



**UNIVERSIDADE FEDERAL DO RURAL DO SEMI-ÁRIDO
CENTRO MULTIDISCIPLINAR DE PAU DOS FERROS
DEPARTAMENTO DE ENGENHARIAS E TECNOLOGIA
PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA
DOCENTE: ÍTALO AUGUSTO SOUZA DE ASSIS**

ERIKY ABREU VELOSO

PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA

PAU DOS FERROS – RN
2025

QUESTÕES RESPONDIDAS

- ☒ QUESTÃO 20 - PESO 1;
- ☒ QUESTÃO 21 - PESO 1;
- ☒ QUESTÃO 22 - PESO 1;
- ☒ QUESTÃO 23 - PESO 1;
- ☒ QUESTÃO 24 - PESO 1;
- ☒ QUESTÃO 25 - PESO 1;
- ☒ QUESTÃO 26 - PESO 1;
- ☒ QUESTÃO 28 - PESO 1;
- ☒ QUESTÃO 30 - PESO 1;
- ☒ QUESTÃO 32 - PESO 2.

PROGRAMAÇÃO DE MEMÓRIA COMPARTILHADA COM OPENMP

• QUESTÃO 20

Baixe o arquivo `omp_trap_1.c` do site do livro e exclua a diretiva `critical`. Compile e execute o programa com cada vez mais threads e valores cada vez maiores de n .

(a) Quantas *threads* e quantos trapézios são necessários antes que o resultado esteja incorreto?

Considerando testes realizados fixando os valores de a e b da seguinte forma: $a = 10$ e $b = 100$, e variando os valores repassados no número de *threads* e trapézios (n), como exposto no Quadro 1, o erro significativo ocorreu a partir de 8 *threads* e 512 trapézios, entretanto, com 16 *threads* e 128, 256 e 1024 trapézios o código também apresentou saídas equivocadas.

Quadro 1. Saídas do código “omp_trap_1.c” com distintos valores de “ n ” e *threads*

Threads/Trapézios	128	256	512	1024
1	3.33007415771 484e+005	3.33001853942 871e+005	3.33000463485 718e+005	3.33000115871 429e+005
2	3.33007415771 484e+005	3.33001853942 871e+005	3.33000463485 718e+005	3.33000115871 429e+005
4	3.33007415771 484e+005	3.33001853942 871e+005	3.33000463485 718e+005	3.33000115871 429e+005
8	3.33007415771 484e+005	3.33001853942 871e+005	3.30135171175 003e+005	3.33000115871 429e+005
16	3.11533882141 113e+005	3.23560087680 817e+005	3.33000463485 718e+005	3.20789903551 340e+005

Válido ressaltar que todas as quantidades de *threads* e trapézios testadas foram na potência de 2, e o valor correto da área da figura calculada pode ser obtido através do cálculo:

$$\int_{10}^{100} x^2 dx = 333000$$

(b) Como o aumento do número de trapézios influencia nas chances do resultado ser incorreto?

Quando consideramos que a variável a ser analisada será a quantidade de trapézios, matematicamente, quanto maior a quantidade de trapézios, mais preciso é o resultado, entretanto, pela remoção da diretiva `critical`, quando maior o número de trapézios, maior vai

ser o número de acesso à variável `global_result_p`, aumentando a condição de corrida entre as *threads*. Logo, quanto maior o número de trapézios, maior a probabilidade do resultado ser equivocado.

(c) Como o aumento do número de *threads* influencia nas chances do resultado ser incorreto?

Em relação à quantidade de *threads*, a saída equivocada é provocada pela condição de corrida existente após a exclusão da diretiva `critical`, logo, com duas *threads* já existe a possibilidade de resultados incorretos para o cálculo da área da figura, entretanto, quanto maior o número de *threads*, maior o número de núcleos concorrendo pelo acesso à variável `global_result_p`.

- QUESTÃO 21

Baixe o arquivo `omp_trap1.c` do site do livro. Modifique o código para que

- ele use o primeiro bloco de código da página 222 do livro e
- o tempo usado pelo bloco paralelo seja 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 tempo no passado. Para obter detalhes sobre cronometragem, consulte a Seção 2.6.4. Lembre-se também de que o OpenMP possui uma diretiva de barreira:

```
# pragma omp barrier
```

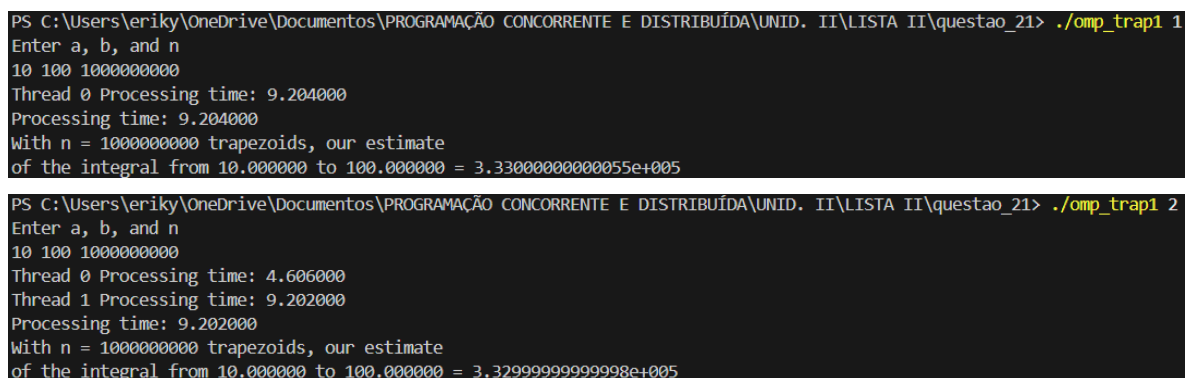
Agora encontre um sistema com pelo menos dois núcleos e cronometre o programa com

- uma thread e um grande valor de n , e
- duas threads e o mesmo valor de n .

(a) O que acontece?

Posteriormente realizar a execução cinco vezes consecutivas com uma e duas *threads* e um valor de n fixo em 1000000000, podemos observar que o valor da mediana com uma única threads é de 9,204 segundos. Após o incremento de mais uma *thread* o tempo de execução permaneceu quase inalterado, possuindo um tempo mediano de 9,202 segundos, como exposto na Figura 1.

Figura 1. Execução do código “`omp_trap1.c`” com diferentes números de threads



```
PS C:\Users\eriky\OneDrive\Documentos\PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA\UNID. II\LISTA II\questao_21> ./omp_trap1 1
Enter a, b, and n
10 100 1000000000
Thread 0 Processing time: 9.204000
Processing time: 9.204000
With n = 1000000000 trapezoids, our estimate
of the integral from 10.000000 to 100.000000 = 3.3300000000055e+005

PS C:\Users\eriky\OneDrive\Documentos\PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA\UNID. II\LISTA II\questao_21> ./omp_trap1 2
Enter a, b, and n
10 100 1000000000
Thread 0 Processing time: 4.606000
Thread 1 Processing time: 9.202000
Processing time: 9.202000
With n = 1000000000 trapezoids, our estimate
of the integral from 10.000000 to 100.000000 = 3.3299999999998e+005
```

A semelhança entre os tempos pode ser explicada, pois a função `Trap(a, b, n)`; está adicionando seu valor diretamente na variável global `global_result`, sendo necessário a adição de um `# pragma omp critical`, serializando o código, como exposto no Código 1. Logo, o código sequencial torna-se mais eficiente, por não possuir overhead de threads

Código 1. Trecho do “omp_trap1.c” (Modificado)

```
C/C++

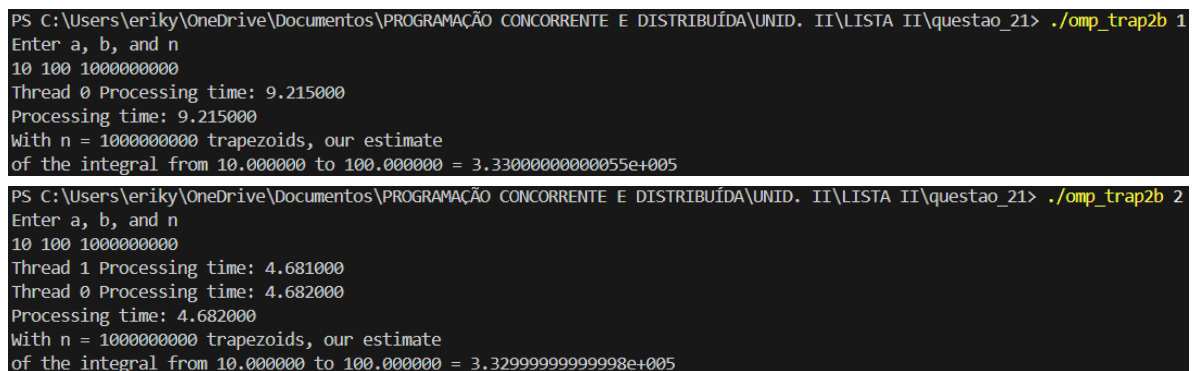
# pragma omp parallel num_threads(thread_count) \
    default (none) private(start, finish) shared(global_result, a, b,
n, global_time)
{
# pragma omp barrier
    start = omp_get_wtime();
# pragma omp critical (result)
    global_result += Trap(a, b, n);
    finish = omp_get_wtime();
    int thread_number = omp_get_thread_num();
    printf("Thread %d Processing time: %f \n", thread_number,
(finish-start));
    if(global_time < (finish-start)){
#        pragma omp critical (time)
        global_time = (finish-start);
    }
}

printf("Processing time: %f \n", global_time);
```

(b) Baixe o arquivo omp_trap2b.c do site do livro. Como seu desempenho se compara?

Analogamente ao teste passado, o código “omp_trap2b.c” foi executado a mesma quantidade de vezes, com os mesmos valores de a , b e n , obtendo-se a mediana do tempo com uma única thread de 9,215 segundos, e com duas threads o tempo mediano sofreu uma redução para 4,682 segundos, como exposto na Figura 2.

Figura 2. Execução do código “omp_trap2b.c” com diferentes números de threads



```
PS C:\Users\eriky\OneDrive\Documentos\PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA\UNID. II\LISTA II\questao_21> ./omp_trap2b 1
Enter a, b, and n
10 100 1000000000
Thread 0 Processing time: 9.215000
Processing time: 9.215000
With n = 1000000000 trapezoids, our estimate
of the integral from 10.000000 to 100.000000 = 3.330000000000055e+005

PS C:\Users\eriky\OneDrive\Documentos\PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA\UNID. II\LISTA II\questao_21> ./omp_trap2b 2
Enter a, b, and n
10 100 1000000000
Thread 1 Processing time: 4.681000
Thread 0 Processing time: 4.682000
Processing time: 4.682000
With n = 1000000000 trapezoids, our estimate
of the integral from 10.000000 to 100.000000 = 3.32999999999998e+005
```

Após os testes, notou-se que para uma thread o tempo de execução permanece semelhante ao código “omp_trap1.c”. Entretanto, com duas ou mais threads o código “omp_trap2b.c” torna-se mais eficiente, isso é possível pelo uso do comando `reduction(+:`

global_result), que remove a problemática da zona crítica do código anterior, pois nesse exemplo, o reduction cria cópias locais privadas da variável global_result para cada thread, e ao final da região paralela, realiza uma soma dos valores particulares, armazenando o resultado na variável original compartilhada.

Código 2. Trecho do “omp_trap2b.c” (Modificado)

C/C++

```
# pragma omp parallel num_threads(thread_count) \
    default (none) private(start, finish) shared(a, b, n,
global_time) reduction(+: global_result)
{
    # pragma omp barrier
    start = omp_get_wtime();
    global_result += Local_trap(a, b, n);
    finish = omp_get_wtime();
    int thread_number = omp_get_thread_num();
    printf("Thread %d Processing time: %f \n", thread_number,
(finish-start));
    if(global_time < (finish-start)){
        # pragma omp critical (time)
        global_time = (finish-start);
    }
}
```

• QUESTÃO 22

Suponha que no incrível computador Bleeblon, variáveis com 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 ele for executado no Bleeblon? Justifique sua resposta.

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

O resultado exposto após a execução do código é: “sum = 1010.0”. Mesmo após os operadores passarem pelas sete operações necessárias para a realização de uma soma, incluindo o truncamento realizado na sexta operação, não ocorre perda significativa de dados, resultando na soma correta, como exposto nas Tabelas 1 à 4.

Tabela 1. Iteração 1 - $i = 0$: $\text{sum} = 0.0$: $a[i] = 4.0$

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	0.000×10^0	4.000×10^0	
2	Compare exponents	0.000×10^0	4.000×10^0	
3	Shift one operand	0.000×10^0	4.000×10^0	
4	Add	0.000×10^0	4.000×10^0	4.000×10^0
5	Normalize result	0.000×10^0	4.000×10^0	4.000×10^0
6	Round result	0.000×10^0	4.000×10^0	4.00×10^0
7	Store result	0.000×10^0	4.000×10^0	4.00×10^0

Tabela 2. Iteração 2 - i = 1 : sum = 4.0 : a[i] = 3.0

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	4.000×10^0	3.000×10^0	
2	Compare exponents	4.000×10^0	3.000×10^0	
3	Shift one operand	4.000×10^0	3.000×10^0	
4	Add	4.000×10^0	3.000×10^0	7.000×10^0
5	Normalize result	4.000×10^0	3.000×10^0	7.000×10^0
6	Round result	4.000×10^0	3.000×10^0	7.00×10^0
7	Store result	4.000×10^0	3.000×10^0	7.00×10^0

Tabela 3. Iteração 3 - i = 2 : sum = 7.0 : a[i] = 3.0

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	7.000×10^0	3.000×10^0	
2	Compare exponents	7.000×10^0	3.000×10^0	
3	Shift one operand	7.000×10^0	3.000×10^0	
4	Add	7.000×10^0	3.000×10^0	10.00×10^0
5	Normalize result	7.000×10^0	3.000×10^0	1.000×10^1
6	Round result	7.000×10^0	3.000×10^0	1.00×10^1
7	Store result	7.000×10^0	3.000×10^0	1.00×10^1

Tabela 4. Iteração 4 - i = 3 : sum = 10.0 : a[i] = 1000.0

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	1.000×10^1	1.000×10^3	
2	Compare exponents	1.000×10^1	1.000×10^3	

3	Shift one operand	0.010×10^3	1.000×10^3	
4	Add	0.010×10^3	1.000×10^3	1.010×10^3
5	Normalize result	0.010×10^3	1.000×10^3	1.010×10^3
6	Round result	0.010×10^3	1.000×10^3	1.01×10^3
7	Store result	0.010×10^3	1.000×10^3	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 operacional atribua as iterações $i = 0, 1$ à thread 0 e $i = 2, 3$ à *thread* 1. Qual é a saída deste código no Bleeblon? Justifique sua resposta.

O resultado exposto após a execução do código é: “sum = 1000.0”. Diferentemente do bloco de código anterior, a execução vai resultar em um valor equivocado. Pode-se notar, como exposto na Tabela 8, que desde execução na *thread* 1 ocorre a perda de dados, a ausência do decimal afeta diretamente a função *reduction*, resultando em uma saída final errônea, como exposto na Tabela 9. Esse erro é ocasionado durante a operação 6, que o valor é truncado para 3 dígitos decimais, como especificado no computador Bleeblon.

Tabela 5. *thread* 0 - Iteração 1 - $i = 0$: sum = 0.0 : $a[i] = 4.0$

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	0.000×10^0	4.000×10^0	
2	Compare exponents	0.000×10^0	4.000×10^0	
3	Shift one operand	0.000×10^0	4.000×10^0	
4	Add	0.000×10^0	4.000×10^0	4.000×10^0
5	Normalize result	0.000×10^0	4.000×10^0	4.000×10^0
6	Round result	0.000×10^0	4.000×10^0	4.00×10^0
7	Store result	0.000×10^0	4.000×10^0	4.00×10^0

Tabela 6. *thread 0* - Iteração 2 - $i = 1$: $\text{sum} = 4.0$: $a[i] = 3.0$

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	4.000×10^0	3.000×10^0	
2	Compare exponents	4.000×10^0	3.000×10^0	
3	Shift one operand	4.000×10^0	3.000×10^0	
4	Add	4.000×10^0	3.000×10^0	7.000×10^0
5	Normalize result	4.000×10^0	3.000×10^0	7.000×10^0
6	Round result	4.000×10^0	3.000×10^0	7.00×10^0
7	Store result	4.000×10^0	3.000×10^0	7.00×10^0

Tabela 7. *thread 1* - Iteração 1 - $i = 2$: $\text{sum} = 0.0$: $a[i] = 3.0$

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	0.000×10^0	3.000×10^0	
2	Compare exponents	0.000×10^0	3.000×10^0	
3	Shift one operand	0.000×10^0	3.000×10^0	
4	Add	0.000×10^0	3.000×10^0	3.000×10^0
5	Normalize result	0.000×10^0	3.000×10^0	3.000×10^0
6	Round result	0.000×10^0	3.000×10^0	3.00×10^0
7	Store result	0.000×10^0	3.000×10^0	3.00×10^0

Tabela 8. *thread 1* - Iteração 2 - $i = 3$: $\text{sum} = 3.0$: $a[i] = 1000.0$

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	3.000×10^0	1.000×10^3	

2	Compare exponents	3.000×10^0	1.000×10^3	
3	Shift one operand	0.003×10^3	1.000×10^3	
4	Add	0.003×10^3	1.000×10^3	1.003×10^3
5	Normalize result	0.003×10^3	1.000×10^3	1.003×10^3
6	Round result	0.003×10^3	1.000×10^3	1.00×10^3
7	Store result	0.003×10^3	1.000×10^3	1.00×10^3

Tabela 9. Redução

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	7.000×10^0	1.000×10^3	
2	Compare exponents	7.000×10^0	1.000×10^3	
3	Shift one operand	0.007×10^3	1.000×10^3	
4	Add	0.007×10^3	1.000×10^3	1.007×10^3
5	Normalize result	0.007×10^3	1.000×10^3	1.007×10^3
6	Round result	0.007×10^3	1.000×10^3	1.00×10^3
7	Store result	0.007×10^3	1.000×10^3	1.00×10^3

- QUESTÃO 23

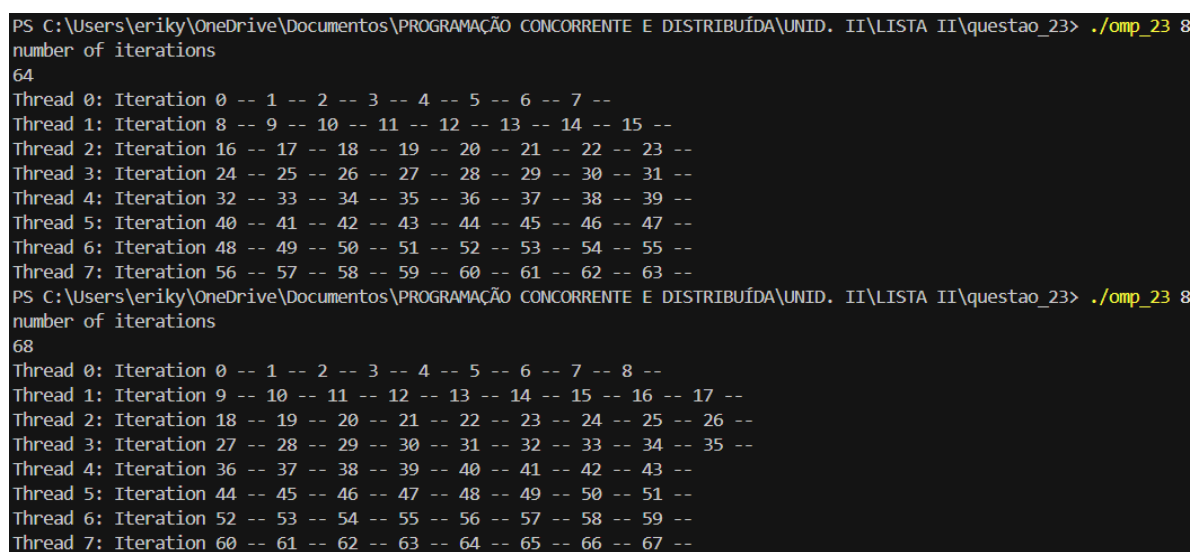
Escreva um programa OpenMP que determine o escalonamento padrão de laços `for` paralelos. Sua entrada deve ser o número de iterações e quantidade de threads e sua saída deve ser quais iterações de um laço `for` paralelizado são executadas por qual thread. Por exemplo, se houver duas threads e quatro iterações, a saída poderá ser:

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

(a) De acordo com a execução do seu programa, qual é o escalonamento padrão de laços `for` paralelos de um programa OpenMP? Porque?

Baseado na execução do programa, foi possível observar que com 64 iterações e 8 threads, cada thread realizou 8 iterações consecutivas, como exposto na Figura 3. Logo, o *chunksize* foi obtido realizando a divisão da quantidade de iterações pelo número de *threads*, sendo possível concluir que o escalonamento padrão é do tipo *Static*. É válido ressaltar, que caso a divisão não seja exata, é adicionada mais uma iteração nas primeiras *threads*.

Figura 3. Execução do código “omp_23.c” com diferentes números de iterações



```
PS C:\Users\eriky\OneDrive\Documentos\PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA\UNID. II\LISTA II\questao_23> ./omp_23 8
number of iterations
64
Thread 0: Iteration 0 -- 1 -- 2 -- 3 -- 4 -- 5 -- 6 -- 7 --
Thread 1: Iteration 8 -- 9 -- 10 -- 11 -- 12 -- 13 -- 14 -- 15 --
Thread 2: Iteration 16 -- 17 -- 18 -- 19 -- 20 -- 21 -- 22 -- 23 --
Thread 3: Iteration 24 -- 25 -- 26 -- 27 -- 28 -- 29 -- 30 -- 31 --
Thread 4: Iteration 32 -- 33 -- 34 -- 35 -- 36 -- 37 -- 38 -- 39 --
Thread 5: Iteration 40 -- 41 -- 42 -- 43 -- 44 -- 45 -- 46 -- 47 --
Thread 6: Iteration 48 -- 49 -- 50 -- 51 -- 52 -- 53 -- 54 -- 55 --
Thread 7: Iteration 56 -- 57 -- 58 -- 59 -- 60 -- 61 -- 62 -- 63 --
PS C:\Users\eriky\OneDrive\Documentos\PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA\UNID. II\LISTA II\questao_23> ./omp_23 8
number of iterations
68
Thread 0: Iteration 0 -- 1 -- 2 -- 3 -- 4 -- 5 -- 6 -- 7 -- 8 --
Thread 1: Iteration 9 -- 10 -- 11 -- 12 -- 13 -- 14 -- 15 -- 16 -- 17 --
Thread 2: Iteration 18 -- 19 -- 20 -- 21 -- 22 -- 23 -- 24 -- 25 -- 26 --
Thread 3: Iteration 27 -- 28 -- 29 -- 30 -- 31 -- 32 -- 33 -- 34 -- 35 --
Thread 4: Iteration 36 -- 37 -- 38 -- 39 -- 40 -- 41 -- 42 -- 43 --
Thread 5: Iteration 44 -- 45 -- 46 -- 47 -- 48 -- 49 -- 50 -- 51 --
Thread 6: Iteration 52 -- 53 -- 54 -- 55 -- 56 -- 57 -- 58 -- 59 --
Thread 7: Iteration 60 -- 61 -- 62 -- 63 -- 64 -- 65 -- 66 -- 67 --
```

Código 3. Código “omp_23.c”

```
C/C++

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

void Usage(char* prog_name);
```

```

int main(int argc, char* argv[]) {
    int thread_count, thread_number, n;
    int *thread_interactions;

    if (argc != 2) Usage(argv[0]);
    thread_count = strtol(argv[1], NULL, 10);
    printf("number of iterations\n");
    scanf("%d", &n);

    thread_interactions = (int *)malloc(sizeof(int) * n);

    # pragma omp parallel num_threads(thread_count) \
        default(none) private(thread_number) shared(thread_interactions, n)
    {
        # pragma omp for
        for (int i = 0; i < n; i++) {
            thread_number = omp_get_thread_num();
            thread_interactions[i] = thread_number;
        }

        for(int k = 0; k < thread_count; k++){
            printf("Thread %d: Iteration ", k);
            for (int i = 0; i < n; i++) {
                if (thread_interactions[i] == k)
                {
                    printf( "%d -- ", i);
                }
            }
            printf("\n");
        }

        free(thread_interactions);
        return 0;
    } /* main */

    void Usage(char* prog_name) {
        fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
        exit(0);
    } /* Usage */
}

```

- QUESTÃO 24

Considere o seguinte laço:

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

Há claramente uma dependência no laço já que o valor de $a[i]$ não pode ser calculado sem o valor de $a[i-1]$. Sugira uma maneira de eliminar essa dependência e paralelizar o laço.

Estipulando distintos valores de n , buscando compreender o funcionamento do trecho de código, e encontrar uma fórmula fechada que realiza a mesma função da linha “ $a[i] = a[i-1] + i$ ”, observou-se que o laço apresentado realiza um somatório de uma sequência de número inteiros em um intervalo de 0 à n , logo a equação presente no `for` pode ser substituída pela fórmula da soma dos números naturais exposta a seguir:

$$\frac{n(n+1)}{2}$$

Realizando a substituição da linha de código “ $a[i] = a[i-1] + i$ ” por “ $a[i] = (i*(i + 1))/2$ ” removemos a dependência existente, pois o único valor necessário para o cálculo é o do índice, não necessitando de informações de índices anteriores como ocorria com $a[i-1]$. A seguir, uma exemplificação do código paralelizado.

Código 4. Trecho do código “omp_24.c”

C/C++

```
explicit_summation = (int *)malloc(sizeof(int) * n);

# pragma omp parallel for num_threads(thread_count) \
    default(none) shared(explicit_summation, n)
for (int i = 0; i < n; i++) {
    explicit_summation[i] = (i*(i + 1))/2;
}

for (int i = 0; i < n; i++) {
    printf("Index: %d - Sum: %d \n", i ,
explicit_summation[i]);
}
```

- QUESTÃO 25

Modifique o programa da regra do trapézio 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 thread. Isso pode ser feito alocando um *array* `iteracoes` de n int's e, na função `Trap`, atribuindo `omp_get_thread_num()` a `iteracoes[i]` na i -ésima iteração do laço `for`. Qual é o escalonamento padrão de iterações em seu sistema? Como o escalonamento `guided` é determinado?

O trecho de código exposto a seguir, expõe a modificação realizada na função `trap`, do código `omp_trap_3.c`, com o intuito de aplicar a cláusula `schedule(runtime)`.

Código 5. Trecho do código modificado (`omp_trap_3.c`)

```
C/C++

double Trap(double a, double b, int n, int thread_count) {
    double h, approx;
    int i;
    int *thread_number;

    thread_number = (int *)malloc(sizeof(int) * n);

    h = (b-a)/n;
    approx = (f(a) + f(b))/2.0;
    # pragma omp parallel for num_threads(thread_count) \
        reduction(+: approx) schedule(runtime)
    for (i = 1; i <= n-1; i++){
        approx += f(a + i*h);
        thread_number[i-1] = omp_get_thread_num();
        printf("Thread number: %d - Iteration: %d \n",
            thread_number[i-1], i);
    }
    approx = h*approx;

    free(thread_number);
    return approx;
} /* Trap */
```

Durante todas as execuções do programa, atribuindo diferentes tipos à variável de ambiente `OMP_SCHEDULE`, a quantidade de *threads* (`thread_count`), os limites inferiores (a) e superiores (b), e os números de trapézios (n) foram fixados em 4, 10, 100 e 16 respectivamente. Válido ressaltar que o número de iterações vai ser o um valor a menos que o número de *threads*.

No primeiro teste realizado, não foi declarado o `OMP_SCHEDULE`, com o intuito de se obter o escalonamento padrão de iterações. Realizando a análise das saídas, como exposto na Figura 4, o possível tipo de escalonamento é o `dynamic` com `chunksize` de 1.

Figura 4. Execução do código “omp_trap_3.c”, sem declarar o `OMP_SCHEDULE`

```
PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_25> ./omp_trap3 4
Enter a, b, and n
10 100 16
Thread number: 0 - Iteration: 1
Thread number: 0 - Iteration: 5
Thread number: 0 - Iteration: 6
Thread number: 0 - Iteration: 7
Thread number: 0 - Iteration: 8
Thread number: 0 - Iteration: 9
Thread number: 0 - Iteration: 10
Thread number: 0 - Iteration: 11
Thread number: 0 - Iteration: 12
Thread number: 0 - Iteration: 13
Thread number: 0 - Iteration: 14
Thread number: 0 - Iteration: 15
Thread number: 2 - Iteration: 3
Thread number: 1 - Iteration: 2
Thread number: 3 - Iteration: 4
With n = 16 trapezoids, our estimate
of the integral from 10.000000 to 100.000000 = 3.33474609375000e+005
```

No escalonamento do tipo `auto`, em que o próprio compilador decide qual escalonamento aplicar, foi possível observar na Figura 5 a execução de um escalonamento do tipo `static`, no qual o `chunksize` foi definido pela quantidade de iterações sobre o número de threads, contudo em caso de resto, as primeiras threads realizam mais iterações.

Figura 5. Execução do código “omp_trap_3.c”, `OMP_SCHEDULE = “auto”`

```
PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_25> $env:OMP_SCHEDULE="auto"
PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_25> ./omp_trap3 4
Enter a, b, and n
10 100 16
Thread number: 0 - Iteration: 1
Thread number: 1 - Iteration: 5
Thread number: 1 - Iteration: 6
Thread number: 1 - Iteration: 7
Thread number: 1 - Iteration: 8
Thread number: 2 - Iteration: 9
Thread number: 2 - Iteration: 10
Thread number: 2 - Iteration: 11
Thread number: 2 - Iteration: 12
Thread number: 0 - Iteration: 2
Thread number: 0 - Iteration: 3
Thread number: 0 - Iteration: 4
Thread number: 3 - Iteration: 13
Thread number: 3 - Iteration: 14
Thread number: 3 - Iteration: 15
With n = 16 trapezoids, our estimate
of the integral from 10.000000 to 100.000000 = 3.33474609375000e+005
```

Realizando a execução com a variável `OMP_SCHEDULE` declarada como `static`, no qual neste exemplo o *chunksize* foi definido em dois, o escalonamento dividiu as iterações igualmente entre as threads em blocos fixos de dois, como observado na Figura 6, onde os intervalos são direcionados ciclicamente de forma ordenada baseada na numeração da thread.

Figura 6. Execução do código “omp_trap_3.c”, `OMP_SCHEDULE = “static,2”`

```
PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_25> $env:OMP_SCHEDULE="static, 2"
PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_25> ./omp_trap3 4
Enter a, b, and n
10 100 16
Thread number: 2 - Iteration: 5
Thread number: 2 - Iteration: 6
Thread number: 2 - Iteration: 13
Thread number: 2 - Iteration: 14
Thread number: 3 - Iteration: 7
Thread number: 3 - Iteration: 8
Thread number: 3 - Iteration: 15
Thread number: 0 - Iteration: 1
Thread number: 0 - Iteration: 2
Thread number: 0 - Iteration: 9
Thread number: 0 - Iteration: 10
Thread number: 1 - Iteration: 3
Thread number: 1 - Iteration: 4
Thread number: 1 - Iteration: 11
Thread number: 1 - Iteration: 12
With n = 16 trapezoids, our estimate
of the integral from 10.000000 to 100.000000 = 3.33474609375000e+005
```

O método de escalonamento `guided` possui como característica principal a inicialização com blocos grandes de dados, no qual sofrem redução a medida que ocorre as iterações, outro atributo, é o fato das iterações serem atribuídas às threads enquanto o loop está em execução, em outras palavras, a thread que estiver ociosa, recebe o próximo bloco de iterações para executar. Analisando a saída exposta na Figura 7, é possível observar que a thread 3 recebe um bloco de 4 iterações, seguido pela thread 0, que recebe um bloco de 3 iterações, e todo o processo a seguir recebem blocos de 2 iterações, por ser o tamanho mínimo do bloco repassado na função.

Figura 7. Execução do código “omp_trap_3.c”, OMP_SCHEDULE = “guided,2”

```
PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_25> $env:OMP_SCHEDULE="guided, 2"
PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_25> ./omp_trap3 4
Enter a, b, and n
10 100 16
Thread number: 3 - Iteration: 1
Thread number: 3 - Iteration: 2
Thread number: 3 - Iteration: 3
Thread number: 3 - Iteration: 4
Thread number: 0 - Iteration: 5
Thread number: 0 - Iteration: 6
Thread number: 0 - Iteration: 7
Thread number: 0 - Iteration: 14
Thread number: 0 - Iteration: 15
Thread number: 2 - Iteration: 10
Thread number: 2 - Iteration: 11
Thread number: 1 - Iteration: 8
Thread number: 1 - Iteration: 9
Thread number: 3 - Iteration: 12
Thread number: 3 - Iteration: 13
With n = 16 trapezoids, our estimate
of the integral from 10.000000 to 100.000000 = 3.33474609375000e+005
```

Por fim, realizou-se a execução com o OMP_SCHEDULE declarado como dynamic, nesse método de escalonamento, semelhante ao anterior, as iterações são atribuídas às threads enquanto o loop está em execução, entretanto, não se é realizado a variação no valor do tamanho dos blocos, sendo fixo e definido pelo *chunksize*, neste exemplo, o valor foi fixado em dois, como exposto na Figura 8.

Figura 8. Execução do código “omp_trap_3.c”, OMP_SCHEDULE = “dynamic,2”

```
PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_25> $env:OMP_SCHEDULE="dynamic,2"
PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_25> ./omp_trap3 4
Enter a, b, and n
10 100 16
Thread number: 0 - Iteration: 1
Thread number: 0 - Iteration: 2
Thread number: 1 - Iteration: 5
Thread number: 1 - Iteration: 6
Thread number: 2 - Iteration: 3
Thread number: 0 - Iteration: 9
Thread number: 3 - Iteration: 7
Thread number: 3 - Iteration: 8
Thread number: 2 - Iteration: 4
Thread number: 2 - Iteration: 15
Thread number: 1 - Iteration: 11
Thread number: 1 - Iteration: 12
Thread number: 0 - Iteration: 10
Thread number: 3 - Iteration: 13
Thread number: 3 - Iteration: 14
With n = 16 trapezoids, our estimate
of the integral from 10.000000 to 100.000000 = 3.33474609375000e+005
```

- QUESTÃO 26

Lembre-se de que todos os blocos estruturados modificados por uma diretiva `critical` formam uma única seção crítica. O que acontece se tivermos um número de diretivas `atomic` nas quais diferentes variáveis estão sendo modificadas? Todas elas são tratadas como uma única seção crítica?

Podemos escrever um pequeno programa que tente determinar isso. A ideia é fazer com que todas as threads executem simultaneamente algo como o código a seguir:

```
int i;
double minha_soma = 0.0;
for (i = 0; i < n; i++)
    #pragma omp atomic
    minha_soma += sin(i);
```

Podemos fazer isso modificando o código com uma diretiva `parallel`:

```
# pragma omp parallel num_threads(thread_count)
{
    int i;
    double minha_soma = 0.0;
    for (i = 0; i < n; i++)
        #pragma omp atomic
        minha_soma += sin(i);
}
```

Observe que já que `minha_soma` e `i` são declaradas no bloco paralelo, cada thread possui sua própria cópia privada. Agora, se medirmos o tempo desse código para um `n` grande com `thread_count = 1` e também quando `thread_count > 1`, contanto que `thread_count` seja menor que o número de núcleos disponíveis, o tempo de execução para a execução de *thread* única deveria ser aproximadamente o mesmo que o tempo para a execução com múltiplas threads se as diferentes execuções de `minha_soma += sin(i)` são tratadas como diferentes seções críticas. Por outro lado, se as diferentes execuções de `minha_soma += sin(i)` são todas tratadas como uma única seção crítica, a execução com múltiplas *threads* deve ser muito mais lenta que a execução de *thread* única. Escreva um programa OpenMP que implemente este teste. Sua implementação do OpenMP permite a execução simultânea de atualizações para diferentes variáveis quando as atualizações são protegidas por diretivas `atomic`?

Diferentemente da diretiva `critical`, o `#pragma omp atomic` afeta somente a operação e a variável específica, logo, diferentes variáveis não são tratadas como uma única seção crítica, sendo possível serem modificadas em paralelo por *threads* diferentes, pois OpenMP trata cada `atomic` individualmente. O Código 6, exemplifica o uso do `atomic`.

Código 6. Código solicitado na questão 26 (omp_26.c)

```
C/C++
#include <stdio.h>
```

```

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

void Usage(char* prog_name);

int main(int argc, char* argv[]) {
    int thread_count, n;
    double global_time;

    if (argc != 2) Usage(argv[0]);
    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter the value of n\n");
    scanf("%d", &n);

    # pragma omp parallel num_threads(thread_count) \
    default (none) shared(global_time, n)
    {
        double start, finish;

    #   pragma omp barrier
        start = omp_get_wtime();
        int i, thread_number;
        double my_sum = 0.0;

        for (i = 1; i <= n; i++) {
    #       pragma omp atomic
            my_sum += sin(i);
        }

        thread_number = omp_get_thread_num();
        printf("Thread: %d - My sum: %.2f\n", thread_number, my_sum);
        finish = omp_get_wtime();
        printf("Thread %d Processing time: %f \n", thread_number,
(finish-start));

    #   pragma omp critical
    {
        if(global_time < (finish-start)){
            global_time = (finish-start);
        }
    }

    printf("Processing time: %f \n", global_time);

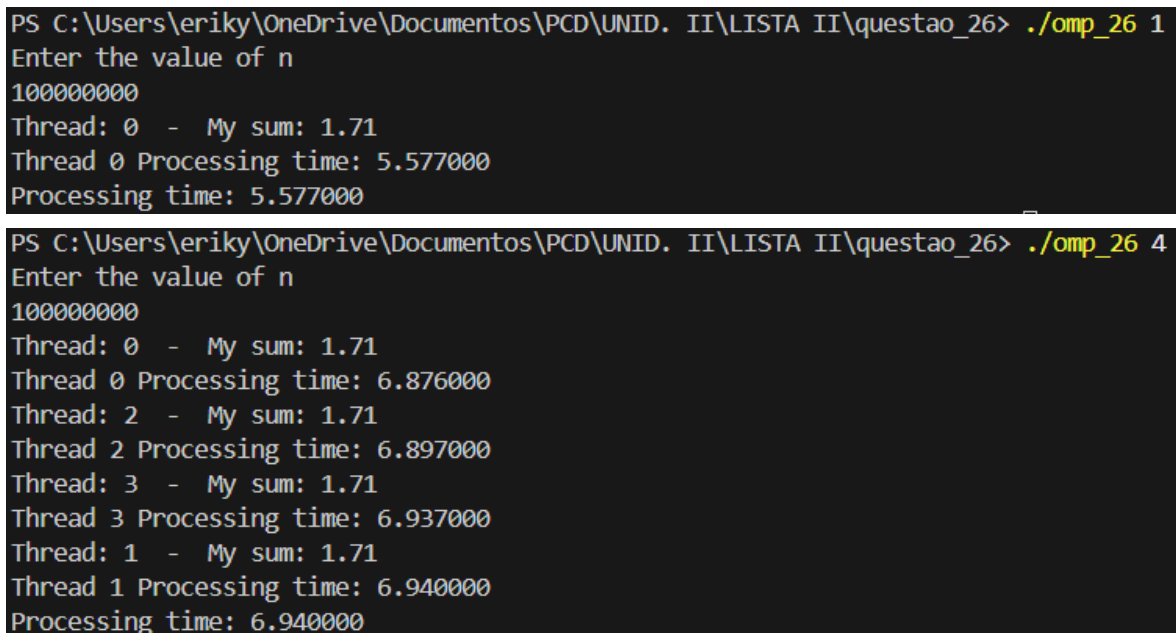
    return 0;
}

```

```
} /* main */
```

Utilizando o código supracitado, fixando o valor de $n = 100.000.000$ e realizando testes variando a quantidade de *threads*, pode-se observar, como exposto na Figura 9, o tempo mediano com uma e quatro *threads* são semelhantes, possuindo uma pequena diferença de aproximadamente 1,37 segundos. Logo, podemos considerar que a implementação do OpenMP realizada permite a execução simultânea de atualizações para as diferentes variáveis privadas de cada *thread*, quando as atualizações são protegidas por diretivas *atomic*.

Figura 9. Execução do código “omp_26.c”



The figure consists of two screenshots of a Windows command prompt window. The first screenshot shows the execution of the program with 1 thread. The second screenshot shows the execution with 4 threads. Both screenshots show the same input for 'n' (100000000) and a similar processing time (around 5.57 seconds for 1 thread and 6.94 seconds for 4 threads).

```
PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_26> ./omp_26 1
Enter the value of n
100000000
Thread: 0 - My sum: 1.71
Thread 0 Processing time: 5.577000
Processing time: 5.577000

PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_26> ./omp_26 4
Enter the value of n
100000000
Thread: 0 - My sum: 1.71
Thread 0 Processing time: 6.876000
Thread: 2 - My sum: 1.71
Thread 2 Processing time: 6.897000
Thread: 3 - My sum: 1.71
Thread 3 Processing time: 6.937000
Thread: 1 - My sum: 1.71
Thread 1 Processing time: 6.940000
Processing time: 6.940000
```

A título de comparação, quando executado o mesmo teste, entretanto utilizando `#pragma omp critical`, o tempo médio de execução com quatro *threads* é aproximadamente 6,8 vezes maior do que o tempo com uma única *thread*, como exposto na Figura 10, deixando evidente a zona crítica única, mesmo trabalhando com variáveis privadas distintas.

Figura 10. Execução do código “omp_26.c” modificado para a diretiva critical

```
PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_26> ./omp_26_copia 1
Enter the value of n
100000000
Thread: 0 - My sum: 1.71
Thread 0 Processing time: 5.482000
Processing time: 5.482000

PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_26> ./omp_26_copia 4
Enter the value of n
100000000
Thread: 2 - My sum: 1.71
Thread 2 Processing time: 36.640000
Thread: 1 - My sum: 1.71
Thread 1 Processing time: 36.709000
Thread: 0 - My sum: 1.71
Thread 0 Processing time: 37.264000
Thread: 3 - My sum: 1.71
Thread 3 Processing time: 37.341000
Processing time: 37.341000
```

- QUESTÃO 28

Lembre-se do exemplo de multiplicação de matrizes e vetores com a entrada 8000×8000 . Assuma que uma linha de cache contém 64 *bytes* ou 8 *doubles*.

(a) Suponha que a *thread* 0 e a *thread* 2 sejam atribuídas a processadores diferentes. É possível que ocorra um falso compartilhamento entre as *threads* 0 e 2 para alguma parte do vetor *y*? Por que?

Sim, considerando o Código 7, e sabendo que o processo envolve multiplicar cada linha da matriz (*i*) pelos elementos do vetor e somar os resultados, gerando cada elemento do novo vetor. Pode-se afirmar que se aplicado um `schedule(static, 1)` ou até mesmo um `schedule(dynamic, 1)`, é possível ocorrer um falso compartilhamento. Exemplo: suponto que a *thread* 0 realiza a multiplicação e soma da linha 0 da matriz com o primeiro elemento do vetor, armazenando seu resultado no `y[0]`. Sabemos que uma linha de cache contém 64 *bytes* ou 8 *doubles*, logo a *thread* 0 busca para sua cache as 8 primeiras posições do vetor (`y[0-7]`), visando alterar a informação `y[0]`, forçando as outras *threads* invalidarem e recarregarem a linha. Paralelamente, considerando um escalonamento do tipo `static` com *chunksize* de 1, a *thread* 2 está responsável por realizar a soma e multiplicação da linha 2 da matriz, pelo terceiro elemento do vetor, armazenando em `y[2]`, ao tentar realizar a escrita, ele percebe que a sua linha de cache está invalidada pela *thread* 0, forçando o acesso à memória RAM para atualização das informações, pois mesmo sem precisar manipular os mesmos dados, eles pertencem a mesma linha de cache, provocando um falso compartilhamento entre as *threads*.

Código 7. Trecho de código para a multiplicação de matrizes por vetores (omp_26.c)

```
C/C++
# pragma omp parallel for num_threads(thread_count) \
    default(none) private(i, j) shared(A, x, y, m, n)
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

(b) E se a *thread* 0 e a *thread* 3 forem atribuídas a processadores diferentes? É possível que ocorra um falso compartilhamento entre elas para alguma parte de *y*?

Analogamente a questão anterior, também existe a possibilidade de ocorrer o falso compartilhamento entre a *thread* 0 e a *thread* 3, em um escalonamento cíclico do tipo `static` com *chunksize* de 1, a divisão da escrita realizada no vetor `y[i]` ocorre da seguinte forma:

Thread 0 : y[0] y[4] ... y[...]
Thread 1 : y[1] y[5] ... y[...]
Thread 2 : y[2] y[6] ... y[...]
Thread 3 : y[3] y[7] ... y[...]

Sabendo que uma linha de cache possui 8 doubles, podemos afirmar que com essa configuração de escalonamento, existe a possibilidade de falso compartilhamento.

- QUESTÃO 30

Utilizando OpenMP, implemente o programa paralelo do histograma discutido no Capítulo 2.

Como solicitado no Capítulo 2 e utilizando como base o código sequencial disponibilizado, o Código 8 é responsável pela criação de um histograma construído paralelamente entre as threads, mantendo o foco na paralelização do loop que busca alocar um valor do vetor data em uma das bin.

Código 8. Código do histograma paralelizado

C/C++

```
int main(int argc, char* argv[]) {
    int bin_count = 20;
    int i, bin, thread_count/*, thread_number*/;
    float min_meas, max_meas;
    float bin_maxes[20];
    int bin_counts[20];
    int data_count;
    float* data;

    /* Check and get command line args */
    if (argc != 5) Usage(argv[0]);
    Get_args(argv, &min_meas, &max_meas, &data_count,
    &thread_count);

    /* Allocate arrays needed */
    data = malloc(data_count*sizeof(float));

    /* Generate the data */
    Gen_data(min_meas, max_meas, data, data_count);

    /* Create bins for storing counts */
    Gen_bins(min_meas, max_meas, bin_maxes, bin_counts,
    bin_count);

    /* Count number of values in each bin */
    # pragma omp parallel num_threads(thread_count) \
    reduction(+: bin_counts) default(none) \
    shared(data, bin_maxes, data_count, bin_count, min_meas)
    private(i, bin/*, thread_number*/)
    {
        # pragma omp for
        for (i = 0; i < data_count; i++) {
            bin = Which_bin(data[i], bin_maxes, bin_count,
            min_meas);

            //thread_number = omp_get_thread_num();
```

```

        //printf("Thread number: %d -> Value: %.2f \n",
thread_number, data[i]);
        bin_counts[bin]++;
    }
}

/* Print the histogram */
Print_histo(bin_maxes, bin_counts, bin_count, min_meas);

free(data);
return 0;

} /* main */

```

Válido salientar que as funções sobrescritas no código 8, exercem as seguintes funções:

- void Usage: exibe uma mensagem de uso do programa caso os argumentos da linha de comando estejam incorretos e encerra o programa.
- void Get_args: extrai e converte os argumentos fornecidos na linha de comando.
- void Gen_data: gera valores aleatórios em ponto flutuante dentro do intervalo [min_meas, max_meas] e armazena no *array* data.
- void Gen_bins: calcula os valores máximos para cada bin (bin_maxes) com base no número de bins e no intervalo dos dados, e inicializa o array bin_counts com zeros.
- int Which_bin: usa busca binária para determinar qual bin um dado específico pertence.
- void Print_histo: imprime o histograma na tela.

- QUESTÃO 32

Count sort é um algoritmo de ordenação serial simples que pode ser implementado da seguinte forma:

```
void Count_sort(int a[], int n) {
    int i, j, count;
    int* temp = malloc(n*sizeof(int));
    for (i = 0; i < n; i++) {
        count = 0;
        for (j = 0; j < n; j++)
            if (a[j] < a[i])
                count++;
            else if (a[j] == a[i] && j < i)
                count++;
        temp[count] = a[i];
    }
    memcpy(a, temp, n*sizeof(int));
    free(temp);
}
```

A ideia básica é que para cada elemento $a[i]$ na lista a , contemos o número de elementos da lista que são menores que $a[i]$. Em seguida, inserimos $a[i]$ em uma lista temporária usando o índice determinado pela contagem. Há um pequeno problema com esta abordagem quando a lista contém elementos iguais, uma vez que eles podem ser atribuídos ao mesmo slot na lista temporária. O código lida com isso incrementando a contagem de elementos iguais com base nos índices. Se $a[i] == a[j]$ e $j < i$, então contamos $a[j]$ como sendo "menor que" $a[i]$.

Após a conclusão do algoritmo, sobrescrevemos o *array* original pelo *array* temporário usando a função da biblioteca de *strings* `memcpy`.

(a) Se tentarmos paralelizar o laço `for i` (o laço externo), quais variáveis devem ser privadas e quais devem ser compartilhadas?

Compartilhadas: `shared(temp, a, n)`

Privadas: `private(count, i, j)`

(b) Se paralelizarmos o laço `for i` usando o escopo especificado na parte anterior, haverá alguma dependência de dados no laço? Explique sua resposta.

Caso o laço `for i` seja paralelizado não haverá dependência, pois durante a execução do laço, cada *thread* ficará responsável por um intervalo de valores de $a[i]$. Durante a execução, nenhuma *thread* precisa acessar ou modificar elementos adjacentes como $a[i-1]$ ou $a[i+1]$, garantindo assim a independência entre as iterações e permitindo a paralelização segura do laço.

(c) Podemos paralelizar a chamada para `memcpy`? Podemos modificar o código para que esta parte da função seja paralelizável?

Sim, é possível realizar a sobrescrita do vetor com a função `memcpy` de maneira paralelizada, particionando o tamanho do vetor `temp`, alocando estaticamente para as diferentes threads, como exposto no Código 9.

Código 9. Trecho de código para paralelizar a chamada para `memcpy`

C/C++

```
#pragma omp barrier
int rest = n % thread_count;
int thread_number = omp_get_thread_num();
int interval, start;
if(rest == 0){
    interval = n / thread_count;
    start = thread_number * interval;
}else{
    if (thread_number < rest){
        interval = (n / thread_count) + 1;
        start = (thread_number * (n/thread_count)) +
thread_number;
    }else{
        interval = (n / thread_count);
        start = (thread_number * interval) + rest;
    }
}

memcpy(&a[start], &temp[start], interval*sizeof(int));
}
```

(d) Escreva um programa em C que inclua uma implementação paralela do *Count sort*.

Código 10. Implementação paralela do Count sort

C/C++

```
void Count_sort_parallel(int a[], int n, int thread_count) {
    int i, j, count;
    int *temp = malloc(n*sizeof(int));

    # pragma omp parallel num_threads(thread_count) \
```

```

        default (none) shared(temp, a, n, thread_count)
private(count, i, j)
    {
        # pragma omp for
        for (i = 0; i < n; i++) {
            count = 0;
            for (j = 0; j < n; j++){
                if (a[j] < a[i])
                    count++;
                else if (a[j] == a[i] && j < i)
                    count++;
            }
            temp[count] = a[i];
        }

        # pragma omp barrier
        int rest = n % thread_count;
        int thread_number = omp_get_thread_num();
        int interval, start;
        if(rest == 0){
            interval = n / thread_count;
            start = thread_number * interval;
        }else{
            if (thread_number < rest){
                interval = (n / thread_count) + 1;
                start = (thread_number * (n/thread_count)) +
thread_number;
            }else{
                interval = (n / thread_count);
                start = (thread_number * interval) + rest;
            }
        }

        memcpy(&a[start], &temp[start], interval*sizeof(int));
    }

    free(temp);
} /*Count_sort_parallel*/

```

(e) Como o desempenho da sua paralelização do *Count sort* se compara à classificação serial? Como ela se compara à função serial `qsort`?

Como exposto no Código 11, para realizar a comparação de desempenho entre as três metodologias de ordenação, foram repassadas três vetores com o mesmo tamanho, e

preenchidos com os mesmos elementos, no qual, para cada método foi calculado o tempo de execução, sendo possível verificar o mais eficiente.

Código 11. Ordenação de elementos de um vetor - “omp_32.c”

C/C++

```
int main(int argc, char* argv[]) {
    int thread_count, n, x;
    double start_time, end_time;
    int *a, *b, *c;

    if (argc != 2) Usage(argv[0]);
    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter the amount of data\n");
    scanf("%d", &n);

    a = (int*)malloc(sizeof(int) * n);
    b = (int*)malloc(sizeof(int) * n);
    c = (int*)malloc(sizeof(int) * n);

    for(int i = 0; i < n; i++){
        x = rand();
        a[i] = x;
        b[i] = x;
        c[i] = x;
    }

    start_time = omp_get_wtime();
    Count_sort(a, n);
    end_time = omp_get_wtime();
    printf("Time count sort: %f \n", (end_time - start_time));

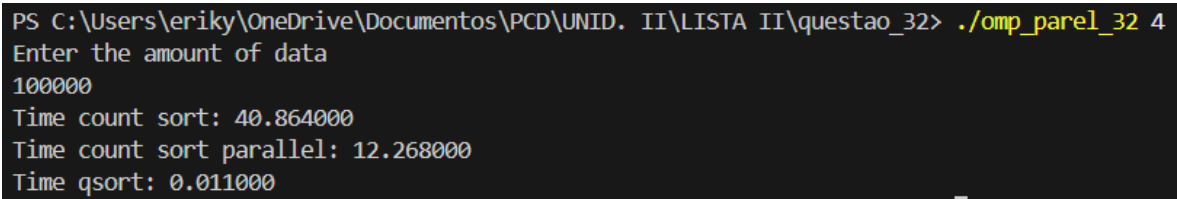
    start_time = omp_get_wtime();
    Count_sort_parallel(b, n, thread_count);
    end_time = omp_get_wtime();
    printf("Time count sort parallel: %f \n", (end_time - start_time));

    start_time = omp_get_wtime();
    qsort(c, n, sizeof(int), compare);
    end_time = omp_get_wtime();
    printf("Time qsort: %f \n", (end_time - start_time));

    free(a);
    free(b);
    free(c);
    return 0;
} /* main */
```

Após realizar a execução, notou-se que o função serial `qsort` mostrou-se mais eficiente, com um tempo consideravelmente inferior aos outros métodos. Além disso, como desejado, o *Count sort* paralelizado apresentou uma eficiência maior, quando comparado com o serializado, apresentando um tempo de aproximadamente 3,3 vezes menor, como exposto na Figura 11.

Figura 11. Execução do código “omp_32.c”

A terminal window with a black background and white text. The prompt is 'PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_32>'. The user enters './omp_parel_32 4'. The program outputs 'Enter the amount of data', then '100000'. It then displays three timing results: 'Time count sort: 40.864000', 'Time count sort parallel: 12.268000', and 'Time qsort: 0.011000'.

```
PS C:\Users\eriky\OneDrive\Documentos\PCD\UNID. II\LISTA II\questao_32> ./omp_parel_32 4
Enter the amount of data
100000
Time count sort: 40.864000
Time count sort parallel: 12.268000
Time qsort: 0.011000
```