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.

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

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

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?

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

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

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?

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?