Programação de Arquiteturas com Memória Compartilhada Utilizando OpenMP

MCZA020-13 - Programação Paralela

Emilio Francesquini
e.francesquini@ufabc.edu.br

2025.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Programação Paralela na UFABC**.
- Estes slides são baseados naqueles produzidos por Peter Pacheco como parte do livro *An Introduction to Parallel Programming* disponíveis em: `https://www.cs.usfca.edu/~peter/ipp/`
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Algumas figuras foram obtidas em: `http://pngimg.com`

OpenMP

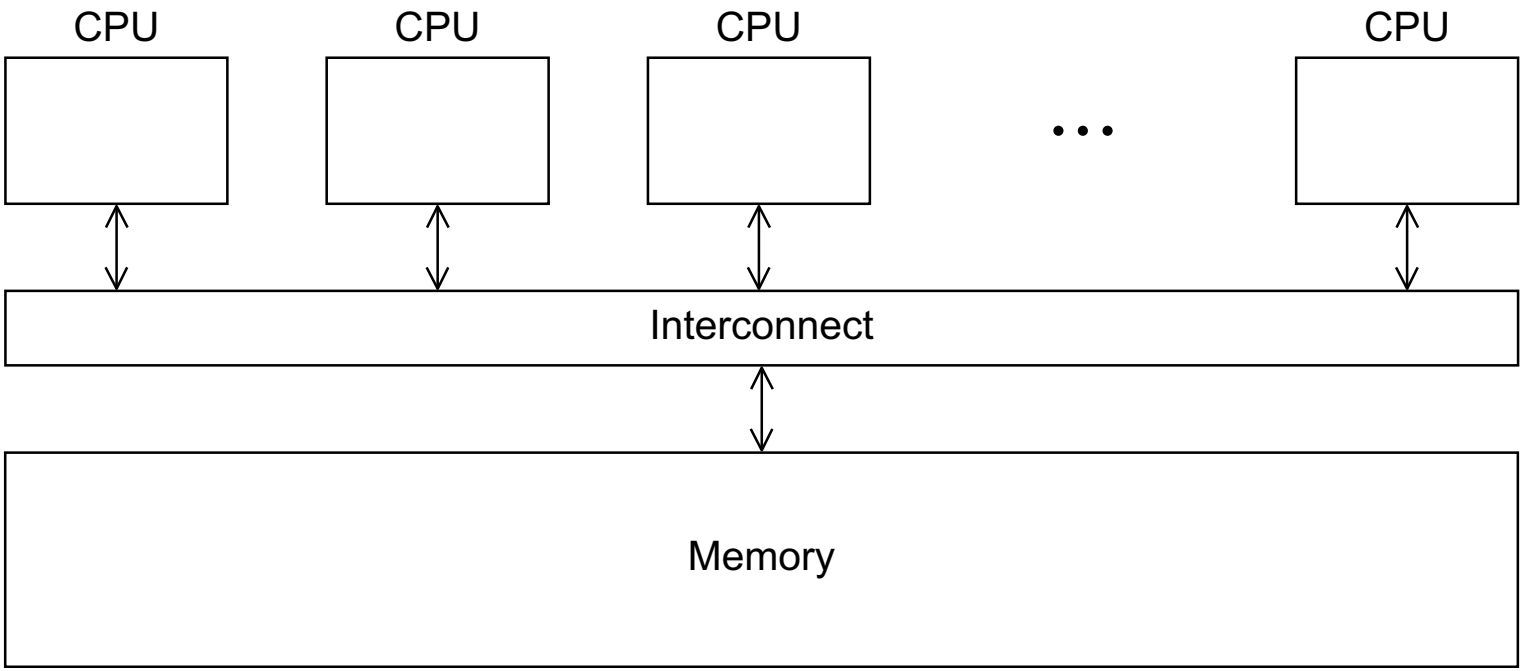
Roteiro

UFABC

- Escrevendo programas usando OpenMP
- Usar OpenMP para paralelizar laços seriais com pequenas mudanças no código fonte
- Explorar paralelismo de tarefas
- Sincronização explícita de threads
- Problemas típicos em programação para máquinas com memória compartilhada

- Uma API para programação paralela em memória compartilhada.
- MP = multiprocessing
- Projetada para sistemas no quais todas as threads ou processos podem, potencialmente, ter acesso à toda memória disponível.
- O sistema é visto como uma coleção de núcleos ou CPUs, no qual todos eles têm acesso à memória principal.

Um sistema com memória compartilhada



Pragmas

UFABC

- Instruções especiais para pre-processamento.
- Tipicamente adicionadas ao sistema para permitir comportamentos que não são parte do especificação básica de C.
- Compiladores que não suportam pragmas ignoram-nos.

```
1 #pragma
```

Hello World!

UFABC

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

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();
   printf("Hello from thread %d of %d\n", my_rank,
   ↪  thread_count);
}

int main(int argc, char* argv[]) {
   int thread_count = strtol(argv[1], NULL, 10);

#pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}
```

Compilando e executando

UFABC

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

./omp_hello 4
```

compilando

executando com 4 threads

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

resultados possíveis

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

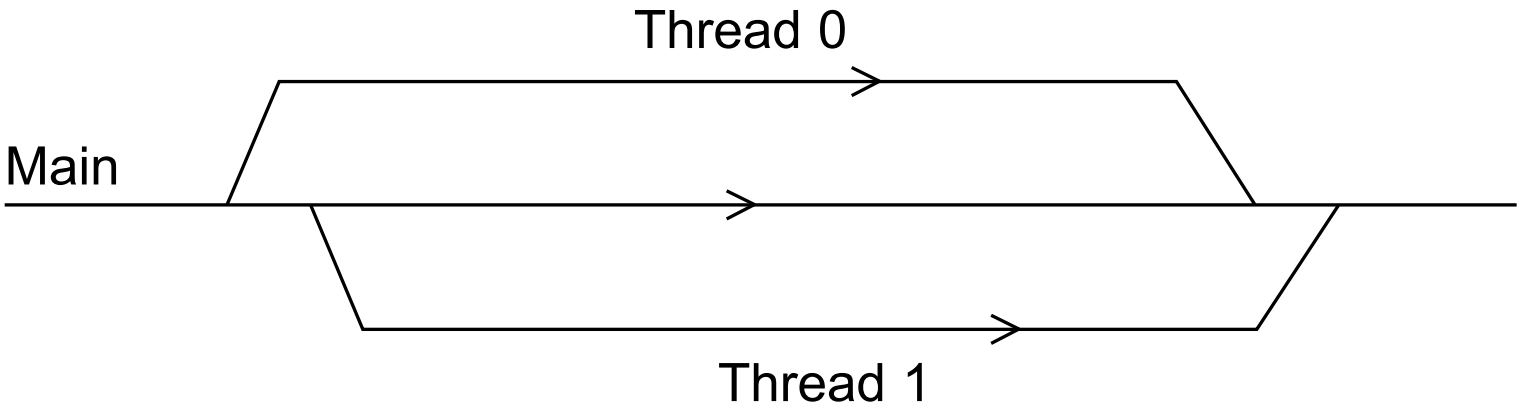
OpenMP pragmas

UFABC

```
1 #pragma omp parallel
```

- Diretiva paralela mais básica.
- O número de threads que executam o bloco que segue o pragma é determinado pelo sistema de runtime.

Um processo de duas threads fazendo fork e join



Cláusula

UFABC

- Uma cláusula (*clause*) é texto que modifica uma diretiva.
- A cláusula `num_threads` pode ser adicionada à uma diretiva paralela.
- Permite o programador especificar o número de threads que devem executar no bloco que segue o pragma.

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

Alguns comentários

UFABC

- Alguns sistemas podem limitar o número de threads que podem ser executadas.
- O padrão OpenMP não garante que serão iniciadas `thread_count` threads.
- A maioria dos sistemas pode iniciar centenas ou, até mesmo, milhares de threads.
- A não ser que desejemos iniciar um número muito grande de threads, quase sempre conseguiremos o número de threads desejado.

Terminologia

UFABC

- Em OpenMP, o conjunto de threads formado pela thread original e pelas novas threads é chamado team.
- A thread original é chamada master, e as threads adicionais são chamadas slaves.

Caso o compilador não tenha suporte ao OpenMP

UFABC

```
1  #include <omp.h>
2
3  //Em vez do acima escrever:
4
5  #ifdef _OPENMP
6  #include <omp.h>
7  #endif
```

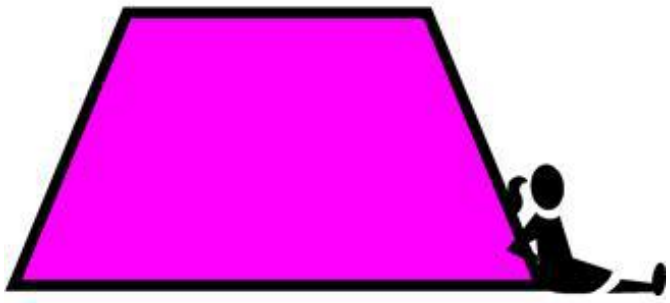
Caso o compilador não tenha suporte ao OpenMP

UFABC

- E no código cercar as chamadas ao OpenMP com `ifdef`

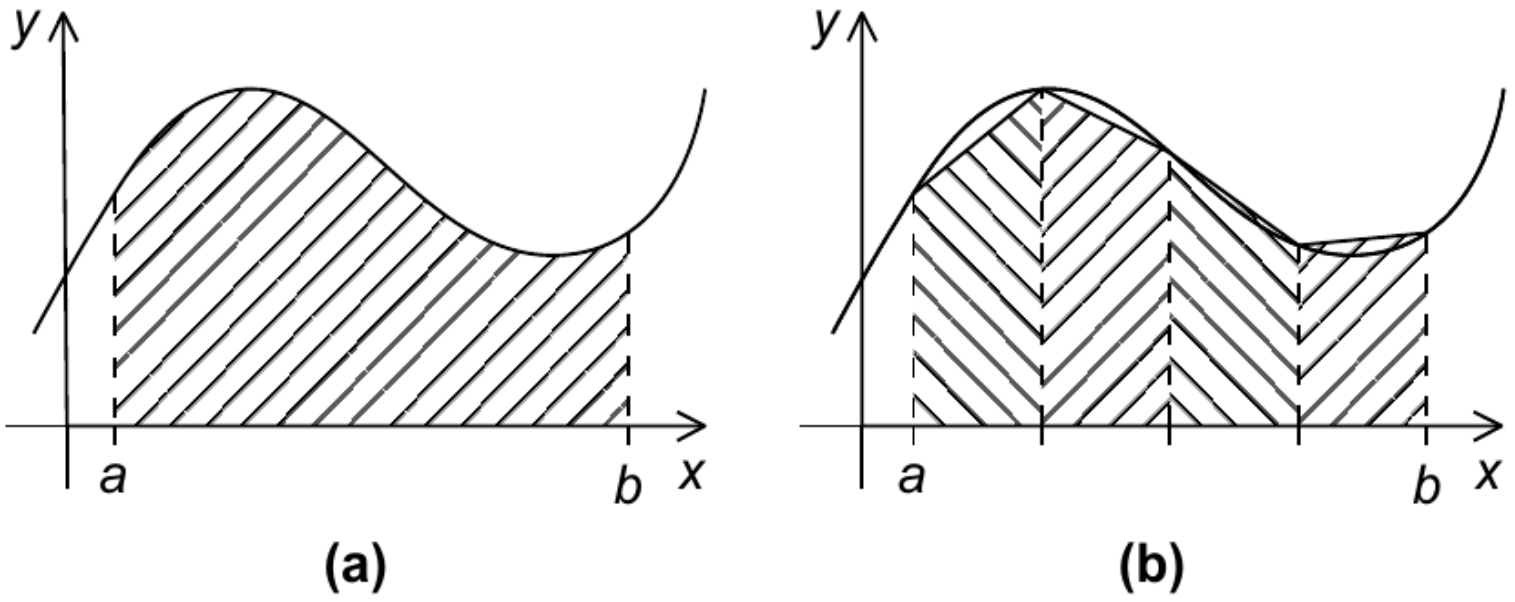
```
1  #ifdef _OPENMP
2  int my_rank = omp_get_thread_num ( );
3  int thread_count = omp_get_num_threads ( );
4  #else
5  int my_rank = 0;
6  int thread_count = 1;
7  #endif
```

A aproximação trapezoidal



A aproximação trapezoidal

A aproximação trapezoidal



Algoritmo sequencial

UFABC

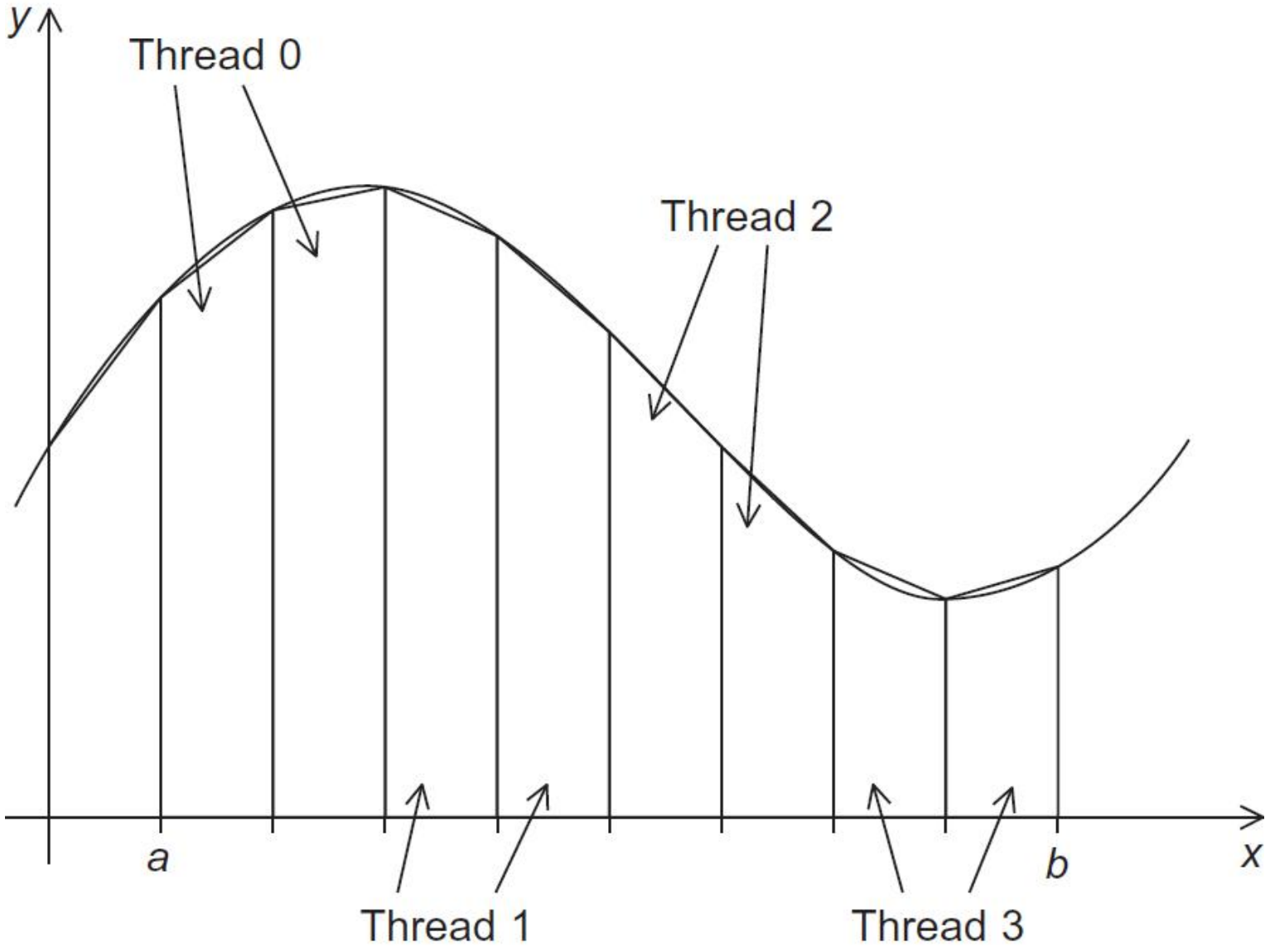
```
1  // Entradas: a, b e n
2  h = (b -a) / n;
3  approx = (f(a) + f(b))/2.0;
4  for (i = 1; i <= n-1; i++) {
5      x_i = a + i * h;
6      approx += f (x_i);
7  }
8  approx = h * approx;
```

Uma primeira versão com OpenMP

UFABC

- Identificamos 2 tipos de tarefas
  - Computação das áreas dos trapézios individualmente
  - Somar a área de cada um dos trapézios
- Não há comunicação entre as tarefas da primeira tarefa, mas todas se comunicam com a segunda tarefa
- Assumimos que haverá bem mais trapézios do que núcleos de processamento
- Logo, agregamos tarefas através da junção de trapézios vizinhos e deixamos a cargo de uma thread (e portanto de um núcleo) o cálculo de cada bloco de trapézios.

Atribuindo trapézios às threads



Condição de corrida

UFABC

Time	Thread 0	Thread 1
0	`global_result = 0` to register	finish `my_result`
1	`my_result = 1` to register	`global_result = 0` to register
2	add `my_result` to `global_result`	`my_result = 2` to register
3	store `global_result = 1`	add `my_result` to `global_result`
4		store `global_result = 2`

Resultados imprevisíveis eram obtidos de execuções quando threads tentavam executar simultaneamente o código:

```
1 global_result += my_result;
```

Exclusão mútua em OpenMP

UFABC

```
1 #pragma omp critical
```

- Apenas um thread pode executar o bloco logo após o pragma por vez
- Atenção é uma seção crítica global! Note que não há identificação alguma!

Primeira versão em OpenMP

UFABC

```
1  ...
2  #include <omp.h>
3
4  int main(int argc, char* argv[]) {
5     double  global_result = 0.0, a, b;
6     int     n, thread_count;
7
8     thread_count = strtol(argv[1], NULL, 10);
9     printf("Enter a, b, and n\n");
10    scanf("%lf %lf %d", &a, &b, &n);
11 #  pragma omp parallel num_threads(thread_count)
12    Trap(a, b, n, &global_result);
13
14    printf("With n = %d trapezoids, our estimate\n", n);
15    printf("of the integral from %f to %f = %.14e\n",
16       a, b, global_result);
17    return 0;
18 }
```

Primeira versão em OpenMP

UFABC

```
1  void Trap(double a, double b, int n, double* glb_res_p){
2     double  h, x, my_result, local_a, local_b;
3     int my_rank = omp_get_thread_num();
4     int thread_count = omp_get_num_threads();
5     h = (b-a)/n;
6     int local_n = n/thread_count;
7     local_a = a + my_rank*local_n*h;
8     local_b = local_a + local_n*h;
9     my_result = (f(local_a) + f(local_b))/2.0;
10    for (int i = 1; i <= local_n-1; i++) {
11      x = local_a + i*h;
12      my_result += f(x);
13    }
14    my_result = my_result*h;
15
16 #  pragma omp critical
17    *glb_res_p += my_result;
18 }
```

Escopo das variáveis

Escopo das variáveis

UFABC

- Em linguagens de programação, o escopo de uma variável é definido pelas partes do programa nas quais as variáveis podem ser usadas.
- Em OpenMP, o escopo de uma variável se refere ao conjunto de threads que podem acessar a variável em um bloco paralelo.

Escopo em OpenMP

UFABC

- Uma variável que pode ser acessada por todas as threads de um *team* possui um escopo shared.
- Uma variável que é acessada por apenas uma thread tem escopo private.
- O escopo das variáveis declaradas antes de um bloco paralelo é shared.

A cláusula de redução

A cláusula de redução

UFABC

- Nós fomos obrigados a declarar uma função com o protótipo:

```
1  void Trap(double a, double b, int n, double* glb_res_p);
```

Quando na verdade eu gostaria de ter feito:

```
1  void Trap(double a, double b, int n);
2  // E no meio do meu código
3
4  ...
5  global_result = Trap(a, b, n);
6  ...
```

A cláusula de redução

UFABC

Se usarmos uma função como:

```
1 double Local_trap(double a, double b, int n);
```

Poderíamos escrever:

```
1 global_result = 0.0;
2 # pragma omp parallel num_threads(thread_count)
3 {
4 #   pragma omp critical
5     global_result += Local_trap(a, b, n);
6 }
```

Mas isso é ruim! Estamos forçando a execução sequencial!

A cláusula de redução

UFABC

- Podemos resolver a execução sequencial declarando uma variável privada fora do bloco crítico dentro do bloco paralelo

```
1  global_result = 0.0;
2  # pragma omp parallel num_threads(thread_count)
3  {
4      double my_result = 0.0; //private
5      my_result += Local_trap(a, b, n);
6  #   pragma omp critical
7      global_result += my_result;
8  }
```

- Mas é quase igual o que tínhamos originalmente pra começo de conversa!
- Dá pra melhorar...

Operadores de redução

UFABC

- Um operador de redução (*reduction operator*) é um operador binário (tal como adição e multiplicação).
- Uma redução é uma computação que repetidamente aplica o mesmo operador de redução a uma sequência de operandos visando obter um único resultado.
- Todos os resultados intermediários da operação devem ser armazenadas na mesma variável: a variável de redução.

Cláusula de redução

UFABC

- Uma cláusula de redução pode ser adicionada a uma diretiva paralela.

```
1 reduction(<operator>: <variable list>)
```

- Alguns operadores disponíveis: `+`, `*`, `&`, `|`, `^`, `&&`, `||`
- Assim, podemos ”consertar”o código anterior:

```
1 global_result = 0.0;
2 # pragma omp parallel num_threads(thread_count) \
3     reduction(+: global_result)
4 global_result += Local_trap(a, b, n);
```

DIRETIVA PARALLEL FOR

Parallel for

UFABC

- Dispara um time de threads para executar o bloco lógico que segue.
- O bloco lógico que segue a diretiva precisa ser um laço for.
- Além disso, a diretiva `parallel for` paraleliza o laço dividindo as iterações entre os threads.

Exemplo

UFABC

```
1     // Entradas: a, b e n
2     h = (b -a) / n;
3     approx = (f(a) + f(b))/2.0;
4  #  pragma omp parallel for num_threads(thread_count) \
5         reduction (+: approx)
6     for (i = 1; i <= n-1; i++) {
7         x_i = a + i * h;
8         approx += f (x_i);
9     }
10    approx = h * approx;
```

Tipos de sentenças for paralelizáveis pelo OpenMP

UFABC

```
for ( index = start ; index < end  ; index++
                      index <= end    ++index
                      index >= end    index--
                      index > end     --index
                                      index += incr
                                      index -= incr
                                      index = index + incr
                                      index = incr + index
                                      index = index - incr )
```

Cuidado...

UFABC

- A variável `index` precisa ser to tipo inteiro ou apontador (e.x., não pode ser `float`).
- As expressões `start`, `end`, and `incr` precisam ter tipos compatíveis. Por exemplo, se `index` é um apontador, então `incr` precisa ser do tipo inteiro.
- As expressões `start`, `end`, and `incr` não podem mudar durante a execução do laço.
- Durante a execução do laço, a variável `index` somente pode ser modificada pela "expressão de incrementar" dentro da sentença for.

UFABC

```
fibo[ 0 ] = fibo[ 1 ] = 1;
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

note 2 threads

```
fibo[ 0 ] = fibo[ 1 ] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

1 1 2 3 5 8 13 21 34 55

this is correct

but sometimes we get this

1 1 2 3 5 8 0 0 0 0

Que tá pegando?

UFABC

- Compiladores OpenMP não checam dependências entre iterações do laço que está sendo paralelizado com a diretiva parallel for.
- Um laço cujos resultados de uma ou mais iterações dependem de outras iterações não pode, no geral, ser corretamente paralelizado por OpenMP.

Estimando π

UFABC

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \ldots + (-1)^n \frac{1}{2n+1} + \ldots\right)$$

```
1  double factor = 1.0;
2  double sum = 0.0;
3  for (i = 0; i < n; i ++) {
4      sum += factor/(2*i +1);
5      factor = -factor;
6  }
7  pi = 4.0 * sum;
```

Estimando π - OpenMP Versão 1

UFABC

```
1     double factor = 1.0;
2     double sum = 0.0;
3  # pragma omp parallel for num_threads(thread_count) \
4        reduction(+: sum)
5     for (i = 0; i < n; i ++) {
6         sum += factor/(2*i +1);
7         factor = -factor;
8     }
9     pi = 4.0 * sum;
```

- Nesta versão temos um *loop carried dependency*: o valor de `fator` depende da iteração anterior do laço.
- Não será paralelizado corretamente.

Estimando π - OpenMP Versão 2

UFABC

```
1     double sum = 0.0;
2  # pragma omp parallel for num_threads(thread_count) \
3        reduction(+: sum) private(factor)
4     for (i = 0; i < n; i ++) {
5        factor = (i % 2 == 0) ? 1 : -1;
6        sum += factor/(2*i +1);
7     }
8     pi = 4.0 * sum;
```

- Assim retiramos a dependência. Não era uma dependência real de dados, por isto conseguimos alterar o código.
- Note que a variável `factor` é definida como `private`, assim, as execuções das diferentes threads não interferem umas com as outras.

A cláusula `default`

UFABC

- Deixa o programador definir o escopo de cada variável em um bloco.

```
1 default (none)
```

- Com esta cláusula o compilador vai requerer que definamos o escopo de cada variável usada em um bloco e que foi declarada for a do bloco.
- Recebe também os valores `shared` (padrão) ou `private`

A cláusula default

UFABC

```
1       double sum = 0.0;
2   # pragma omp parallel for num_threads(thread_count) \
3           default(none) reduction(+: sum) \
4           private(i, factor) shared(n)
5       for (i = 0; i < n; i ++) {
6           factor = (i % 2 == 0) ? 1 : -1;
7           sum += factor/(2*i +1);
8       }
9       pi = 4.0 * sum;
```

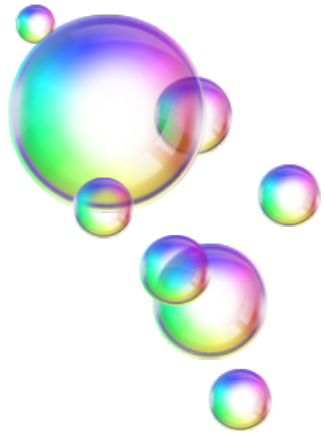
Laços em OpenMP Parte 2 - Ordenação

A estrela mal compreendida - Bubble Sort

UFABC

```c
void Bubble_sort(
                  int a[] /* in/out */,
                  int n   /* in */) {
    int list_length, i, temp;

    for (list_length = n; list_length >= 2;
    ↪  list_length--)
        for (i = 0; i < list_length - 1; i++)
            if (a[i] > a[i+1]) {
                temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
            }
}
```



Ordenação por transposição par-ímpar

UFABC

- Funciona em uma sequência de fases
- Fases pares - comparam e trocam iniciando dos índices pares
  - `(a[0],a[1]),(a[2],a[3]),(a[4],a[5])...`
- Fases ímpares - comparam e trocam iniciando dos índices ímpares
  - `a[0],(a[1],a[2]),(a[3],a[4]),(a[5],a[6])...`

Exemplo

UFABC

- Começo: `5, 9, 4, 3`
- Fase par: compara e troca `(5, 9), (4, 3)`
  - Resultado: `5, 9, 3, 4`
- Fase ímpar: compara e troca `(9, 3)`
  - Resultado: `5, 3, 9, 4`
- Fase par: compara e troca `(5, 3), (9, 4)`
  - Resultado: `3, 5, 4, 9`
- Fase ímpar: compara e troca `(5, 4)`
  - Resultado: `3, 4, 5, 9`

Código sequencial

UFABC

```
1  void Odd_even_sort(int a[], int n) {
2      int phase, i, temp;
3      for (phase = 0; phase < n; phase++)
4          if (phase % 2 == 0) { /* Even phase */
5              for (i = 1; i < n; i += 2)
6                  if (a[i - 1] > a[i]) {
7                      temp = a[i];
8                      a[i] = a[i - 1];
9                      a[i - 1] = temp;
10                 }
11         } else { /* Odd phase */
12             for (i = 1; i < n - 1; i += 2)
13                 if (a[i] > a[i+1]) {
14                     temp = a[i];
15                     a[i] = a[i+1];
16                     a[i+1] = temp;
17                 }
18         }
19 }
```

OpenMP - Primeira Versão

UFABC

```
1  void Odd_even_sort(int a[], int n) {
2    int phase, i, temp;
3    for (phase = 0; phase < n; phase++)
4      if (phase % 2 == 0) { /* Even phase */
5  #     pragma omp parallel for num_threads(thr_count) \
6          default(none) shared(a, n) private(i, tmp)
7        for (i = 1; i < n; i += 2)
8          if (a[i - 1] > a[i]) {
9            temp = a[i]; a[i] = a[i - 1]; a[i - 1] = temp;
10         }
11     } else { /* Odd phase */
12 #     pragma omp parallel for num_threads(thr_count) \
13         default(none) shared(a, n) private(i, tmp)
14       for (i = 1; i < n - 1; i += 2)
15         if (a[i] > a[i+1]) {
16           temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;
17         }
18     }
19 }
```

OpenMP - Segunda Versão

UFABC

```
1  void Odd_even_sort(int a[], int n) {
2      int phase, i, temp;
3  #   pragma omp parallel num_threads(thread_count) \
4        default(none) shared(a, n) private(i, tmp, phase)
5      for (phase = 0; phase < n; phase++)
6        if (phase % 2 == 0) { /* Even phase */
7  #       pragma omp for
8            for (i = 1; i < n; i += 2)
9              if (a[i - 1] > a[i]) {
10               temp = a[i]; a[i] = a[i - 1]; a[i - 1] =
                   ↪  temp;
11             }
12       } else { /* Odd phase */
13 #       pragma omp for
14           for (i = 1; i < n - 1; i += 2)
15             if (a[i] > a[i+1]) {
16               temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;
17             }
18       }
19 }
```

Comparação de desempenho entre as duas versões

UFABC

Número de threads	1	2	3	4
Duas diretivas `parallel for`	0.770	0.453	0.358	0.305
Duas diretivas `for`	0.732	0.376	0.294	0.239

Escalonamento de laços

Escalonamento de laços

- Queremos paralelizar o seguinte laço

```
1  sum = 0.0
2  for (i = 0; i <= n; i++)
3      sum += f(i);
```

Thread	Iterações
0	0, n/t, 2n/t, ...
1	1, n/t + 1, 2n/t + 1, ...
...	...
t - 1	t - 1, n / t + t - 1, 2n / t + t - 1, ...

- Atribuição cíclica

Definindo f

UFABC

```
1  double f (int i) {
2      int j, start = i * (i + 1) /2, finish = start + i;
3      double return_val = 0.0;
4      for (j = start; j <= finish; j++) {
5          return_val += sin(j);
6      }
7      return return_val;
8  }
```

Resultados

- `f(i)` chama a função `sin`, `i` vezes
- Assuma que o tempo para executar `f(2 * i)` seja aproximadamente o dobro do tempo de chamar `f(i)`

Considerando $n = 10000$

Threads	Atribuição	Tempo	Speedup
1	-	3.67	1
2	default	2.76	1.33
2	cíclico	1.84	1.99

A cláusula schedule

UFABC

- Default schedule

```
1  sum = 0.0
2  #pragma omp parallel for num_threads(thread_count) \
3      reduction(+:sum)
4  for (i = 0; i <= n; i++)
5      sum += f(i);
```

- Cyclic schedule

```
1  sum = 0.0
2  #pragma omp parallel for num_threads(thread_count) \
3      reduction(+:sum) schedule(static, 1)
4  for (i = 0; i <= n; i++)
5      sum += f(i);
```

~schedule (type, chunksize)

UFABC

- `type` pode ser:
  - static - as iterações podem ser atribuídas aos threads antes do loop ser executado.
  - dynamic ou guided - as iterações são atribuídas aos threads enquanto o laço está executando.
  - auto - o compilador e/ou o ambiente de execução determinam o schedule.
  - runtime - o escalonamento é determinado durante a execução (através de uma configuração passada como variável de ambiente, por exemplo).
- O `chunksize` deve ser um inteiro positivo.

Escalonamento `static`

- Doze iterações: 0,1,2,...11 e três threads

```
schedule(static, 1)
```

Thread Rank	Iterações
0	0, 3, 6, 9
1	1, 4, 7, 10
2	2, 5, 8, 11

Escalonamento `static`

UFABC

- Doze iterações: 0,1,2,...11 e três threads

```
schedule(static, 2)
```

Thread Rank	Iterações
0	0, 1, 6, 7
1	2, 3, 8, 9
2	4, 5, 10, 11

Escalonamento `static`

UFABC

- Doze iterações: 0,1,2,...11 e três threads

```
schedule(static, 4)
```

Thread Rank	Iterações
0	0, 1, 2, 3
1	4, 5, 6, 7
2	8, 9, 10, 11

Escalonamento `dynamic`

UFABC

- As iterações também são quebradas em nacos (*chunks*) de tamanho `chunksize` consecutivas
- Cada thread executa um naco e quando uma thread termina um naco, ele requisita um novo para inicar o processamento ao ambiente de execução
- O processo se repete enquanto houver nacos disponíveis
- O valor `chunksize` pode ser omitido. Neste caso um valor padrão de 1 é utilizado.

Escalonamento `guided`

UFABC

- Assim como o dinâmico, cada thread também executa um naco e quando acaba requisita um novo ao sistema de execução.
- Contudo, no escalonador `guided` conforme os nacos são completados, o tamanho dos novos nacos diminuem.
- Se nenhum `chunksize` for especificado, o tamanho dos nacos diminui até 1.
- Se `chunksize` for especificado, o tamanho do naco diminui até `chunksize` com a exceção do último naco (que pode ser menor do que `chunksize`).

Exemplo: Aproximação Trapezoidal

UFABC

- Atribuição de iterações 1 a 9999 usando o escalonador `guided` com duas threads

Table 5.3 Assignment of Trapezoidal Rule Iterations 1–9999 using a `guided` Schedule with Two Threads

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1–5000	5000	4999
1	5001–7500	2500	2499
1	7501–8750	1250	1249
1	8751–9375	625	624
0	9376–9687	312	312
1	9688–9843	156	156
0	9844–9921	78	78
1	9922–9960	39	39
1	9961–9980	20	19
1	9981–9990	10	9
1	9991–9995	5	4
0	9996–9997	2	2
1	9998–9998	1	1
0	9999–9999	1	0

Escalonamento `runtime`

UFABC

- O sistema usa a variável de ambiente `OMP_SCHEDULE` para determinar, em tempo de execução, como fazer o escalonamento.
- A variável `OMP_SCHEDULE` pode receber qualquer um dos valores que puderem ser usados para `static`, `dynamic` ou `guided`

Produtores e consumidores

- Podem ser enxergadas como uma abstração de uma fila de consumidores aguardando para pagar pelas suas compras em um mercado
- Uma estrutura natural e que aparece recorrentemente em aplicações multithreaded.
- Por exemplo, suponha que haja diversos threads *produtores* e diversos *threads* consumidores.
  - Produtores produzem, por exemplo, requisições por dados
  - Consumidores podem consumir essas requisições produzindo ou encontrado os dados requisitados.

Passagem de mensagem

UFABC

- Cada thread pode, por exemplo, ter uma fila de mensagens compartilhada e, quando um thread desejar enviar uma mensagem para ou outro, ele pode enfileirar a mensagem na fila de mensagens do thread de destino.
- Um thread pode receber uma mensagem desenfileirando a mensagem que estiver no início da sua fila.

Rascunho da solução

UFABC

```
1  for (sent_msgs = 0; sent_msg < send_max; sent_msgs++) {
2      Send_msg();
3      Try_receive();
4  }
5
6  while (!Done())
7      Try_receive();
```

Enviando mensagens

UFABC

```
1       msg = random();
2       dest = random % thread_count;
3   #   pragma omp critical
4       Enqueue(queue, dest, my_rank, msg);
```

Recebendo mensagens

UFABC

```
1       if (queue_size == 0) return;
2       else if (queue_size == 1)
3   #       pragma omp critical
4           Dequeue(queue, &src, &msg);
5       else
6           Dequeue(queue, &src, &msg);
7       Print_message(src, msg);
```

- Como manter `queue_size` protegido contra condições de corrida?

Detectando a finalização da execução

UFABC

```
1  queue_size = enqueued - dequeued;
2  if (queue_size == 0 && done_sending == thread_count)
3      return TRUE;
4  else
5      return FALSE;
```

Cada thread incrementa `done_sending` quando acabar o seu laço.

Inicialização (1)

UFABC

- Quando o programa começar a executar, um único thread, o thread principal, lê os argumentos da linha de comando e aloca uma array de filas de mensagens com um elemento para cada thread.
- Essa array precisa ser compartilhada entre todos os threads já que cada thread pode enviar mensagens para qualquer outro thread e, portanto, pode enfileirar mensagens em qualquer uma das filas.

Inicialização (2)

UFABC

- Um ou mais threads podem terminar de alocar as suas filas antes dos demais.
- Precisamos de uma barreira explícita para que quando um thread encontrar a barreira ele fique esperando até que os demais threads do time atinjam a barreira também.
- Depois que todos os threads tiverem alcançado a barreira, então todos os membros do time podem prosseguir.

```
1 #pragma omp barrier
```

Mas pera aí!

UFABC

```
1  queue_size = enqueued - dequeued;
2  if (queue_size == 0 && done_sending == thread_count)
3      return TRUE;
4  else
5      return FALSE;
```

- E como incrementar `done_sending`?

A diretiva `atomic`

UFABC

```
#pragma omp atomic
```

- Ao contrário da diretiva **`critical`**, a diretiva **`atomic`** pode apenas proteger seções críticas que consistem de uma simples atribuição em C.
- Além disto, os formatos aceitos pela diretiva **`atomic`** são os seguintes:

```
x <op>= <expressão>; //Não pode fazer referência a x
x++;
x--;
++x;
--x;

#pragma omp atomic
x += y++; //Protege escrita a x mas não a y
```

A diretiva `atomic`

UFABC

- `<op>` pode ser um dos seguintes operadores binários
  - `+ - * / & ^ | << >>`
- Muitos processadores oferecem instruções especiais e atômicas que fazem *load-modify-store* atomicamente
- Uma seção crítica com esse formato pode ser executada de maneira muito mais eficiente do que através de construções mais elaboradas e gerais para isolamento de sessões críticas.

Seções críticas em OpenMP

UFABC

Temos, neste momento, 3 seções críticas em nosso programa

- `done_sending++;`
- `Enqueue(q_p, my rank, mesg);`
- `Dequeue(q_p, &src, &mesg);`

Mas nem todas elas precisam ser isoladas:

- `done_sending++;` é independente das demais
- Entre o segundo e o terceiro não é preciso que sejam totalmente isolados
  - O Thread 0 pode estar enfileirando na caixa de mensagens do Thread 2
  - Enquanto isto o Thread 5 poderia estar enfileirando na caixa de mensagens do Thread 0

Seções críticas em OpenMP

UFABC

- Para o OpenMP temos neste momento 2 seções críticas
  - `done_sending++;` que é protegido pelo atomic
  - `Enqueue(q_p, my_rank, mesg);` e `Dequeue(q_p, &src, &mesg);` protegidos pelo `critical`
- Como todos os blocos que estão dentro do `critical` serão serializados, isso pode ser muito ruim para o desempenho.

Seções críticas em OpenMP

UFABC

- OpenMP tem suporte ao uso de nomes para seções críticas

```
1 #pragma omp critical(name)
```

- Quando fazemos isto, blocos protegidos por diretivas `critical` com nomes diferentes podem ser executadas simultaneamente.
- Contudo, nomes e regiões são definidos em tempo de compilação. Não é possível criar regiões críticas (tal qual fizemos com Pthread mutexes) em tempo de execução.
- Logo não resolve o nosso problema para as filas (pois o número de filas é definido pelo usuário como um parâmetro).

Travas (*locks*)

UFABC

- Uma trava (*lock*) é uma estrutura de dados e funções associadas que permitem o programador garantir exclusão mútua em uma seção crítica.
- Rascunho do modo de uso:

```
// Executado por uma thread
Inicializa a estrutura de lock;
...
// Executado por múltiplas threads
Tenta obter a trava;
Seçao crítica;
Destrava;
...
// Executado por uma thread
Destrói a estrutuda de lock;
```

```
1  void omp_init_lock(omp_lock_t* lock_p     /* out */);
2  void omp_set_lock( omp_lock_t* lock_p     /* in/out */);
3  void omp_unset_lock(omp_lock_t* lock_p    /* in/out */);
4  void omp_destroy_lock(omp_lock_t* lock_p /* in/out */);
```

Solução usando locks

UFABC

Saímos disto:

```
1  #  pragma omp critical
2     Enqueue(queue, dest, my_rank, msg);
```

Para isto:

```
1  omp_set_lock(&q_p->lock);
2  Enqueue(queue, dest, my_rank, msg);
3  omp_unset_lock(&q_p->lock);
```

Solução usando locks

UFABC

Saímos disto:

```
1  #  pragma omp critical
2     Dequeue(q_p, &src, &msg);
```

Para isto:

```
1  omp_set_lock(&q_p->lock);
2  Dequeue(q_p, &src, &msg);
3  omp_unset_lock(&q_p->lock);
```

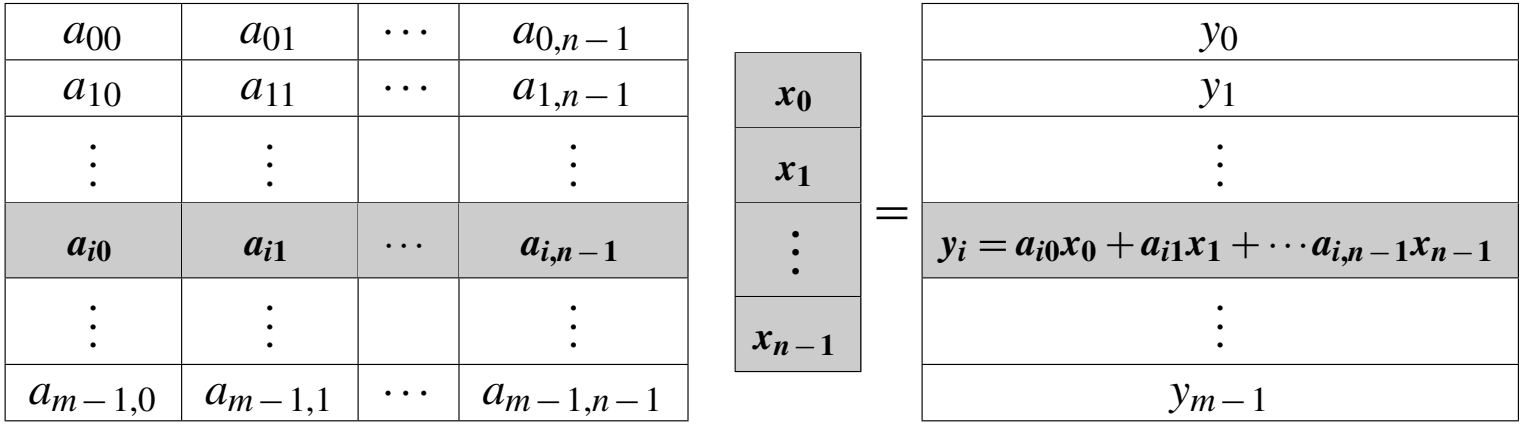
Algumas ressalvas...

UFABC

1. Você não deveria misturar diferentes tipos de exclusão mútua para uma única seção crítica.
2. O OpenMP não garante que as travas sejam justas.
3. Pode ser perigoso aninhar estruturas de exclusão mútua diferentes.

```
1  #pragma omp critical
2  y = f (x);
3  // E em outra thread
4  #pragma omp atomic
5  y++;
6  ...
7  double f(double x) { //chamado na linha 2
8  #    pragma omp critical
9       z = g(x); /* z é compartilhado */
10      ...
11 }
```

Exemplo: Multiplicação Matriz-Vetor




```
1  for (i = 0; i < m ; i++) {
2      y [i] = 0.0;
3      for (j = 0; j < n; j++)
4          y[i] += A[i][j]*x[j];
5  }
```

Exemplo: Multiplicação Matriz-Vetor

UFABC

```
1  #    pragma omp parallel for num_threads(thread_count) \
2           default(none) private(i,j) shared(A, x, y, m, n)
3       for (i = 0; i < m ; i++) {
4           y [i] = 0.0;
5           for (j = 0; j < n; j++)
6               y[i] += A[i][j]*x[j];
7       }
```

Table 5.4 Run-Times and Efficiencies of Matrix-Vector Multiplication (times in seconds)

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Thread Safety

UFABC

```
1  void Tokenize(
2    char* lines[] /* in/out */,
3    int line count /* in */,
4    int thread count /* in */) {
5    int my_rank, i, j;
6    char *my_token;
7
8   ...
```

Thread Safety

UFABC

```
1   # pragma omp parallel num threads(thread_count) \
2       default(none) private(my_rank, i, j, my_token) \
3       shared(lines, line_count)
4     { my_rank = omp_get_thread_num();
5   #   pragma omp for schedule(static, 1)
6       for (i = 0; i < line_count; i++) {
7         printf("Thread %d > line %d = %s", my_rank, i,
8                lines[i]);
9         j = 0;
10        my_token = strtok(lines[i], " \t\n");
11        while (my_token != NULL) {
12          printf("Thread %d > token %d = %s\n", my_rank,
13                 j, my_token);
14          my_token = strtok(NULL, " \t\n");
15          j++;
16        }
17      } /* for i */
18    } /* omp parallel */
19  } /* Tokenize */
```

Conclusões

Conclusões

UFABC

- OpenMP é uma padrão para a programação de sistemas com memória compartilhada
- OpenMP utiliza tanto funções especiais quanto diretivas do preprocessador chamadas de pragmas
- Programas OpenMP iniciam diversos threads e não diversos processos
- Diversas diretivas do OpenMP pode ser modificadas por cláusulas

Conclusões

- Um dos maiores problemas para o desenvolvimento de programas em arquiteturas com memória compartilhada é a ocorrência de condições de corrida
- OpenMP provê diversos mecanismos para garantir exclusão mútua em seções críticas
  - Diretivas `critical`
  - Diretivas `critical` nomeadas
  - Diretivas `atomic`
  - Travas (*locks*) simples

Conclusões

- Por padrão, a maior parte das implementações utiliza um particionamento por bloco das iterações do laço
- OpenMP oferece uma variedade de opções de escalonamento
- Em OpenMP o escopo da variável é a coleção de threads a partir dos quais a variável é acessível
- Uma redução é uma computação que repetidamente aplica a mesma operação de redução a uma sequência de operandos para obter um único resultado.