

Lista do capítulo 5 de programação concorrente e distribuída:

Bruno Victor Paiva Da Silva

Questões respondidas

5.12 EXERCISES:

[Questão 5.2](#) peso 1

[Questão 5.3](#) peso 1

[Questão 5.5](#) peso 1

[Questão 5.6](#) peso 1

[Questão 5.8](#) peso 1

[Questão 5.9](#) peso 1

[Questão 5.13](#) peso 1

[Questão 5.16](#) peso 1

5.2. Faça o download do omp_trap_1.c no site do livro e exclua a directiva critical. Agora compile e execute o programa com mais e mais threads e valores de n cada vez maiores. Quantos threads e quantos trapézios são necessários antes que o resultado seja incorreto?

Foi feita uma execução com valores de $a = 1000$ e $b = 2500$, inicialmente para definir o número de threads a serem usadas começamos com 1 e em seguida duplicamos o valor de threads, pois o valor de $n = 4000$.

```
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ gcc -g -Wall -fopenmp -o omp_trap1 omp_trap1.c
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ ./omp_trap1 1
Enter a, b, and n
1000
2500
4000
With n = 4000 trapezoids, our estimate
of the integral from 1000.000000 to 2500.000000 = 4.8750003515625e+09
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ gcc -g -Wall -fopenmp -o omp_trap1 omp_trap1.c
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ ./omp_trap1 2
Enter a, b, and n
1000
2500
4000
With n = 4000 trapezoids, our estimate
of the integral from 1000.000000 to 2500.000000 = 4.8750003515625e+09
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ gcc -g -Wall -fopenmp -o omp_trap1 omp_trap1.c
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ ./omp_trap1 4
Enter a, b, and n
1000
2500
4000
With n = 4000 trapezoids, our estimate
of the integral from 1000.000000 to 2500.000000 = 3.95507815136719e+09
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ gcc -g -Wall -fopenmp -o omp_trap1 omp_trap1.c
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ ./omp_trap1 8
Enter a, b, and n
1000
2500
4000
With n = 4000 trapezoids, our estimate
of the integral from 1000.000000 to 2500.000000 = 3.56762697509766e+09
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$
```

Percebemos que com a remoção da diretiva **critical** já tivemos uma mudança de resultado com apenas 4 threads com a execução com 5000 e 6000 interações tivemos uma maior queda no resultado porém ocorreu quando foi executado com 8 threads

```
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ ./omp_trap1 1
Enter a, b, and n
1000
2500
5000
With n = 5000 trapezoids, our estimate
of the integral from 1000.000000 to 2500.000000 = 4.8750002250000e+09
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ ./omp_trap1 2
Enter a, b, and n
1000
2500
5000
With n = 5000 trapezoids, our estimate
of the integral from 1000.000000 to 2500.000000 = 4.8750002250000e+09
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ ./omp_trap1 4
Enter a, b, and n
1000
2500
5000
With n = 5000 trapezoids, our estimate
of the integral from 1000.000000 to 2500.000000 = 4.8750002250000e+09
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ ./omp_trap1 8
Enter a, b, and n
1000
2500
5000
With n = 5000 trapezoids, our estimate
of the integral from 1000.000000 to 2500.000000 = 2.5209961050000e+09
```

```

• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ ./omp_trap1 1
Enter a, b, and n
1000
2500
6000
With n = 6000 trapezoids, our estimate
of the integral from 1000.000000 to 2500.000000 = 4.8750001562500e+09
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ ./omp_trap1 2
Enter a, b, and n
1000
2500
6000
With n = 6000 trapezoids, our estimate
of the integral from 1000.000000 to 2500.000000 = 4.8750001562500e+09
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ ./omp_trap1 4
Enter a, b, and n
1000
2500
6000
With n = 6000 trapezoids, our estimate
of the integral from 1000.000000 to 2500.000000 = 4.8750001562500e+09
• brunopaiva@Linux:~/ExercicieOpenMP/5.2$ ./omp_trap1 8
Enter a, b, and n
1000
2500
6000
With n = 6000 trapezoids, our estimate
of the integral from 1000.000000 to 2500.000000 = 2.07641602148438e+09

```

5.3. Modifique omp_trap_1.c para que

a. ele use o primeiro bloco de código na página 222 e

b. o tempo usado pelo bloco paralelo é cronometrado usando a função OpenMP `omp_get_wtime()`. A sintaxe é `double omp_get_wtime(void)`. Ele retorna o número de segundos que se passaram desde algum momento no passado. Para obter detalhes sobre os tempos, consulte a Seção 2.6.4. Lembre-se também que o OpenMP possui uma diretiva de barreira:

```
# pragma omp barrier
```

Agora encontre um sistema com pelo menos dois núcleos e cronometre o programa com

c. uma thread e um grande valor de n, e

d. dois segmentos e o mesmo valor de n.

O que acontece? Faça o download do `omp_trap_2.c` no site do livro. Como se compara o desempenho? Explique suas respostas.

como podemos perceber abaixo o código sequencial sendo executado apenas com 1 threads se mostrou mais eficiente do que o que foi executado com 2 threads.

```

• brunopaiva@Linux:~/ExercicieOpenMP/5.3$ gcc -g -Wall -fopenmp -o omp_trap1 omp_trap1.c
• brunopaiva@Linux:~/ExercicieOpenMP/5.3$ ./omp_trap1 1
Enter a, b, and n
1
5000
10000000
With n = 10000000 trapezoids, our estimate
of the integral from 1.000000 to 5000.000000 = 4.16666666663317e+10
Time: 0.064205
• brunopaiva@Linux:~/ExercicieOpenMP/5.3$ ./omp_trap1 2
Enter a, b, and n
1
5000
10000000
With n = 10000000 trapezoids, our estimate
of the integral from 1.000000 to 5000.000000 = 4.16666666663371e+10
Time: 0.074712
• brunopaiva@Linux:~/ExercicieOpenMP/5.3$

```

```

• brunopaiva@Linux:~/ExercicieOpenMP/5.3B$ gcc -g -Wall -fopenmp -o omp_trap2 omp_trap2.c -lm
• brunopaiva@Linux:~/ExercicieOpenMP/5.3B$ ./omp_trap2 1
Enter a, b, and n
1
5000
10000000
With n = 10000000 trapezoids, our estimate
of the integral from 1.000000 to 5000.000000 = 4.16666666663317e+10
Elapsed time: 8.377126e-02 seconds
• brunopaiva@Linux:~/ExercicieOpenMP/5.3B$ ./omp_trap2 2
Enter a, b, and n
1
5000
10000000
With n = 10000000 trapezoids, our estimate
of the integral from 1.000000 to 5000.000000 = 4.16666666663371e+10
Elapsed time: 5.490769e-02 seconds
• brunopaiva@Linux:~/ExercicieOpenMP/5.3B$ █

```

Isso é decorrente do local de utilização do `omp critical`, onde a seção crítica do primeiro código impede as thread executar paralelamente o cálculo do LocalTrap, assim, uma thread tem que esperar a outra terminar de executar o cálculo. Enquanto, no segundo a região crítica no segundo código é apenas na soma do valor global, soma de todos os LocalTrap.

5.5. Suponha que no incrível computador Bleeblon, as variáveis do tipo float possam armazenar três dígitos decimais. Suponha também que os registradores de ponto flutuante do Bleeblon possam armazenar quatro dígitos decimais e que, após qualquer operação de ponto flutuante, o resultado seja arredondado para três dígitos decimais antes de ser armazenado. Agora suponha que um programa C declare um array `a` da seguinte forma:

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

a. Qual é a saída do seguinte bloco de código se for executado no Bleeblon?

```

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

```

Time	Operation	Operad 1	Operad 2	Result
1	Fetch operands	0.0 x 10 ⁰	4.0 x 10 ⁰	
2	Compare expon	0.0 x 10 ⁰	4.0 x 10 ⁰	
3	Shift one operan	0.0 x 10 ⁰	4.0 x 10 ⁰	
4	Add	0.0 x 10 ⁰	4.0 x 10 ⁰	4.0 x 10 ⁰
5	Normalize result	0.0 x 10 ⁰	4.0 x 10 ⁰	4.0 x 10 ⁰

6	Round result	0.0×10^0	4.0×10^0	0.40×10^1
7	Store result	0.0×10^0	4.0×10^0	0.40×10^1

Time	Operation	Operad 1	Operad 2	Result
1	Fetch operands	0.40×10^1	3.0×10^0	
2	Compare expon	0.40×10^1	3.0×10^0	
3	Shift one operan	0.40×10^1	0.30×10^1	
4	Add	0.40×10^1	0.30×10^1	0.70×10^1
5	Normalize result	0.40×10^1	0.30×10^1	0.70×10^1
6	Round result	0.40×10^1	0.30×10^1	0.70×10^1
7	Store result	0.40×10^1	0.30×10^1	0.70×10^1

Time	Operation	Operad 1	Operad 2	Result
1	Fetch operands	0.70×10^1	3.0×10^0	
2	Compare expon	0.70×10^1	3.0×10^0	
3	Shift one operan	0.70×10^1	$0,30 \times 10^1$	
4	Add	0.70×10^1	$0,30 \times 10^1$	1.00×10^1
5	Normalize result	0.70×10^1	$0,30 \times 10^1$	1.00×10^1
6	Round result	0.70×10^1	$0,30 \times 10^1$	$0,10 \times 10^2$
7	Store result	0.70×10^1	$0,30 \times 10^1$	$0,10 \times 10^2$

Time	Operation	Operad 1	Operad 2	Result
1	Fetch operands	$0,10 \times 10^2$	1.00×10^3	
2	Compare expon	$0,10 \times 10^2$	1.00×10^3	
3	Shift one operan	$0,01 \times 10^3$	1.00×10^3	
4	Add	$0,01 \times 10^3$	1.00×10^3	1.01×10^3
5	Normalize result	$0,01 \times 10^3$	1.00×10^3	1.01×10^3

6	Round result	$0,01 \times 10^3$	1.00×10^3	1.01×10^3
7	Store result	$0,01 \times 10^3$	1.00×10^3	1.01×10^3

b. Agora considere o seguinte código:

```
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);
```

Suponha que o sistema de tempo de execução atribua iterações $i = 0, 1$ para thread 0 e $i = 2, 3$ ao thread 1. Qual é a saída desse código no Bleeblon?

Thread 0:

Time	Operation	Operad 1	Operad 2	Result
1	Fetch operands	$0,400 \times 10^1$	$3,000 \times 10^0$	
2	Compare expon	$0,400 \times 10^1$	$3,000 \times 10^0$	
3	Shift one operan	$0,400 \times 10^1$	$0,300 \times 10^1$	
4	Add	$0,400 \times 10^1$	$0,300 \times 10^1$	$0,700 \times 10^1$
5	Normalize result	$0,400 \times 10^1$	$0,300 \times 10^1$	$0,70 \times 10^1$
6	Round result	$0,400 \times 10^1$	$0,300 \times 10^1$	$0,70 \times 10^1$
7	Store result	$0,400 \times 10^1$	$0,300 \times 10^1$	$0,70 \times 10^1$

Thread 1:

Time	Operation	Operad 1	Operad 2	Result
1	Fetch operands	$3,000 \times 10^0$	$1,000 \times 10^3$	
2	Compare expon	$3,000 \times 10^0$	$1,000 \times 10^3$	
3	Shift one operan	$0,003 \times 10^3$	$1,000 \times 10^3$	
4	Add	$0,003 \times 10^3$	$1,000 \times 10^3$	$1,003 \times 10^3$
5	Normalize result	$0,003 \times 10^3$	$1,000 \times 10^3$	$1,00 \times 10^3$
6	Round result	$0,003 \times 10^3$	$1,000 \times 10^3$	$1,00 \times 10^3$
7	Store result	$0,003 \times 10^3$	$1,000 \times 10^3$	$1,00 \times 10^3$

Thread 0:

Time	Operation	Operad 1	Operad 2	Result
1	Fetch operands	$0,700 \times 10^1$	$1,000 \times 10^3$	
2	Compare expon	$0,700 \times 10^1$	$1,000 \times 10^3$	
3	Shift one operan	$0,007 \times 10^3$	$1,000 \times 10^3$	
4	Add	$0,007 \times 10^3$	$1,000 \times 10^3$	$1,007 \times 10^3$
5	Normalize result	$0,007 \times 10^3$	$1,000 \times 10^3$	$1,00 \times 10^3$

6	Round result	$0,007 \times 10^3$	$1,000 \times 10^3$	$1,00 \times 10^3$
7	Store result	$0,007 \times 10^3$	$1,000 \times 10^3$	$1,00 \times 10^3$

5.6. Escreva um programa OpenMP que determine o agendamento padrão de loops for paralelos. Sua entrada deve ser o número de iterações e sua saída deve ser quais iterações de um loop for paralelizado são executadas por qual thread. Por exemplo, se houver dois threads e quatro iterações, a saída pode ser:

Thread 0: Iterations 0 → 1

Thread 1: Iterations 2 → 3

```

• brunopaiva@Linux:~/ExercicieOpenMP/5.6$ gcc -g -o Agendamento -fopenmp Agendamento.c
• brunopaiva@Linux:~/ExercicieOpenMP/5.6$ ./Agendamento
Enter the number of threads: 4
Enter the number of iterations: 12

Thread 0: Iteration 0
Thread 0: Iteration 1
Thread 0: Iteration 2
Thread 3: Iteration 9
Thread 3: Iteration 10
Thread 3: Iteration 11
Thread 2: Iteration 6
Thread 2: Iteration 7
Thread 2: Iteration 8
Thread 1: Iteration 3
Thread 1: Iteration 4
Thread 1: Iteration 5
• brunopaiva@Linux:~/ExercicieOpenMP/5.6$

```

5.8. Considere o loop

$a[0] = 0;$

for ($i = 1; i < n; i++$)

$a[i] = a[i - 1] + i;$

Há claramente uma dependência de loop, pois o valor de $a[i]$ não pode ser calculado sem o valor de $a[i - 1]$. Você consegue ver uma maneira de eliminar essa dependência e paralelizar o loop?

Há um padrão nos valores de $a[i]$ (0, 1, 3, 6, 10, 15...), com base nisso é possível achar uma fórmula explícita com base nos valores. Utilizando o valor de i como base para o cálculo, chegamos em $((i * (i + 1)) / 2)$, os resultados batem com o anterior.

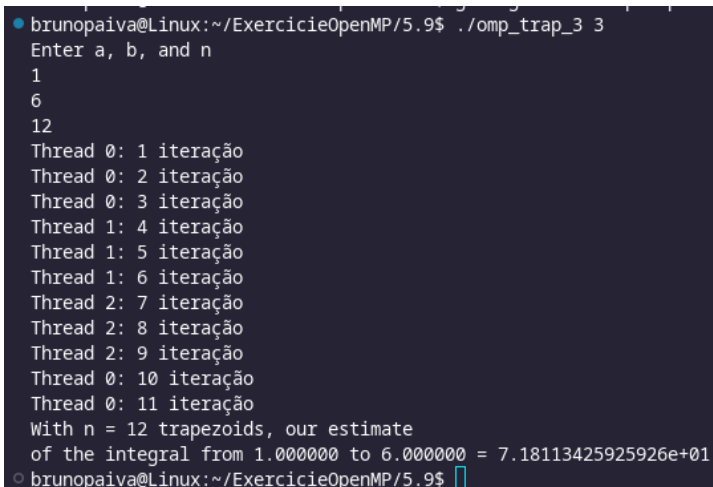
Agora para paralelizar um for utilizamos a diretiva for, como boas práticas e definimos quais

variáveis serão privadas e compartilhadas.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char const *argv[]) {
    int i;
    int *a;
    int thread_count = strtol(argv[1], NULL, 10);
    int n = strtol(argv[2], NULL, 10);
    a = (int *)malloc(n * sizeof(int));
    # pragma omp parallel for num_threads(thread_count) \
    default(none) private(i) shared(a, n)
    for (i = 0; i < n; i++)
        a[i] = (i * (i + 1)) / 2;
    for (i = 0; i < n; i++)
        printf("%d\n", a[i]);
    return 0;
} /* main */
```

5.9. Modifique o programa de regras trapezoidais que usa uma diretiva parallel for (omp_trap_3.c) para que o parallel for seja modificado por uma cláusula schedule(runtime). Execute o programa com várias atribuições à variável de ambiente OMP_SCHEDULE e determine quais iterações são atribuídas a qual encadeamento. Isso pode ser feito alocando iterações de um array de n ints e na função Trap atribuindo omp_get_thread_num() a iterations[i] na i-ésima iteração do loop for. Qual é a atribuição padrão de iterações em seu sistema? Como são determinados os schedule guided?

schedule(static, 3)



```
brunopaiva@Linux:~/ExercicieOpenMP/5.9$ ./omp_trap_3 3
Enter a, b, and n
1
6
12
Thread 0: 1 iteração
Thread 0: 2 iteração
Thread 0: 3 iteração
Thread 1: 4 iteração
Thread 1: 5 iteração
Thread 1: 6 iteração
Thread 2: 7 iteração
Thread 2: 8 iteração
Thread 2: 9 iteração
Thread 0: 10 iteração
Thread 0: 11 iteração
With n = 12 trapezoids, our estimate
of the integral from 1.000000 to 6.000000 = 7.18113425926e+01
brunopaiva@Linux:~/ExercicieOpenMP/5.9$
```

schedule(dynamic, 3)


```

of the integral from 1.000000 to 6.000000 = 7.18113425925926e+01
• brunopaiva@Linux:~/ExercicieOpenMP/5.9$ gcc -g -Wall -fopenmp -o omp_trap_3 omp_trap_3.c
• brunopaiva@Linux:~/ExercicieOpenMP/5.9$ ./omp_trap_3 3
Enter a, b, and n
1
6
12
Thread 0: 1 iteração
Thread 0: 2 iteração
Thread 0: 3 iteração
Thread 1: 4 iteração
Thread 1: 5 iteração
Thread 1: 6 iteração
Thread 2: 7 iteração
Thread 2: 8 iteração
Thread 2: 9 iteração
Thread 0: 10 iteração
Thread 0: 11 iteração
With n = 12 trapezoids, our estimate
of the integral from 1.000000 to 6.000000 = 7.18113425925926e+01

```

schedule(guided, 3)

```

• brunopaiva@Linux:~/ExercicieOpenMP/5.9$ gcc -g -Wall -fopenmp -o omp_trap_3 omp_trap_3.c
• brunopaiva@Linux:~/ExercicieOpenMP/5.9$ ./omp_trap_3 3
Enter a, b, and n
1
6
12
Thread 1: 1 iteração
Thread 1: 2 iteração
Thread 1: 3 iteração
Thread 1: 4 iteração
Thread 0: 5 iteração
Thread 0: 6 iteração
Thread 0: 7 iteração
Thread 2: 8 iteração
Thread 2: 9 iteração
Thread 2: 10 iteração
Thread 0: 11 iteração
With n = 12 trapezoids, our estimate
of the integral from 1.000000 to 6.000000 = 7.18113425925926e+01

```

schedule(auto)

```

• brunopaiva@Linux:~/ExercicieOpenMP/5.9$ gcc -g -Wall -fopenmp -o omp_trap_3 omp_trap_3.c
• brunopaiva@Linux:~/ExercicieOpenMP/5.9$ ./omp_trap_3 3
Enter a, b, and n
1
6
12
Thread 0: 1 iteração
Thread 0: 2 iteração
Thread 0: 3 iteração
Thread 0: 4 iteração
Thread 1: 5 iteração
Thread 1: 6 iteração
Thread 1: 7 iteração
Thread 1: 8 iteração
Thread 2: 9 iteração
Thread 2: 10 iteração
Thread 2: 11 iteração
With n = 12 trapezoids, our estimate
of the integral from 1.000000 to 6.000000 = 7.18113425925926e+01

```

schedule(runtime)

```

• brunopaiva@linux:~/ExercicieOpenMP/5.9$ gcc -g -Wall -fopenmp -o omp_trap_3 omp_trap_3.c
• brunopaiva@linux:~/ExercicieOpenMP/5.9$ ./omp_trap_3 3
Enter a, b, and n
1
6
12
Thread 1: 1 iteração
Thread 2: 2 iteração
Thread 0: 3 iteração
Thread 2: 4 iteração
Thread 1: 5 iteração
Thread 2: 6 iteração
Thread 1: 7 iteração
Thread 0: 8 iteração
Thread 2: 9 iteração
Thread 1: 10 iteração
Thread 0: 11 iteração
With n = 12 trapezoids, our estimate
of the integral from 1.000000 to 6.000000 = 7.18113425925926e+01

```

5.13. Lembre-se do exemplo de multiplicação de matriz-vetor com uma entrada de 8000×8000 . Suponha que a thread 0 e a thread 2 sejam atribuídas a processadores diferentes. Se uma linha de cache contém 64 bytes ou 8 double s, é possível ocorrer falso compartilhamento entre as threads 0 e 2 para qualquer parte do vetor y ? Por quê? E se a thread 0 e a thread 3 forem atribuídas a processadores diferentes, é possível ocorrer falso compartilhamento entre elas para qualquer parte de y ?

R:

Thread 0: y[0], y[1], ..., y[1999]

Thread 1: y[2000], y[2001], ..., y[3999]

Thread 2: y[4000], y[4001], ..., y[5999]

Thread 3: y[6000], y[6001], ..., y[7999]

Podemos definir que se a distribuição for ciclica haverá o falso compartilhamento pois terá interações na mesma linha de cache.

5.16. Embora strtok_r seja thread-safe, ele tem a propriedade bastante infeliz de modificar gratuitamente a string de entrada. Escreva um tokenizer que seja thread-safe e não modifique a string de entrada.

Uma forma de evitar que a string seja modificada quando utilizada na função strtok_r é realizar uma cópia da string (lines[i]) em outra variável - chamei ela de copia - e utilizar essa cópia na função. Assim, a cópia será modificada e não a string original.

```
• brunopaiva@Linux:~/ExercicieOpenMP/5.16$ ./Questao16 2 < arquivo.txt
Enter text
Thread 0 > line 0 = tiago alan
Thread 0 > token 0 = tiago
Thread 0 > token 1 = alan
Thread 0 > After tokenizing, my line = tiago
Thread 0 > line 2 = bruno
Thread 0 > token 0 = bruno
Thread 0 > After tokenizing, my line = bruno
Thread 0 > line 4 = caio
Thread 0 > token 0 = caio
Thread 0 > After tokenizing, my line = caio
Thread 0 > line 6 = pedro
Thread 0 > token 0 = pedro
Thread 0 > After tokenizing, my line = pedro
Thread 1 > line 1 = felipe
Thread 1 > token 0 = felipe
Thread 1 > After tokenizing, my line = felipe
Thread 1 > line 3 = kennedy
Thread 1 > token 0 = kennedy
Thread 1 > After tokenizing, my line = kennedy
Thread 1 > line 5 = gabriel
Thread 1 > token 0 = gabriel
Thread 1 > After tokenizing, my line = gabriel
Thread 1 > line 7 = ruan
Thread 1 > token 0 = ruan
Thread 1 > After tokenizing, my line = ruan
○ brunopaiva@Linux:~/ExercicieOpenMP/5.16$
```