Universidade Federal do Rural do Semi-Árido
Centro Multidisciplinar de Pau dos Ferros
Departamento de Engenharias e Tecnologia
PEX1272 - Programação Concorrente e Distribuída
Professor: Ítalo Assis

Lista de Exercícios

QUESTÃO 20 - FEITA (PESO 1)

QUESTÃO 21 - FEITA (PESO 1)

QUESTÃO 22 - FEITA (PESO 1)

QUESTÃO 23 - FEITA (PESO 1)

QUESTÃO 24 - FEITA (PESO 1)

QUESTÃO 25 - FEITA (PESO 1)

QUESTÃO 28 - FEITA (PESO 1)

QUESTÃO 31 - FEITA (PESO 2)

QUESTÃO 32 - FEITA (PESO 2)

Programação de memória compartilhada com OpenMP

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? **Nos meus testes obtive erro com 4 threads a partir do tamanho 200.**

(b) Como o aumento do número de trapézios influencia nas chances do resultado ser incorreto? **Mais trapezios também significam mais iterações por thread, e, mais iterações signifca que cada thread irá realizar mais somas globais, acessando a região critica mais vezes.**

(c) Como o aumento do número de *threads* influencia nas chances do resultado ser incorreto? **Nós temos uma condição de corrida e então quanto maior a quantidade de threads maior as chances delas realizarem a soma global ao mesmo tempo.**

P/N	100	200	400	800

1	3.33350000000000e-01	3.33337500000000e-01	3.33334375000000e-01	3.33333593750000e-01
2	3.33350000000000e-01	3.33337500000000e-01	3.33334375000000e-01	3.33333593750000e-01
4	3.33350000000000e-01	2.34378125000000e-01	4.16671875000000e-02	2.34375195312500e-01

21. Baixe o arquivo omp_trap_1.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 pas sado. Para obter detalhes sobre cronometragem, consulte a Seção 2.6.4. Lembre-se também de que o OpenMP possui uma diretiva de barreira:

  # barreira pragma omp

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

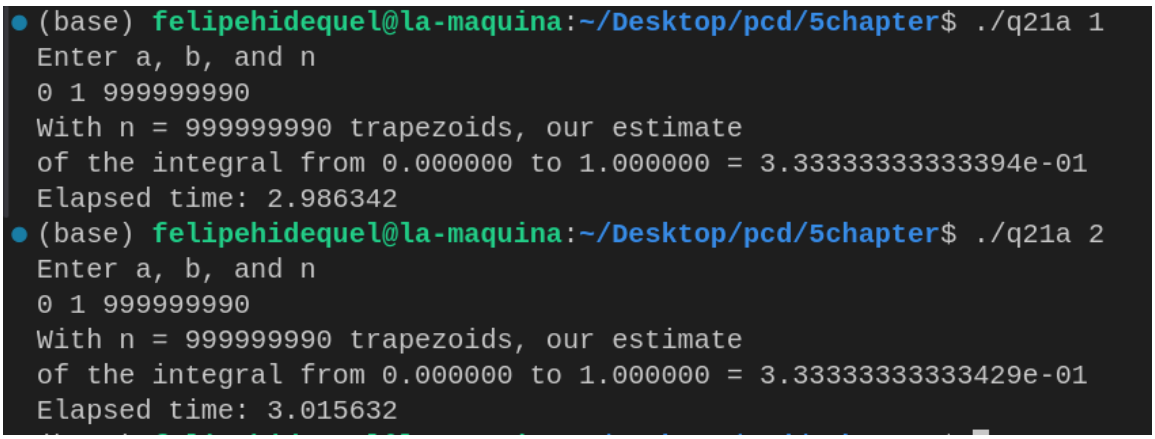
- uma *thread* e um grande valor de *n*, e
- duas *threads* e o mesmo valor de *n*.

(a) O que acontece?

```
  double start, end;
  start = omp_get_wtime();
#pragma omp parallel num_threads(thread_count)
  {
#pragma omp critical
    global_result += Trap(a, b, n);
  }
  end = omp_get_wtime();
```

```c
double Trap(double a, double b, int n) {
  double h, x, my_result;
  double local_a, local_b;
  int i, local_n;
  int my_rank = omp_get_thread_num();
  int thread_count = omp_get_num_threads();

  h = (b - a) / n;
  local_n = n / thread_count;
  local_a = a + my_rank * local_n * h;
  local_b = local_a + local_n * h;
  my_result = (f(local_a) + f(local_b)) / 2.0;
  for (i = 1; i <= local_n - 1; i++) {
    x = local_a + i * h;
    my_result += f(x);
  }
  return my_result * h;
}
```

N = 999999990



O tempo de execução com 2 threads foi pior que o com 1 thread, isso por que a omp_critical faz a função Trap ser acessada de forma sequencial e ainda temos o overhead criado pelo omp_critical.

(b) Baixe o arquivo omp_trap_2b.c do site do livro. Como seu desempenho se compara? Explique suas respostas.

```
(base) felipehidequel@la-maquina:~/Desktop/pcd/5chapter$ ./q21b 1
Enter a, b, and n
0 1 999999990
With n = 999999990 trapezoids, our estimate
of the integral from 0.000000 to 1.000000 = 3.33333333333394e-01
elapsed time: 2.976975
(base) felipehidequel@la-maquina:~/Desktop/pcd/5chapter$ ./q21b 2
Enter a, b, and n
0 1 999999990
With n = 999999990 trapezoids, our estimate
of the integral from 0.000000 to 1.000000 = 3.33333333333429e-01
elapsed time: 1.862747
(base) felipehidequel@la-maquina:~/Desktop/pcd/5chapter$
```

O desempenho melhora com 2 threads, isso porquê, nessa versão de código foi substituido o uso da omp_critical que forçava a chamada de local_trap() de forma sequencial por uma redução global utilizando a clausula reduction:

```
double start, end;
  start = omp_get_wtime();
#pragma omp parallel num_threads(thread_count) reduction(+ : global_result)
  { global_result += Local_trap(a, b, n); }
  end = omp_get_wtime();
```

22. Suponha que no incrível computador Bleeblon, variáveis com tipo float possam arma zenar 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 );
```

Iteração 1 (i=0):Soma sum = 0.0 e a[0] = 4.0

Time	Operation	Operad 1	Operad 2	Result
1	Fetch operands	0.000×10^0	4.000×10^0	
2	Compare expon	0.000×10^0	4.000×10^0	
3	Shift	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

Iteração 2 (i=1):Soma sum = 4.0 e a[1]=3.0

Time	Operation	Operad 1	Operad 2	Result
1	Fetch operands	4.000×10^0	3.000×10^0	
2	Compare expon	4.000×10^0	3.000×10^0	
3	Shift	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

Iteração 3 (i=2):Soma sum = 7.0 e a[2]=3.0

Time	Operation	Operad 1	Operad 2	Result
1	Fetch operands	7.000×10^0	3.000×10^0	
2	Compare expon	7.000×10^0	3.000×10^0	
3	Shift	7.000×10^0	3.000×10^0	

4	**Add**	**$7.000x10^{0}$**	**$3.000x10^{0}$**	**$1.000x10^{1}$**
5	**Normalize Result**	**$7.000x10^{0}$**	**$3.000x10^{0}$**	**$1.000x10^{1}$**
6	**Round result**	**$7.000x10^{0}$**	**$3.000x10^{0}$**	**$1.000x10^{1}$**
7	**Store result**	**$7.000x10^{0}$**	**$3.000x10^{0}$**	**$1.00x10^{1}$**

Iteração 4 (i=3):Soma sum = 10.0 e a[3]=1000.0

Time	Operation	Operad 1	Operad 2	Result
1	**Fetch operands**	**$1.000x10^{1}$**	**$1.000x10^{3}$**	
2	**Compare expon**	**$1.000x10^{1}$**	**$1.000x10^{3}$**	
3	**Shift**	**$0.010x10^{3}$**	**$1.000x10^{3}$**	
4	**Add**	**$0.010x10^{3}$**	**$1.000x10^{3}$**	**$1.010x10^{3}$**
5	**Normalize Result**	**$0.010x10^{3}$**	**$1.000x10^{3}$**	**$1.010x10^{3}$**
6	**Round result**	**$0.010x10^{3}$**	**$1.000x10^{3}$**	**$1.01x10^{3}$**
7	**Store result**	**$0.010x10^{3}$**	**$1.000x10^{3}$**	**$1.01x10^{3}$**

A saída da ultima soma será 1010.

(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.

Thread 0:

Time	Operation	Operad 1	Operad 2	Result
1	**Fetch operands**	**$4.000x10^{0}$**	**$3.000x10^{0}$**	
2	**Compare expon**	**$4.000x10^{0}$**	**$3.000x10^{0}$**	
3	**Shift**	**$4.000x10^{0}$**	**$3.000x10^{0}$**	
4	**Add**	**$4.000x10^{0}$**	**$3.000x10^{0}$**	**$7.000x10^{0}$**

5	**Normalize Result**	**$4.000x10^{0}$**	**$3.000x10^{0}$**	**$7.000x10^{0}$**
6	**Round result**	**$4.000x10^{0}$**	**$3.000x10^{0}$**	**$7.000x10^{0}$**
7	**Store result**	**$4.000x10^{0}$**	**$3.000x10^{0}$**	**$7.000x10^{0}$**

Thread 1:

Time	Operation	Operad 1	Operad 2	Result
1	**Fetch operands**	**$3.000x10^{0}$**	**$1.000x10^{3}$**	
2	**Compare expon**	**$3.000x10^{0}$**	**$1.000x10^{3}$**	
3	**Shift**	**$0.003x10^{3}$**	**$1.000x10^{3}$**	
4	**Add**	**$0.003x10^{3}$**	**$1.000x10^{3}$**	**$1.003x10^{3}$**
5	**Normalize Result**	**$0.003x10^{3}$**	**$1.000x10^{3}$**	**$1.003x10^{3}$**
6	**Round result**	**$0.003x10^{3}$**	**$1.000x10^{3}$**	**$1.00x10^{3}$**
7	**Store result**	**$0.003x10^{3}$**	**$1.000x10^{3}$**	**$1.00x10^{3}$**

Redução

Time	Operation	Operad 1	Operad 2	Result
1	**Fetch operands**	**$7.000x10^{0}$**	**$1.000x10^{3}$**	
2	**Compare expon**	**$7.000x10^{0}$**	**$1.000x10^{3}$**	
3	**Shift**	**$0.007x10^{3}$**	**$1.000x10^{3}$**	
4	**Add**	**$0.007x10^{3}$**	**$1.000x10^{3}$**	**$1.007x10^{3}$**
5	**Normalize Result**	**$0.007x10^{3}$**	**$1.000x10^{3}$**	**$1.007x10^{3}$**
6	**Round result**	**$0.007x10^{3}$**	**$1.000x10^{3}$**	**$1.00x10^{3}$**
7	**Store result**	**$0.007x10^{3}$**	**$1.000x10^{3}$**	**$1.00x10^{3}$**

A saida vai ser 1000, pois como demonstrado nas tabelas, antes de salvar ele arredonda pra 3 digitos tornando o resultado da soma global errado.

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?

Código:

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

int main(int argc, char *argv[]) {
 int n, thread_count, i;
 int rank;
 int *iteractions;
 if (argc < 3) {
   printf("Usage: \n");
   printf("./q23 <Number of iteractions> <thread count>\n");
   return 1;
 }

 n = strtol(argv[1], NULL, 10);
 thread_count = strtol(argv[2], NULL, 10);
 iteractions = (int *)malloc(sizeof(int) * n);

#pragma omp parallel num_threads(thread_count) default(none) private(          \
   i,rank) shared(n,iteractions)
 {

#pragma omp for
   for (i = 0; i < n; i++) {
     rank = omp_get_thread_num();
     iteractions[i] = rank;
   }
 }

 for (int i = 0; i < n; i++) {
   printf("Iteração %d -- thread %d\n", i, iteractions[i]);
 }
 free(iteractions);
```

```
 return 0;
}
```

Nesse exemplo vamos rodar o laço com 16 iterações e 2 threads:

```
(base) felipehidequel@la-maquina:~/Desktop/pcd/ListaOpenMP/codigos$ ./q23.o 16 2
Iteração 0 -- thread 0
Iteração 1 -- thread 0
Iteração 2 -- thread 0
Iteração 3 -- thread 0
Iteração 4 -- thread 0
Iteração 5 -- thread 0
Iteração 6 -- thread 0
Iteração 7 -- thread 0
Iteração 8 -- thread 1
Iteração 9 -- thread 1
Iteração 10 -- thread 1
Iteração 11 -- thread 1
Iteração 12 -- thread 1
Iteração 13 -- thread 1
Iteração 14 -- thread 1
Iteração 15 -- thread 1
```

Cada núcleo faz N/P iterações, nesse caso, cada thread fez 8 iterações. O tipo de escalonamento que tem o chunksize por padrão n/p é o static, por isso esse escalonador é o static.

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.

A operação a[i] = a[i-1] +i; soma ao indice i o valor do indice anterior com i, sabendo disso, podemos adaptar essa formula para a formula de somatorio dos naturais:

a[i] = (i*(i+1))/2;

dessa forma mantemos a logíca da operação e eliminamos a dependencia, já que agora não precisamos saber o valor de a[i-1] para estimar a[i].

código:

```
void serial_func(int n) {
  int a[n];

  a[0] = 0;
  for (int i = 0; i < n; i++) {
    a[i] = a[i - 1] + i;
  }

  printf("Serial:\n");
  for (int i = 0; i < n; i++) {
    printf("%d\n", a[i]);
  }
}

void parallel_func(int n, int thread_count) {
  int a[n];

  a[0] = 0;
#pragma omp parallel num_threads(thread_count) default(none) shared(a, n)
  {
#pragma omp for
    for (int i = 0; i < n; i++) {
      a[i] = (i*(i+1))/2;
    }
  }
printf("Parallel:\n");
 for (int i = 0; i < n; i++) {
   printf("%d\n", a[i]);
 }
}
```

Para fins de comparação, obtemos a seguinte saída:

```
felipehidequel@la-maquina:~/Desktop/pcd/5chapter$ ./q24 8 4
Serial:
0
1
3
6
10
15
21
28
Parallel:
0
1
3
6
10
15
21
28
felipehidequel@la-maquina:~/Desktop/pcd/5chapter$
```

8 é a quantidade de iterações e 4, o número de threads.

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). Exe cute 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?

Modificação:

```
double Trap(double a, double b, int n, int thread_count) {
  double h, approx;
  int i;
  int *iterations = (int *)malloc(n * sizeof(int));

  h = (b - a) / n;
  approx = (f(a) + f(b)) / 2.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
  for (i = 1; i <= n - 1; i++) {

    iterations[i] = omp_get_thread_num();
```

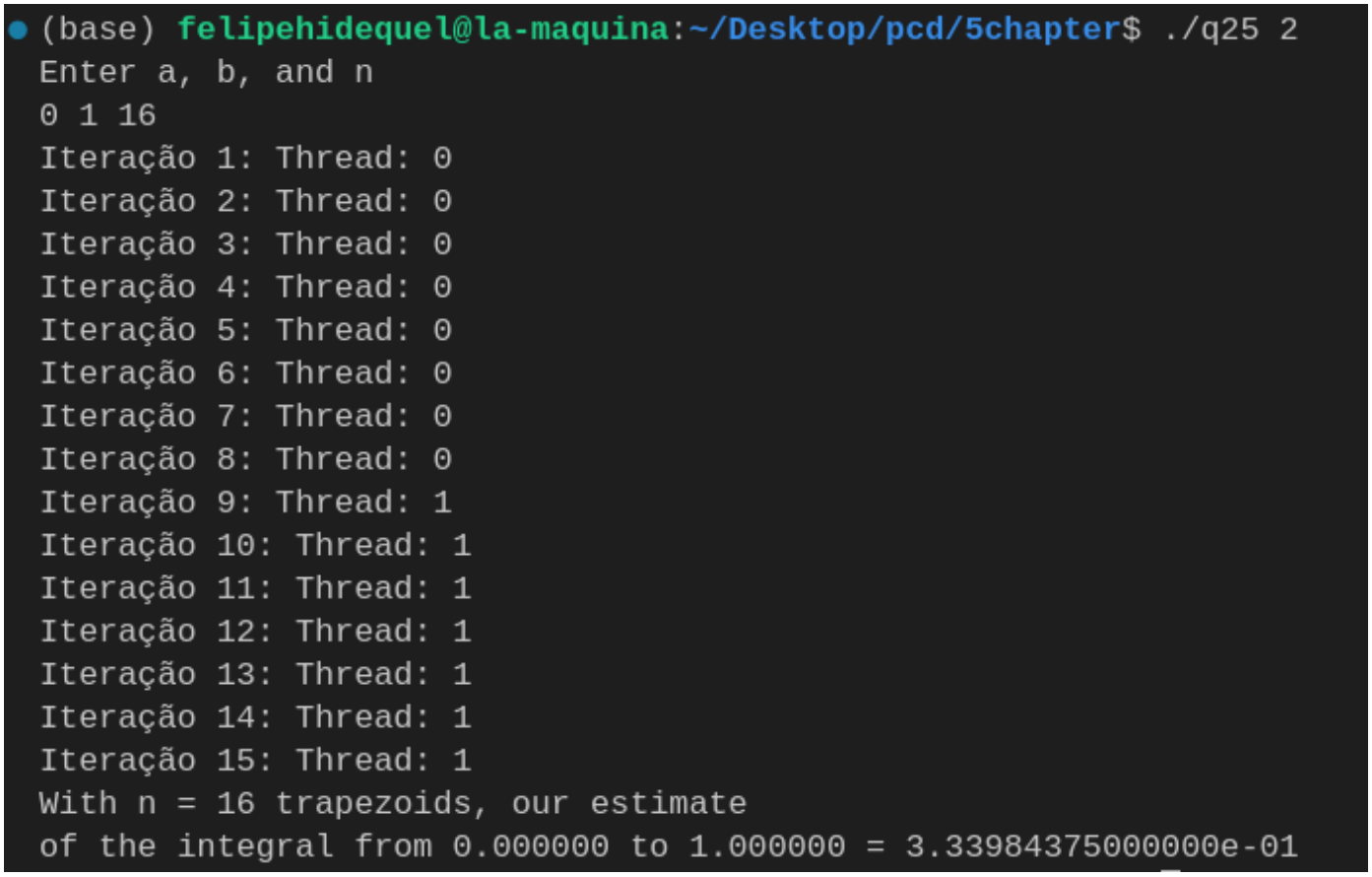
```
    approx += f(a + i * h);
  }
  approx = h * approx;
    for (i = 1; i <= n-1; i++) {
        printf("Iteração %d: Thread: %d\n", i, iterations[i]);
    }

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

Sem schedule :

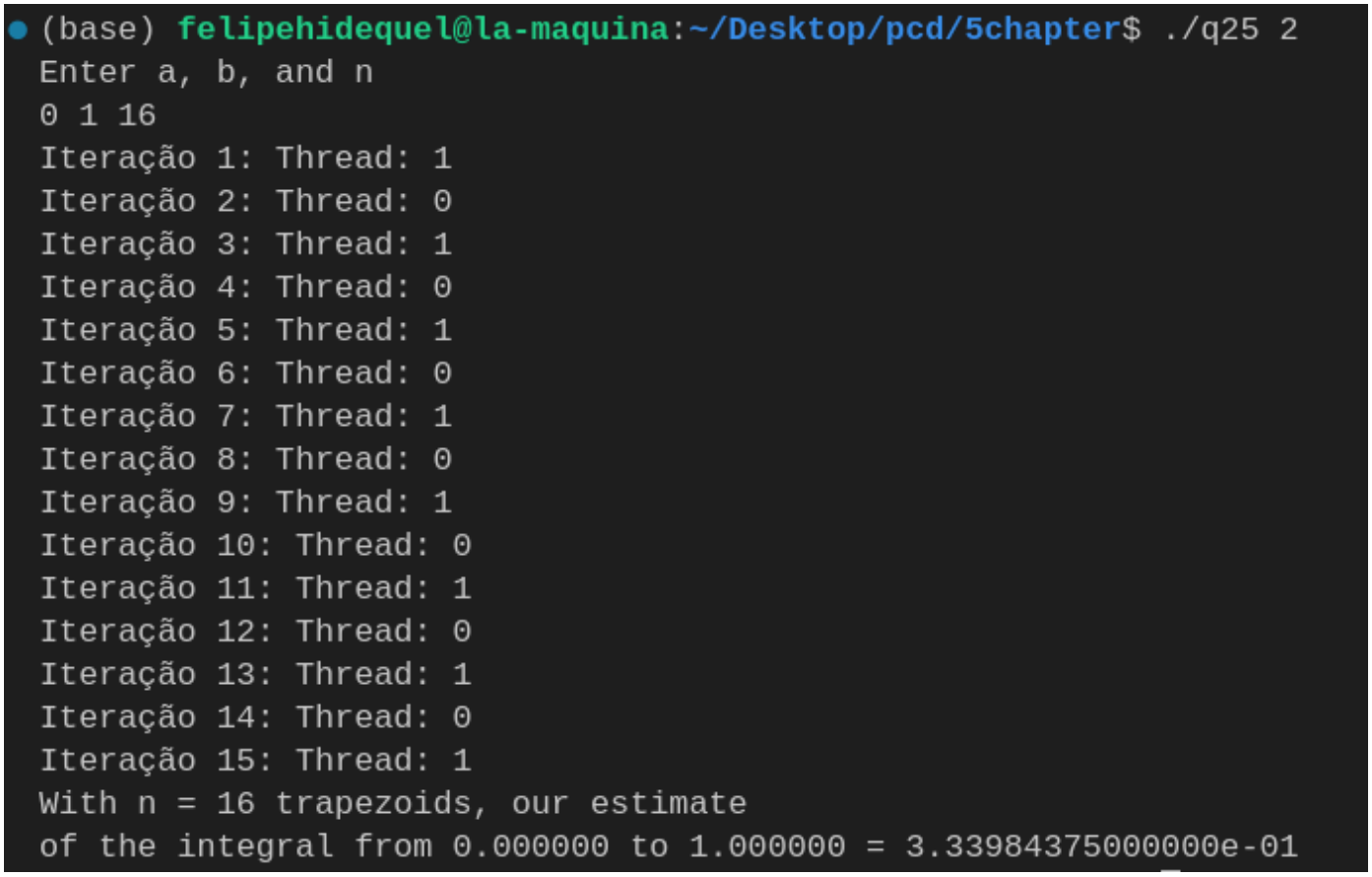
```
(base) felipehidequel@la-maquina:~/Desktop/pcd/5chapter$ ./q25 2
Enter a, b, and n
0 1 16
Iteração 1: Thread: 0
Iteração 2: Thread: 0
Iteração 3: Thread: 0
Iteração 4: Thread: 0
Iteração 5: Thread: 0
Iteração 6: Thread: 0
Iteração 7: Thread: 0
Iteração 8: Thread: 0
Iteração 9: Thread: 1
Iteração 10: Thread: 1
Iteração 11: Thread: 1
Iteração 12: Thread: 1
Iteração 13: Thread: 1
Iteração 14: Thread: 1
Iteração 15: Thread: 1
With n = 16 trapezoids, our estimate
of the integral from 0.000000 to 1.000000 = 3.339843750000000e-01
```

Temos uma distribuição estática com tamanho de bloco n/p, o que me leva a dizer que por padrão o sistema está utilizando o escalonador static com chunk default;

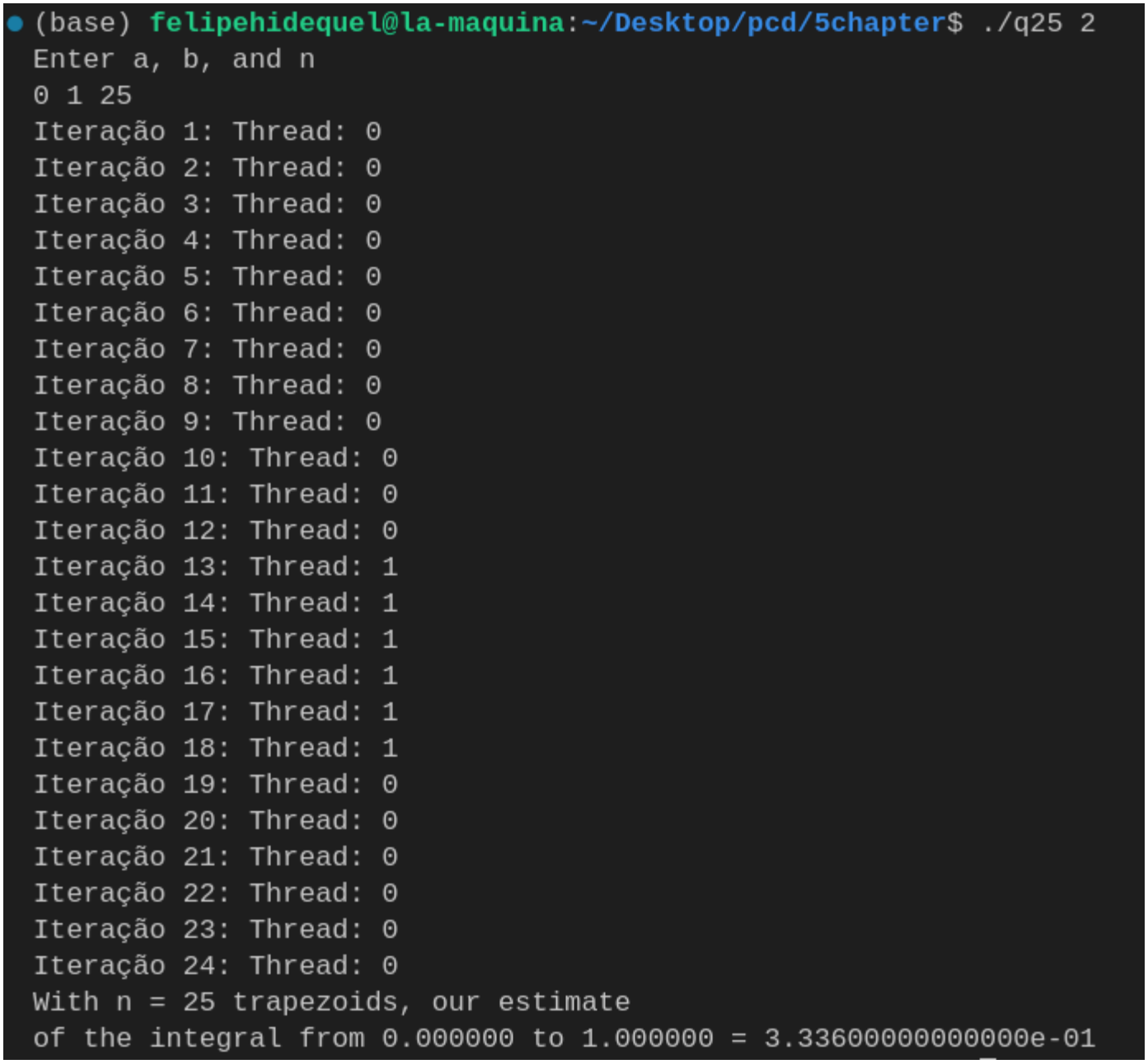
export OMP_SCHEDULE=static:



export OMP_SCHEDULE=dynamic



export OMP_SCHEDULE=guided



O Guided faz uma distribuição sob demanda variando o tamanho do chunk, no inicio ele deu um bloco grande de iterações para a thread 0, e para thread 1 deu um bloco menor e então ele vai diminuindo o tamanho do bloco a cada distribução até chegar no tamanho de chunk padrão que é 1.

export OMP_SCHEDULE=auto

```
(base) felipehidequel@la-maquina:~/Desktop/pcd/5chapter$ ./q25 2
Enter a, b, and n
0 1 16
Iteração 1: Thread: 0
Iteração 2: Thread: 0
Iteração 3: Thread: 0
Iteração 4: Thread: 0
Iteração 5: Thread: 0
Iteração 6: Thread: 0
Iteração 7: Thread: 0
Iteração 8: Thread: 0
Iteração 9: Thread: 1
Iteração 10: Thread: 1
Iteração 11: Thread: 1
Iteração 12: Thread: 1
Iteração 13: Thread: 1
Iteração 14: Thread: 1
Iteração 15: Thread: 1
With n = 16 trapezoids, our estimate
of the integral from 0.000000 to 1.000000 = 3.339843750000000e-01
```

Para auto ele aparenta está utilizando o escalonador static, já que a sáida foi muito parecida e cada thread fez um bloco de iterações consecutivas com tamanho n/p.

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?

(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?

O falso compartilhamento ocorre quanto 2 ou mais threads tem acesso a uma mesma cacheline. Se a distribuição a distribuição não for ciclica nós teriamos o seguinte cénario:

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]

Cada linha de cache vai ter 8 elementos de y consecutivos.
Como cada thread processa uma sequência contínua de índices, os elementos atribuídos a diferentes threads ficam em regiões de memória separadas.

Agora no cenário em temos uma distribuição ciclica:

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[...]

Cada linha de cache vai ter 8 elementos de y consecutivos. Nesse exemplo com o tamanho de bloco igual a 1, todas as threads compartilham a mesma linha de cache, atestando um cenário de falso compartilhamento.

Resposta final:
Distribuição estática: Não ocorre falso compartilhamento em nenhum dos casos.

Distribuição ciclica: Ocorre falso compartilhamento em ambos os casos

3.1 Questões extra

31. Suponha que lançamos dardos aleatoriamente em um alvo quadrado. Vamos considerar o centro desse alvo como sendo a origem de um plano cartesiano e os lados do alvo medem 2 pés de comprimento. Suponha também que haja um círculo inscrito no alvo. O raio do círculo é 1 pé e sua área é π pés quadrados. Se os pontos atingidos pelos dardos estiverem distribuídos uniformemente (e sempre acertamos o alvo), então o número de dardos atingidos dentro do círculo deve satisfazer aproximadamente a equação

$$\frac{qtd\_no\_circulo}{num\_lancamentos} = \frac{\pi}{4}$$

já que a razão entre a área do círculo e a área do quadrado é $\frac{\pi}{4}$.
Podemos usar esta fórmula para estimar o valor de π com um gerador de números aleatórios:


```
qtd_no_circulo = 0;
for (lancamento = 0; lancamento < num_lancamentos; lancamento++) {
x = double aleatório entre -1 e 1;
y = double aleatório entre -1 e 1;
distancia_quadrada = x * x + y * y;
if (distancia_quadrada <= 1) qtd_no_circulo++;
}
estimativa_de_pi = 4 * qtd_no_circulo/((double) num_lancamentos);
```

Isso é chamado de método "Monte Carlo", pois utiliza aleatoriedade (o lançamento do dardo).

Escreva um programa OpenMP que use um método de Monte Carlo para estimar π. Leia o número total de lançamentos antes de criar as *threads*. Use uma cláusula de reduction para encontrar o número total de dardos que atingem o círculo. Imprima o resultado após encerrar a região paralela. Você deve usar long long ints para o número de acertos no círculo e o número de lançamentos, já que ambos podem ter que ser muito grandes para obter uma estimativa razoável de π.

Código:

```
double pi_monte_carlo_parallel(long long int lancamentos, int thread_count);

int main(int argc, char *argv[]) {
  if (argc < 3) {
    printf(
        "Usage: ./monte_carlo <número de lançamentos> <número de threads>\n");
    exit(1);
  }
  double pi;
  int n, thread_count;

  n = strtol(argv[1], NULL, 10);
  thread_count = strtol(argv[2], NULL, 10);

  pi = 0.0;
  pi = pi_monte_carlo_parallel(n, thread_count);
  printf("Pi: %f\n", pi);

  return 0;
}

double pi_monte_carlo_parallel(long long int lancamentos, int thread_count) {
  long long int qtd_no_circulo = 0;
  double x, y, distancia_quadrada;
#pragma omp parallel default(none)                \
shared(lancamentos, qtd_no_circulo) private(x, y, distancia_quadrada)
  {
    unsigned seed = omp_get_thread_num() * time(NULL);
#pragma omp for reduction(+ : qtd_no_circulo)
    for (int lancamento = 0; lancamento < lancamentos; lancamento++) {
      x = (double)rand_r(&rank) / RAND_MAX * 2 - 1;
      y = (double)rand_r(&rank) / RAND_MAX * 2 - 1;
```

```
      distancia_quadrada = x * x + y * y;
      if (distancia_quadrada <= 1)
        qtd_no_circulo++;
    }
  }

  return 4 * qtd_no_circulo / ((double)lancamentos);
}
```

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 ele mentos 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: a,n,temp
privadas: i, j, count

(b) Se paralelizarmos o laço for i usando o escopo especificado na parte anterior,

haverá alguma dependência de dados no laço? Explique sua resposta.

Não haverá dependencias por que cada thread estará trabalhando com um pedacinho de a, fazendo suas contagens individualmente.

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

Podemos utilizando uma distribuição estática:

```
#pragma omp barrier
    int rest = n % thread_count;
    int my_rank = omp_get_thread_num();
    int my_chunk = n / thread_count + (my_rank < rest);
    int my_start = my_rank * my_chunk +
                   (my_rank < rest ? my_rank : rest);

    memcpy(&a[my_start], &temp[my_start], my_chunk * sizeof(int));
  }
```

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

```
int main(int argc, char *argv[]) {
  if (argc < 3) {
    printf("Usage: \n");
    printf("./32 <Number of elements> <number of threads>\n");
    return 1;
  }

  int n = strtol(argv[1], NULL, 10);
  int thread_count = strtol(argv[2], NULL, 10);

  int *b = init_vect(n);

  Count_sort_parallel(b, n, thread_count);
  free(b);

  return 0;
}

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);
}

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(a, n, temp, thread_count) private(i, j, count)
  {
#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 rank = omp_get_thread_num();
    int chunk = n / thread_count + (my_rank < rest);
    int start = rank * chunk + (rank < rest ? rank : rest);

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

  free(temp);
}
```

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

Trecho de código:

```
start = omp_get_wtime();
Count_sort(a, n);
end = omp_get_wtime();
printf("Serial time: %f\n", end - start);

start = omp_get_wtime();
Count_sort_parallel(b, n, thread_count);
end = omp_get_wtime();
printf("Parallel time: %f\n", end - start);

start = omp_get_wtime();
qsort(c, n, sizeof(int), compare);
end = omp_get_wtime();
printf("Qsort time: %f\n", end - start);
```

Saida:

```
(base) felipehidequel@la-maquina:~/Desktop/pcd/5chapter$ ./q32 100000 8
Serial time: 44.347219
Parallel time: 9.376984
Qsort time: 0.007772
```

Em tempo de execução, a nossa proposta de countsort paralelo é aproximadamente 4 vezes mais rápida que a função serial, mesmo assim ainda é muito mais lenta que o qsort isso por a complexidade do qsort ser muito menor que a de nosso algoritmo.