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

QUESTÕES QUE VALEM A SEGUNDA UNIDADE
Lavínia Dantas de Mesquita
Questões que respondi: 20, 21, 22, 23, 24, 25, 26, 28, 29, 31*

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?

Usando 6 threads e 6,12,24 trapézios e o resultado já estava diferente do que o que foi dado com a área crítica. Entretanto, quanto maior o número de trapézios, maior a chance do resultado ser correto (60 foi o valor mínimo para que isso acontecesse).

Usando 2 threads e 2,4,6,8,10 trapézios, o resultado estava correto até executar com 8 trapézios, no caso de poucas threads, quanto menos trapézios, melhor a máquina se saiu.

Independente da quantidade de threads e trapézios, o resultado pode sair errado graças ao indeterminismo, entretanto, com menos threads e mais trapézios o resultado tem maiores chances de ser correto graças a diminuição da condição de corrida.

*A diretiva **#pragma omp critical** é usada para garantir que apenas uma thread atualize o resultado global de cada vez, garantindo que não haja perda de informações.*

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

O aumento do número de trapézios influencia positivamente nas chances do resultado ser mais preciso, mas ainda existem chances de erros devido por não estarmos usando a região crítica. O aumento do número de trapézios faz com que a chances de acesso mútuo entre threads a variável global seja menor, favorecendo o resultado final, visto que a perda de informação se torna menor. O que evita a condição de corrida

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

O aumento do número de threads influencia as chances do resultado ser incorreto principalmente devido à necessidade de sincronização e à distribuição equilibrada do trabalho entre os threads.
O aumento de threads aumenta a chance de acesso mútuo à variável global de soma, o que leva à perda de informações graças à condição de corrida.

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 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?

Após alguns testes podemos observar que quando o programa executa com duas ou mais threads o programa demora mais a terminar o processo, devido ao fato de que elas tem que esperar as outras terminarem para conseguir acessar a região crítica e evitar a condição de corrida.

```
Tempo passado: 0.000000e+000
PS C:\Users\lavin\Área de Trabalho\ch5> ./omp_trap1 1
Enter a, b, and n
0
9
200000000
With n = 200000000 trapezoids, our estimate
of the integral from 0.000000 to 9.000000 = 2.42999999999934e+002
Tempo passado: 5.991000e+000
PS C:\Users\lavin\Área de Trabalho\ch5> ./omp_trap1 2
Enter a, b, and n
0
9
200000000
With n = 200000000 trapezoids, our estimate
of the integral from 0.000000 to 9.000000 = 2.42999999999987e+002
Tempo passado: 6.212000e+000
PS C:\Users\lavin\Área de Trabalho\ch5>
```

(b) Baixe o arquivo omp_trap_2.c do site do livro. Como seu desempenho se compara? Explique suas respostas.

```
PS C:\Users\lavin\Área de Trabalho\ch5> gcc -g -Wall -fopenmp -o omp_
PS C:\Users\lavin\Área de Trabalho\ch5> ./omp_trap2a 1
Enter a, b, and n
0
9
200000000
With n = 200000000 trapezoids, our estimate
of the integral from 0.000000 to 9.000000 = 2.42999999999934e+002
PS C:\Users\lavin\Área de Trabalho\ch5> ./omp_trap2a 2
Enter a, b, and n
0
9
200000000
With n = 200000000 trapezoids, our estimate
of the integral from 0.000000 to 9.000000 = 2.42999999999987e+002
PS C:\Users\lavin\Área de Trabalho\ch5>
```

O segundo programa se sai melhor pois não há o uso de uma região crítica, ou seja, nada de esperar para executar a ação de soma, e sim, apenas somarem seus valores na variável global.

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?

```
8
int i ;
float sum = 0.0;
for (i = 0; i < 4; i++)
sum += a[ i];
printf ("sum = %4.1f\n", sum );
```

Os pontos flutuantes são armazenados no computador de uma forma similar a uma notação científica. O array, depois que eu converti com ajuda de uma calculadora online pois não lembro como se faz na mão, pode ser visto como: a[] = {2.00e+00, 2.00e+00, 4.00e+00, 1.00e+03}.
Quando os valores forem adicionados usando o primeiro exemplo, a soma é:

Depois que i = 0: sum = 2.00e+00
Depois que i = 1: sum = 4.00e+00

Depois que i = 2: sum = 8.00e+00

Quando i = 3, o valor correspondente vai ser guardado por um tempinho, e quando for para a memória ele vai ser arredondado para 1.01e+00, somando tudo, a soma vai sair como 1010.0 visto que é/,3 sequencial.

(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?

Quando os valores são somados em paralelo, vai existir uma variável privada para cada thread e vai ser usada para que cada uma armazene sua soma parcial. Quando elas completarem sua iteração, a thread 0 vai ter 4.00e+00, a thread 1 vai ter 1.00e+03 já que 1.004e+03 vai ser arredondado.

Quando ambas forem pro registrador, ele vai ficar como 1.004e+03 e quando forem pra memória principal vão ficar como 1.00e+03 na soma, que dá 1000.0.

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?

```
Escalonamento: 2
Chunksize: 1

Thread          Iteracoes
------          ----------
   1            0 -- 52
   2            53 -- 60
   1            61 -- 73
   2            74 -- 82
   1            83 -- 85
   2            86 -- 88
   1            89 -- 90
   2            91 -- 92
   1            93 -- 94
   2            95 -- 95
   1            96 -- 96
   2            97 -- 97
   1            98 -- 98
   2            99 -- 99
   0            100 -- 100
PS C:\Users\lavin\Área de Trabalho\ch5> ./questao23 6 100

Escalonamento: 2
Chunksize: 1

Thread          Iteracoes
------          ----------
   2            0 -- 99
   0            100 -- 100
```

```
PS C:\Users\lavin\Área de Trabalho\ch5> ./questao23 6 100

Escalonamento: 2
Chunksize: 1

Thread          Iteracoes
------          ----------
   2            0 -- 99
   0            100 -- 100
PS C:\Users\lavin\Área de Trabalho\ch5> ./questao23 6 100

Escalonamento: 2
Chunksize: 1

Thread          Iteracoes
------          ----------
   0            0 -- 16
   2            17 -- 52
   0            53 -- 60
   2            61 -- 67
   0            68 -- 73
   2            74 -- 85
   0            86 -- 88
   2            89 -- 90
   0            91 -- 92
   2            93 -- 94
   0            95 -- 95
   2            96 -- 96
   0            97 -- 97
   2            98 -- 98
   0            99 -- 100
```

No meu caso, o escalonamento é dinâmico, nem todas as minhas threads trabalham, e o trabalho é dividido conforme a disponibilidade da thread em questão.

Essa é uma estratégia que busca otimizar o uso de recursos em ambientes de computação paralela e sistemas distribuídos, permitindo que o sistema operacional ou o ambiente de execução ajuste

automaticamente a alocação de threads para núcleos de processamento, com base em critérios como a carga atual do sistema, a natureza das tarefas em execução e as características específicas do hardware.

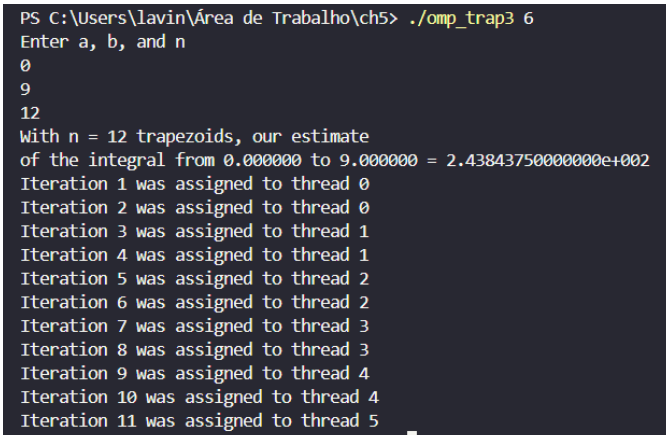
24. Considere o seguinte laço:

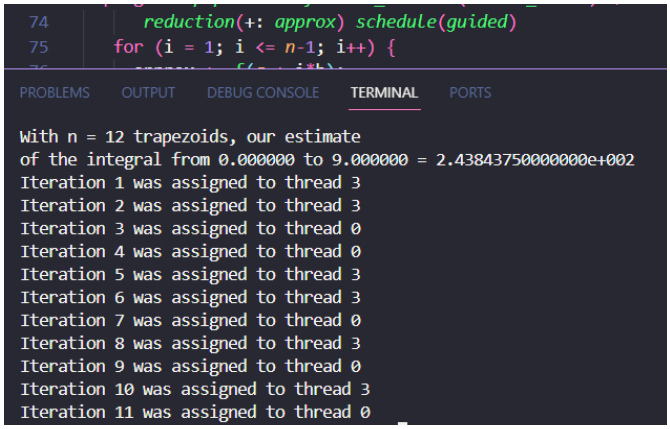
```
a[0] = 0;
for ( i = 1; i < n ; i++)
        a[i] = a[i-1] + i;
```

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.**

As somas parciais podem ser calculadas previamente, dentro de outro vetor, para que não exista mais a dependência do termo anterior para calcular a próxima soma. Mas o programa deixa de ser paralelo.
Outra solução é usar a soma de Gauss, onde a soma dos termos de um conjunto de valores pode ser calculada pela soma de dois termos equidistantes, multiplicada pelo número de pares da sequência de valores, ou seja, por n/2: i(i+1)/2*
Desse jeito tira a dependência e mantém o paralelismo.

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 iterações de n int's e, na função Trap, atribuindo omp_get_thread_num() a iterações[i] na i-ésima iteração do laço for. **Qual é o escalonamento padrão de iterações em seu sistema? Como o escalonamento guided é determinado?**





Usando schedule(runtime), eu uso o padrão da OMP_SCHEDULE da minha máquina, como podemos observar, é estático, as minhas threads dividiram o trabalho das iterações propostas de forma nivelada.
O escalonamento guided foca em distribuir as iterações de um loop entre os threads de forma que o tamanho dos pedaços de iterações diminua progressivamente. Isso é feito para garantir um equilíbrio de carga entre os threads, útil em situações onde o tempo de execução varia significativamente entre cada iteração.

26. Lembre-se de que todos os blocos estruturados modificados por uma diretiva critical formam uma única secção crítica. O que acontece se tivermos um número de diretivas 9 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?

Quando usamos diretivas atômicas, a implementação OpenMP permite que exista a execução simultânea de atualizações para variáveis distintas, ou seja, diferentes seções podem ser criadas independentemente e executam as ações sem a necessidade de esperar a finalização das outras, o que permite que não haja condição de corrida além de criar mais de uma zona crítica. Eu estou usando 6 threads antes de compilar, 8 na primeira compilação e 12 na segunda compilação.

```
PS C:\Users\lavin\Área de Trabalho\Códigos Segunda Unidade PCD> ./questao26
Single-thread: 0.156000
Multi-thread: 0.029000
PS C:\Users\lavin\Área de Trabalho\Códigos Segunda Unidade PCD> gcc -o questao26 questao26.c -fopenmp
PS C:\Users\lavin\Área de Trabalho\Códigos Segunda Unidade PCD> ./questao26
Single-thread: 0.150000
Multi-thread: 0.019000
PS C:\Users\lavin\Área de Trabalho\Códigos Segunda Unidade PCD> gcc -o questao26 questao26.c -fopenmp
PS C:\Users\lavin\Área de Trabalho\Códigos Segunda Unidade PCD> ./questao26
Single-thread: 0.158000
Multi-thread: 0.028000
PS C:\Users\lavin\Área de Trabalho\Códigos Segunda Unidade PCD> ./questao26
Single-thread: 0.155000
Multi-thread: 0.029000
PS C:\Users\lavin\Área de Trabalho\Códigos Segunda Unidade PCD> ./questao26
Single-thread: 0.152000
Multi-thread: 0.021000
```

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?
Os elementos vão ser divididos assim, mais ou menos:

Thread 0: y[0], … , y[1999]
Thread 1: y[2000], … , y[3999]
Thread 2: y[4000], … , y[5999]
Thread 3: y[6000], … , y[7999]

Para ocorrer um falso compartilhamento, tem que ocorrer o caso de que algum elemento do y pertença a threads diferentes. Na thread 0, a linha que está mais próxima dos elementos da 2 é a linha que tem y[1999]. A linha que contém p y[1999] é essa aqui:

y[1999] y[2000] y[2001] y[2002] y[2003] y[2004] y[2005] y[2006]

Vendo que o menor elemento da thread 2 é 4000, é impossível que esses elementos pertençam a 0 e 2 ao mesmo tempo.
No segundo caso é o mesmo conceito, considerando os valores e a quantidade de informações em uma linha, é impossível que aconteça falso compartilhamento graças a atribuição de memória específica.

29. Embora strtok_r seja thread-safe, ele tem a propriedade bastante infeliz de modificar a string de entrada. Escreva um método para gerar tokens que seja thread-safe e não modifique a string de entrada.
A ideia do meu código era para que cada thread alocasse sua própria cópia da linha, processar a cópia e liberar a memória alocada para a cópia dentro do looping.
Para garantir que a string original não fosse modificada, não havendo vazamento de memória. Entretanto, não consegui implementar pois strtok_r não existe no mingw, portanto não consigo utilizar. Portanto, eu fiz a questão de forma alternativa, criando uma função que replica o comportamento da strtok_r.

QUESTÕES EXTRAS

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 abaixo, já que a razão entre a área do círculo e a área do quadrado é π/ 4.

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

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.

Função principal: A função main é o ponto de entrada do programa. Ela solicita ao usuário que insira o número total de lançamentos (iterações) que serão realizados para estimar Pi.

Variáveis de controle:

- *num_lancamentos: para armazenar o número de lançamentos inserido pelo usuário*
- *qtd_no_circulo: para contar quantos desses lançamentos caem dentro de um círculo de raio 1, centrado na origem.*

Cálculo de Pi: O cálculo de Pi é realizado em um loop paralelizado entre as threads, onde cada iteração gera um par de coordenadas aleatórias (x, y) dentro de um quadrado 2x2. Se a distância quadrada dessas coordenadas for menor ou igual a 1 (ou seja, o ponto cai dentro do círculo), a variável qtd_no_circulo é incrementada. A redução (reduction(+:qtd_no_circulo)) serve para garantir que todas as threads atualizem corretamente a variável qtd_no_circulo de forma segura, para não haver perda de informações.

Estimativa de Pi: O valor de Pi é estimado multiplicando o número de pontos que caíram dentro do círculo (qtd_no_circulo) por 4 e dividindo pelo número total de lançamentos (num_lancamentos), que é a reelaboração da fórmula mostrada na questão.

MEUS CÓDIGOS:

Para a questão 20, 21:

```
 * Compile: gcc -g -Wall -fopenmp -o omp_trap1 omp_trap1.c
 * Usage:    ./omp_trap1 <number of threads>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

void Usage(char* prog_name);
double f(double x);
double Local_trap(double a, double b, int n);

int main(int argc, char* argv[]) {
   double  a, b;
   int     n;
   int     thread_count;
   double start, end;

   if (argc != 2) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
   if (n % thread_count != 0) Usage(argv[0]);
   double global_result = 0.0;
```

```
   start = omp_get_wtime();
#  pragma omp parallel num_threads(thread_count)
   #pragma omp critical
   global_result += Local_trap(a, b, n);
   end = omp_get_wtime();
   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   printf("Tempo passado: %e", end - start);
   return 0;
}

void Usage(char* prog_name) {

   fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
   fprintf(stderr, "   number of trapezoids must be evenly divisible by\n");
   fprintf(stderr, "   number of threads\n");
   exit(0);
}  double f(double x) {
   double return_val;

   return_val = x*x;
   return return_val;
}
double Local_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);
   }
   my_result = my_result*h;
```

```
   return my_result;
}
```

Para a questão 21:

```
 * Compile: gcc -g -Wall -fopenmp -o omp_trap2a omp_trap2a.c -lm
 * Usage:   ./omp_trap2a <number of threads>
 *
 * Notes:
 *   1.  The function f(x) is hardwired.
 *   2.  This version assumes that n is evenly divisible by the
 *       number of threads
 * IPP:  Section 5.4 (p. 222)
 */

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

void Usage(char* prog_name);
double f(double x);
double Local_trap(double a, double b, int n);

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

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

   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;
```

```
      my_result += Local_trap(a, b, n);
#     pragma omp critical
      global_result += my_result;
   }

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}
void Usage(char* prog_name) {

   fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
   fprintf(stderr, "   number of trapezoids must be evenly divisible by\n");
   fprintf(stderr, "   number of threads\n");
   exit(0);
}
double f(double x) {
   double return_val;

   return_val = x*x;
   return return_val;
}

double Local_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);
   }
   my_result = my_result*h;
```

```
   return my_result;
}
```

Para a questão 25:

```
 * Compile: gcc -g -Wall -fopenmp -o omp_trap3 omp_trap3.c
 * Usage:    ./omp_trap3 <number of threads>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

void Usage(char* prog_name);
double f(double x);
double Trap(double a, double b, int n, int thread_count);

int thread_assignments[1000];

int main(int argc, char* argv[]) {
   double global_result = 0.0;
   double a, b;
   int     n;
   int     thread_count;

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

   global_result = Trap(a, b, n, thread_count);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);

   for (int i = 1; i < n; i++) {
                printf("Iteration %d was assigned to thread %d\n", i,
thread_assignments[i]);
   }
```

```
   return 0;
}

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

double f(double x) {
   double return_val;
   return_val = x*x;
   return return_val;
}

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

   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++) {
     approx += f(a + i*h);
     thread_assignments[i] = omp_get_thread_num();
   }
   approx = h*approx;

   return approx;
}
```

Para a questão 23:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
/*
compilar: gcc -g -Wall -fopenmp -o questao23 questao23.c
executar: ./questao23 <numero_de_threads> <tamanho_do_vetor>
```

```
 */

void Print_iters(int iterations[], long n);

int main(int argc, char *argv[])
{
    omp_sched_t sched;
    int num_threads, chunksize;
    int tamanho, i;
    num_threads = strtol(argv[1], NULL, 10);
    tamanho = strtol(argv[2], NULL, 10);
    int *iterations;
    iterations = (int *)malloc((tamanho + 1) * sizeof(int));
    omp_get_schedule(&sched, &chunksize);
#pragma omp parallel num_threads(num_threads)

#pragma omp for schedule(guided)
    for (i = 0; i < tamanho; i++)
    {
        iterations[i] = omp_get_thread_num();
    }

    printf("\nEscalonamento: %i \n", sched);
    printf("Chunksize: %i \n", chunksize);
    Print_iters(iterations, tamanho);
    free(iterations);
}

void Print_iters(int iterations[], long n)
{
    int i, start_iter, stop_iter, which_thread;

    printf("\n");
    printf("Thread\t\tIteracoes\n");
    printf("------\t\t----------\n");
    which_thread = iterations[0];
    start_iter = stop_iter = 0;
    for (i = 0; i <= n; i++)
        if (iterations[i] == which_thread)
            stop_iter = i;
        else
```

```
        {
            printf("%4d  \t\t%d -- %d\n", which_thread, start_iter,
                    stop_iter);
            which_thread = iterations[i];
            start_iter = stop_iter = i;
        }
    printf("%4d  \t\t%d -- %d\n", which_thread, start_iter,
            stop_iter);
}
```

Para a questão 26:

```
#include <stdio.h>
#include <math.h>
#include <omp.h>
//compilar: gcc -o questao26 questao26.c -fopenmp
//rodar ./questao26

int main() {
    int n = 1000000, minhasthreads = 12;
    double resultado1 = 0.0,resultado2 = 0.0, tempoi, tempof;

    //Teste com um thread
    tempoi = omp_get_wtime();
    #pragma omp parallel for num_threads(1) reduction(+:resultado1)
    for (int i = 0; i < n; i++) {resultado1 += sin(i);}
    tempof = omp_get_wtime();
    printf("Single-thread: %f \n", tempof - tempoi);

    //Teste com vários threads
    tempoi = omp_get_wtime();
    #pragma omp parallel for num_threads(minhasthreads) reduction(+:resultado2)
    for (int i = 0; i < n; i++) {resultado2 += sin(i);}
    tempof = omp_get_wtime();
    printf("Multi-thread: %f \n", tempof - tempoi);

    return 0;
}
```

Para a questão 29:

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

char *my_strtok_r(char *str, const char *delim, char **saveptr) {
    if (str == NULL) {
        str = *saveptr;
    }
    if (str == NULL) {
        return NULL;
    }

    char *token_start = str;
    char *token_end = strpbrk(token_start, delim);

    if (token_end == NULL) {
        *saveptr = NULL;
        return token_start;
    }

    *token_end = '\0';
    *saveptr = token_end + 1;

    return token_start;
}

int main() {
    char str[] = "Hello, World! How are you?";
    char *saveptr;
    char *token = my_strtok_r(str, " ", &saveptr);

    while (token != NULL) {
        printf("%s\n", token);
        token = my_strtok_r(NULL, " ", &saveptr);
    }

    return 0;
}
```

Para a questão 31:

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

int main() {
    long long int num_lancamentos;
    long long int qtd_no_circulo = 0;
    double estimativa_de_pi;
    printf("Digite o número total de lançamentos: ");
    scanf("%lld", &num_lancamentos);

    #pragma omp parallel for reduction(+:qtd_no_circulo)
        for (long long int lancamento = 0; lancamento < num_lancamentos;
lancamento++) {
      double x = (double)rand() / RAND_MAX * 2.0 - 1.0;
      double y = (double)rand() / RAND_MAX * 2.0 - 1.0;
      double distancia_quadrada = x * x + y * y;
        if (distancia_quadrada <= 1) qtd_no_circulo++;
    }
    estimativa_de_pi = 4.0 * qtd_no_circulo / (double)num_lancamentos;
    printf("Estimativa de Pi: %lf\n", estimativa_de_pi);

    return 0;
}
```