suppose that our system consists of two dual-core processors.

a. What is the minimum number of cache lines that are needed to store the vector y ? Sabendo que y terá 8 elementos, temos a possibilidade de que y[0] seja o primeiro elemento de uma linha de cache, e nesse caso poderá estar contido completamente dentro de uma linha apenas

b. What is the maximum number of cache lines that are needed to store the vector y ? Se y[0] não for o primeiro elemento, temos que os elementos a partir de y[0] serão alocados em uma linha de cache, e quando a linha acabar, os elementos restantes serão alocados na linha de cache seguinte. Portanto, duas linhas é o máximo possível para alocar y.

c. If the boundaries of cache lines always coincide with the boundaries of 8-byte double s, in how many different ways can the components of y be assigned to cache lines? Existem 8 formas diferentes de alocar os componentes de y em linhas de cache. Cada forma depende da posição inicial de y[0], que pode ocupar a primeira posição da linha de cache, a segunda, a terceira ou até a última, no total de 8.

d. If we only consider which pairs of threads share a processor, in how many different ways can four threads be assigned to the processors in our computer? Here, we're assuming that cores on the same processor share cache. É possível escolher threads 0 e 1, 0 e 2, 0 e 3, 1 e 2, 1 e 3, 2 e 3. Contudo, percebemos que ao escolher 0 e 1, automaticamente 2 e 3 vão para o outro processador. E de forma semelhante, as outras escolher também tem esse tipo de relação. Portanto, na realidade existem 3 possibilidades: 0 e 1, 0 e 2, 0 e 3.

e. Is there an assignment of components to cache lines and threads to processors that will result in no false-sharing in our example? In other words, is it possible that the threads assigned to one processor will have their components of y in one cache line, and the threads assigned to the other processor will have their components in a different cache line? Sim, considerando o caso em que os 4 primeiros elementos (y[0]..y[3]) estão associados às threas 0 e 1, e presentes em uma única cache line, e os 4 últimos elementos (y[4]..y[7]) estão associados às threads 2 e 3 e em uma outra cache line.

f. How many assignments of components to cache lines and threads to processors are there? Existem 3 possíveis formas de distribuir as threads e 8 formas de distribuir os elementos de y na cache, logo 24 formas no total.

g. Of these assignments, how many will result in no false sharing? Para que não acontecá false sharing é necessário que thread 0 e 1 estejam no mesmo processador, e portanto 2 e 3 no outro. Porém a divisão de y na cache precisa ser exatamente y[0]..y[3] em uma linha e y[4]..y[7] em outra. Portanto apenas uma configuração não resulta em false sharing.

2.11 Questão 5.15

a. Modify the matrix-vector multiplication program so that it pads the vector y when there's a possibility of false sharing. The padding should be done so that if the threads execute in lock-step, there's no possibility that a single cache line containing an element of y will be shared by two or more threads. Suppose, for example, that a cache line stores eight doubles and we run the program with four threads. If we allocate storage for at least 48 doubles in y , then, on each pass through the for i loop, there's no possibility that two threads will simultaneously access the same cache line.

b. Modify the matrix-vector multiplication program so that each thread uses private storage for its part of y during the for i loop. When a thread is done computing its part of y, it should copy its private storage into the shared variable.

c. How does the performance of these two alternatives compare to the original program. How do they compare to each other?

```
1  /* File:
2   *     omp_mat_vect.c
3   *
4   *
5   * Purpose:
6   *     Computes a parallel matrix-vector product.  Matrix
7   *     is distributed by block rows.  Vectors are distributed by
8   *     blocks.  Unless the DEBUG flag is turned on this version
9   *     uses a random number generator to generate A and x.
10  *
11  * Input:
12  *     None unless compiled with DEBUG flag.
```

```
13  *      With DEBUG flag, A, x
14  *
15  * Output:
16  *      y: the product vector
17  *      Elapsed time for the computation
18  *
19  * Compile:
20  *    gcc -g -Wall -o omp_mat_vect omp_mat_vect.c -lpthread
21  * Usage:
22  *    omp_mat_vect <thread_count> <m> <n>
23  *
24  * Notes:
25  *      1.  Storage for A, x, y is dynamically allocated.
26  *      2.  Number of threads (thread_count) should evenly divide both
27  *          m and n.  The program doesn't check for this.
28  *      3.  We use a 1-dimensional array for A and compute subscripts
29  *          using the formula A[i][j] = A[i*n + j]
30  *      4.  Distribution of A, x, and y is logical:  all three are
31  *          globally shared.
32  *      5.  DEBUG compile flag will prompt for input of A, x, and
33  *          print y
34  *      6.  Uses the OpenMP library function omp_get_wtime() to
35  *          return the time elapsed since some point in the past
36  *
37  * IPP:     Section 5.9 (pp. 253 and ff.)
38  */
39
40 #include <stdio.h>
41 #include <stdlib.h>
42 #include <omp.h>
43
44 /* Serial functions */
45 void Get_args(int argc, char* argv[], int* thread_count_p,
46       int* m_p, int* n_p);
47 void Usage(char* prog_name);
48 void Gen_matrix(double A[], int m, int n);
49 void Read_matrix(char* prompt, double A[], int m, int n);
50 void Gen_vector(double x[], int n);
51 void Read_vector(char* prompt, double x[], int n);
52 void Print_matrix(char* title, double A[], int m, int n);
53 void Print_vector(char* title, double y[], double m);
54
55 /* Parallel function */
56 void Omp_mat_vect(double A[], double x[], double y[],
57       int m, int n, int thread_count);
```

```
58
59 /*--------------------------------------------------------------------
       */
60 int main(int argc, char* argv[]) {
61    int      thread_count;
62    int      m, n;
63    double* A;
64    double* x;
65    double* y;
66
67    Get_args(argc, argv, &thread_count, &m, &n);
68
69    A = malloc(m*n*sizeof(double));
70    x = malloc(n*sizeof(double));
71    y = malloc((m*8)*sizeof(double));
72
73  # ifdef DEBUG
74       Read_matrix("Enter the matrix", A, m, n);
75       Print_matrix("We read", A, m, n);
76       Read_vector("Enter the vector", x, n);
77       Print_vector("We read", x, n);
78  # else
79       Gen_matrix(A, m, n);
80 /*    Print_matrix("We generated", A, m, n); */
81       Gen_vector(x, n);
82 /*    Print_vector("We generated", x, n); */
83  # endif
84
85    Omp_mat_vect(A, x, y, m, n, thread_count);
86
87 #  ifdef DEBUG
88       Print_vector("The product is", y, m);
89 #  else
90 /*    Print_vector("The product is", y, m); */
91 #  endif
92
93    free(A);
94    free(x);
95    free(y);
96
97    return 0;
98 }  /* main */
99
100
101 /*--------------------------------------------------------------------
```

```
102  * Function:  Get_args
103  * Purpose:   Get command line args
104  * In args:   argc, argv
105  * Out args:  thread_count_p, m_p, n_p
106  */
107 void Get_args(int argc, char* argv[], int* thread_count_p,
108       int* m_p, int* n_p)  {
109
110    if (argc != 4) Usage(argv[0]);
111    *thread_count_p = strtol(argv[1], NULL, 10);
112    *m_p = strtol(argv[2], NULL, 10);
113    *n_p = strtol(argv[3], NULL, 10);
114    if (*thread_count_p <= 0 || *m_p <= 0 || *n_p <= 0) Usage(argv[0])
       ;
115
116 }  /* Get_args */
117
118 /*--------------------------------------------------------------------
119  * Function:  Usage
120  * Purpose:   print a message showing what the command line should
121  *            be, and terminate
122  * In arg :   prog_name
123  */
124 void Usage (char* prog_name) {
125    fprintf(stderr, "usage: %s <thread_count> <m> <n>\n", prog_name);
126    exit(0);
127 }  /* Usage */
128
129 /*--------------------------------------------------------------------
130  * Function:    Read_matrix
131  * Purpose:     Read in the matrix
132  * In args:     prompt, m, n
133  * Out arg:     A
134  */
135 void Read_matrix(char* prompt, double A[], int m, int n) {
136    int             i, j;
137
138    printf("%s\n", prompt);
139    for (i = 0; i < m; i++)
140       for (j = 0; j < n; j++)
141          scanf("%lf", &A[i*n+j]);
142 }  /* Read_matrix */
143
144 /*--------------------------------------------------------------------
145  * Function: Gen_matrix
```

```
146  * Purpose:  Use the random number generator random to generate
147  *     the entries in A
148  * In args:  m, n
149  * Out arg:  A
150  */
151 void Gen_matrix(double A[], int m, int n) {
152    int i, j;
153    for (i = 0; i < m; i++)
154       for (j = 0; j < n; j++)
155          A[i*n+j] = random()/((double) RAND_MAX);
156 }  /* Gen_matrix */
157
158 /*--------------------------------------------------------------------
159  * Function: Gen_vector
160  * Purpose:  Use the random number generator random to generate
161  *     the entries in x
162  * In arg:   n
163  * Out arg:  A
164  */
165 void Gen_vector(double x[], int n) {
166    int i;
167    for (i = 0; i < n; i++)
168       x[i] = random()/((double) RAND_MAX);
169 }  /* Gen_vector */
170
171 /*--------------------------------------------------------------------
172  * Function:        Read_vector
173  * Purpose:         Read in the vector x
174  * In arg:          prompt, n
175  * Out arg:         x
176  */
177 void Read_vector(char* prompt, double x[], int n) {
178    int    i;
179
180    printf("%s\n", prompt);
181    for (i = 0; i < n; i++)
182       scanf("%lf", &x[i]);
183 }  /* Read_vector */
184
185
186 /*--------------------------------------------------------------------
187  * Function:  Omp_mat_vect
188  * Purpose:   Multiply an mxn matrix by an nx1 column vector
189  * In args:   A, x, m, n, thread_count
190  * Out arg:   y
```

```
191  */
192 void Omp_mat_vect(double A[], double x[], double y[],
193       int m, int n, int thread_count) {
194    int i, j;
195    double start, finish, elapsed;
196
197    start = omp_get_wtime();
198 #  pragma omp parallel for num_threads(thread_count)  \
199       default(none) private(i, j)  shared(A, x, y, m, n)
200    for (i = 0; i < m; i++) {
201       y[i*8] = 0.0;
202       for (j = 0; j < n; j++)
203          y[i*8] += A[i*n+j]*x[j];
204    }
205    finish = omp_get_wtime();
206    elapsed = finish - start;
207    printf("Elapsed time = %e seconds\n", elapsed);
208
209 }  /* Omp_mat_vect */
210
211
212 /*--------------------------------------------------------------------
213  * Function:     Print_matrix
214  * Purpose:      Print the matrix
215  * In args:      title, A, m, n
216  */
217 void Print_matrix( char* title, double A[], int m, int n) {
218    int   i, j;
219
220    printf("%s\n", title);
221    for (i = 0; i < m; i++) {
222       for (j = 0; j < n; j++)
223          printf("%4.1f ", A[i*n + j]);
224       printf("\n");
225    }
226 }  /* Print_matrix */
227
228
229 /*--------------------------------------------------------------------
230  * Function:     Print_vector
231  * Purpose:      Print a vector
232  * In args:      title, y, m
233  */
234 void Print_vector(char* title, double y[], double m) {
235    int   i;
```

```
236
237     printf("%s\n", title);
238     for (i = 0; i < m; i++)
239        printf("%4.1f ", y[i]);
240     printf("\n");
241  }  /* Print_vector */
```

```
1  /* File:
2   *     omp_mat_vect.c
3   *
4   *
5   * Purpose:
6   *     Computes a parallel matrix-vector product.  Matrix
7   *     is distributed by block rows.  Vectors are distributed by
8   *     blocks.  Unless the DEBUG flag is turned on this version
9   *     uses a random number generator to generate A and x.
10  *
11  * Input:
12  *     None unless compiled with DEBUG flag.
13  *     With DEBUG flag, A, x
14  *
15  * Output:
16  *     y: the product vector
17  *     Elapsed time for the computation
18  *
19  * Compile:
20  *    gcc -g -Wall -o omp_mat_vect omp_mat_vect.c -lpthread
21  * Usage:
22  *    omp_mat_vect <thread_count> <m> <n>
23  *
24  * Notes:
25  *     1.  Storage for A, x, y is dynamically allocated.
26  *     2.  Number of threads (thread_count) should evenly divide both
27  *         m and n.  The program doesn't check for this.
28  *     3.  We use a 1-dimensional array for A and compute subscripts
29  *         using the formula A[i][j] = A[i*n + j]
30  *     4.  Distribution of A, x, and y is logical:  all three are
31  *         globally shared.
32  *     5.  DEBUG compile flag will prompt for input of A, x, and
33  *         print y
34  *     6.  Uses the OpenMP library function omp_get_wtime() to
35  *         return the time elapsed since some point in the past
36  *
37  * IPP:    Section 5.9 (pp. 253 and ff.)
38  */
```

```
39
40 #include <stdio.h>
41 #include <stdlib.h>
42 #include <omp.h>
43
44 /* Serial functions */
45 void Get_args(int argc, char* argv[], int* thread_count_p,
46       int* m_p, int* n_p);
47 void Usage(char* prog_name);
48 void Gen_matrix(double A[], int m, int n);
49 void Read_matrix(char* prompt, double A[], int m, int n);
50 void Gen_vector(double x[], int n);
51 void Read_vector(char* prompt, double x[], int n);
52 void Print_matrix(char* title, double A[], int m, int n);
53 void Print_vector(char* title, double y[], double m);
54
55 /* Parallel function */
56 void Omp_mat_vect(double A[], double x[], double y[],
57       int m, int n, int thread_count);
58
59 /*-------------------------------------------------------------------
   */
60 int main(int argc, char* argv[]) {
61    int      thread_count;
62    int      m, n;
63    double* A;
64    double* x;
65    double* y;
66
67    Get_args(argc, argv, &thread_count, &m, &n);
68
69    A = malloc(m*n*sizeof(double));
70    x = malloc(n*sizeof(double));
71    y = malloc(m*sizeof(double));
72
73  # ifdef DEBUG
74       Read_matrix("Enter the matrix", A, m, n);
75       Print_matrix("We read", A, m, n);
76       Read_vector("Enter the vector", x, n);
77       Print_vector("We read", x, n);
78  # else
79       Gen_matrix(A, m, n);
80 /*    Print_matrix("We generated", A, m, n); */
81       Gen_vector(x, n);
82 /*    Print_vector("We generated", x, n); */
```

```
83  # endif
84
85     Omp_mat_vect(A, x, y, m, n, thread_count);
86
87  #  ifdef DEBUG
88        Print_vector("The product is", y, m);
89  #  else
90  /*    Print_vector("The product is", y, m); */
91  #  endif
92
93     free(A);
94     free(x);
95     free(y);
96
97     return 0;
98  }  /* main */
99
100
101 /*--------------------------------------------------------------------
102  * Function:  Get_args
103  * Purpose:   Get command line args
104  * In args:   argc, argv
105  * Out args:  thread_count_p, m_p, n_p
106  */
107 void Get_args(int argc, char* argv[], int* thread_count_p,
108       int* m_p, int* n_p)  {
109
110    if (argc != 4) Usage(argv[0]);
111    *thread_count_p = strtol(argv[1], NULL, 10);
112    *m_p = strtol(argv[2], NULL, 10);
113    *n_p = strtol(argv[3], NULL, 10);
114    if (*thread_count_p <= 0 || *m_p <= 0 || *n_p <= 0) Usage(argv[0])
       ;
115
116 }  /* Get_args */
117
118 /*--------------------------------------------------------------------
119  * Function:  Usage
120  * Purpose:   print a message showing what the command line should
121  *            be, and terminate
122  * In arg :   prog_name
123  */
124 void Usage (char* prog_name) {
125    fprintf(stderr, "usage: %s <thread_count> <m> <n>\n", prog_name);
126    exit(0);
```

```
127 }  /* Usage */
128
129 /*---------------------------------------------------------------------
130  * Function:     Read_matrix
131  * Purpose:      Read in the matrix
132  * In args:      prompt, m, n
133  * Out arg:      A
134  */
135 void Read_matrix(char* prompt, double A[], int m, int n) {
136    int             i, j;
137
138    printf("%s\n", prompt);
139    for (i = 0; i < m; i++)
140       for (j = 0; j < n; j++)
141          scanf("%lf", &A[i*n+j]);
142 }  /* Read_matrix */
143
144 /*---------------------------------------------------------------------
145  * Function: Gen_matrix
146  * Purpose:  Use the random number generator random to generate
147  *     the entries in A
148  * In args:  m, n
149  * Out arg:  A
150  */
151 void Gen_matrix(double A[], int m, int n) {
152    int i, j;
153    for (i = 0; i < m; i++)
154       for (j = 0; j < n; j++)
155          A[i*n+j] = random()/((double) RAND_MAX);
156 }  /* Gen_matrix */
157
158 /*---------------------------------------------------------------------
159  * Function: Gen_vector
160  * Purpose:  Use the random number generator random to generate
161  *     the entries in x
162  * In arg:   n
163  * Out arg:  A
164  */
165 void Gen_vector(double x[], int n) {
166    int i;
167    for (i = 0; i < n; i++)
168       x[i] = random()/((double) RAND_MAX);
169 }  /* Gen_vector */
170
171 /*---------------------------------------------------------------------
```

```
172  * Function:         Read_vector
173  * Purpose:          Read in the vector x
174  * In arg:           prompt, n
175  * Out arg:          x
176  */
177 void Read_vector(char* prompt, double x[], int n) {
178    int    i;
179
180    printf("%s\n", prompt);
181    for (i = 0; i < n; i++)
182       scanf("%lf", &x[i]);
183 }  /* Read_vector */
184
185
186 /*--------------------------------------------------------------------
187  * Function:  Omp_mat_vect
188  * Purpose:   Multiply an mxn matrix by an nx1 column vector
189  * In args:   A, x, m, n, thread_count
190  * Out arg:   y
191  */
192 void Omp_mat_vect(double A[], double x[], double y[],
193       int m, int n, int thread_count) {
194    int i, j;
195    double start, finish, elapsed;
196
197    start = omp_get_wtime();
198 #  pragma omp parallel num_threads(thread_count)  \
199    default(none) private(i, j)  shared(A, x, y, m, n, thread_count)
200    {
201        double* local_y = malloc((m/thread_count)*sizeof(double));
202        int index = 0;
203        #  pragma omp for
204        for (i = 0; i < m; i++) {
205           local_y[index] = 0.0;
206           for (j = 0; j < n; j++)
207              local_y[index] += A[i*n+j]*x[j];
208           index++;
209        }
210
211        index = 0;
212        # pragma omp for
213        for(i = 0; i < m; i++)
214             y[i] = local_y[index++];
215
216         free(local_y);
```

```
217     }
218     finish = omp_get_wtime();
219     elapsed = finish - start;
220     printf("Elapsed time = %e seconds\n", elapsed);
221
222 }   /* Omp_mat_vect */
223
224
225 /*--------------------------------------------------------------------
226  * Function:     Print_matrix
227  * Purpose:      Print the matrix
228  * In args:      title, A, m, n
229  */
230 void Print_matrix( char* title, double A[], int m, int n) {
231     int   i, j;
232
233     printf("%s\n", title);
234     for (i = 0; i < m; i++) {
235         for (j = 0; j < n; j++)
236             printf("%4.1f ", A[i*n + j]);
237         printf("\n");
238     }
239 }   /* Print_matrix */
240
241
242 /*--------------------------------------------------------------------
243  * Function:     Print_vector
244  * Purpose:      Print a vector
245  * In args:      title, y, m
246  */
247 void Print_vector(char* title, double y[], double m) {
248     int   i;
249
250     printf("%s\n", title);
251     for (i = 0; i < m; i++)
252         printf("%4.1f ", y[i]);
253     printf("\n");
254 }   /* Print_vector */
```

c) O programa com variáveis privadas rodou em 9.583470e-02 seconds para 8 threads e matriz 9000x9000.
O programa original, para as mesmas configurações teve média de 1.078727e-01 seconds.
O programa com padding teve média de 1.063539e-01 seconds.