

DISCIPLINA

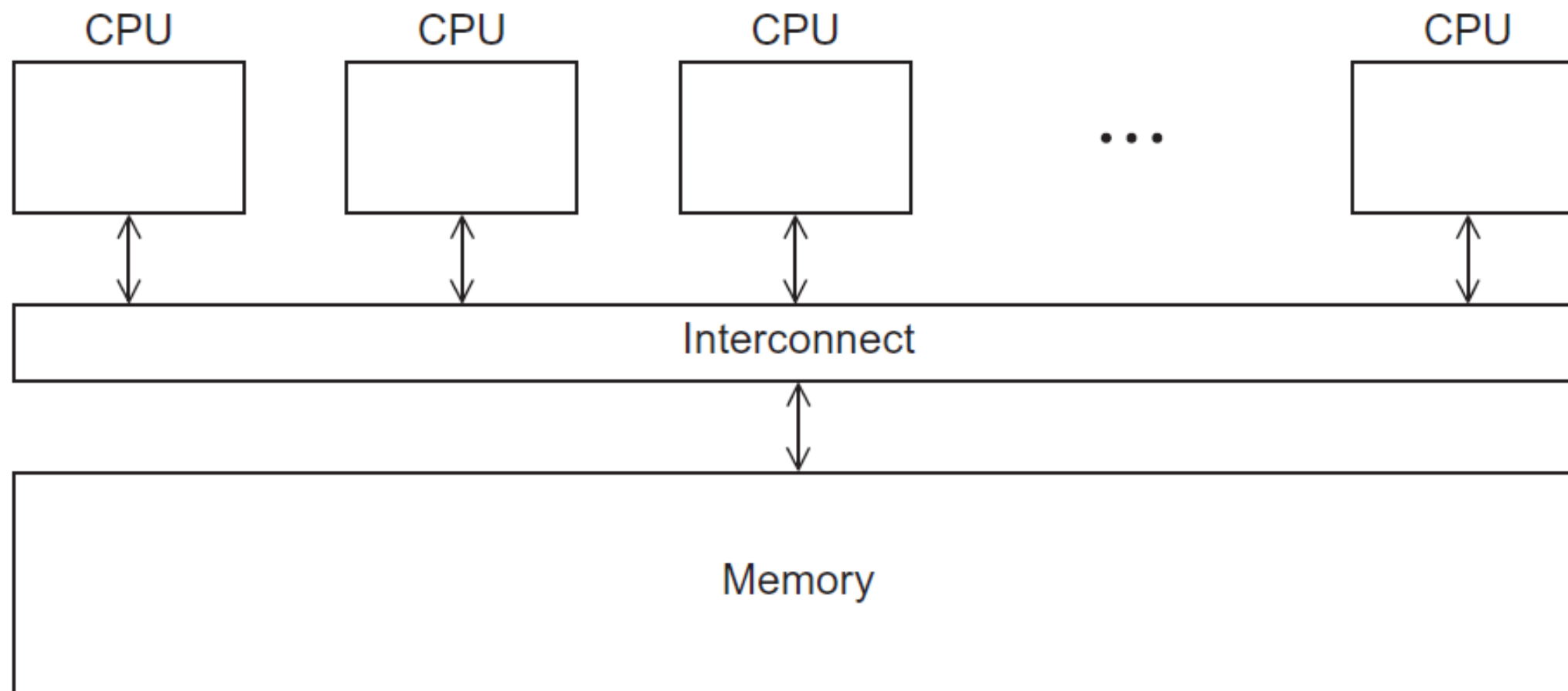
Introdução à Computação Paralela - OpenMP -

Prof. Kayo Gonçalves

BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

OpenMP

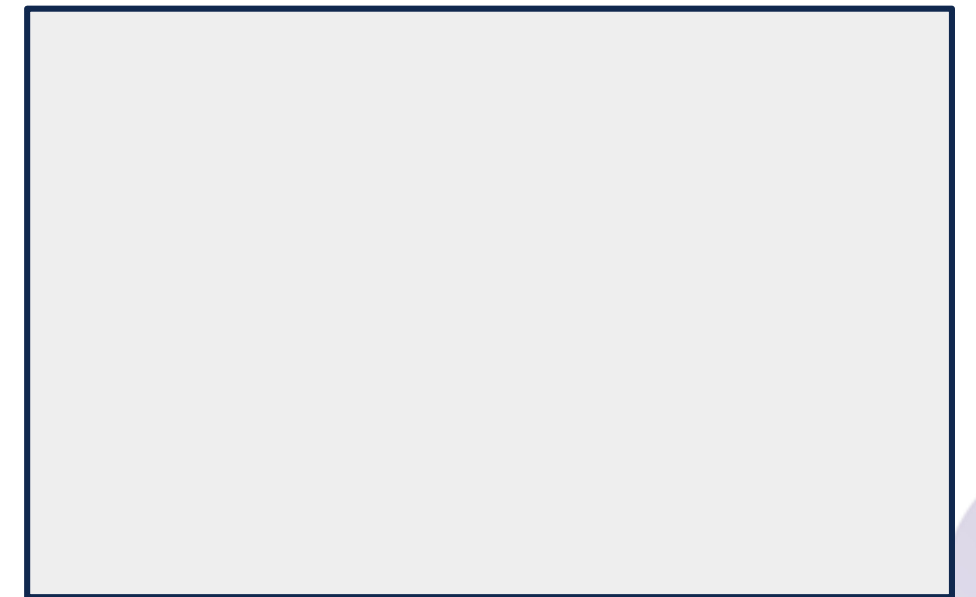
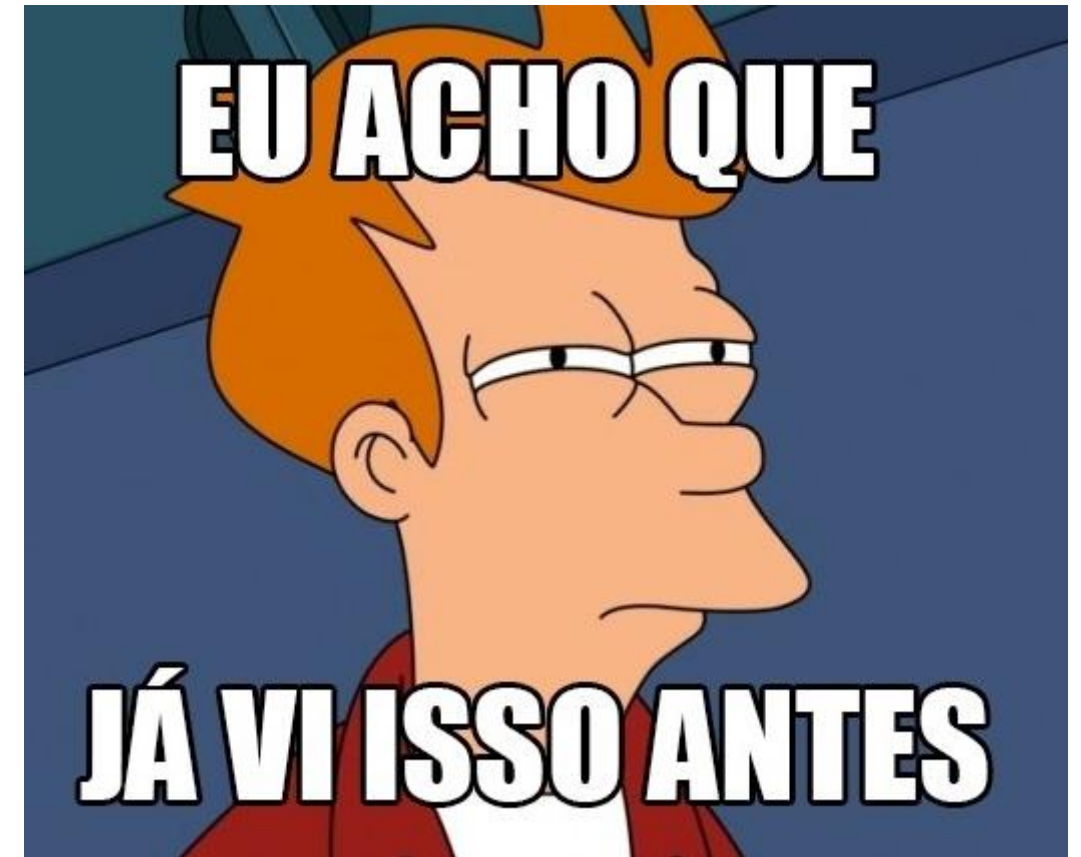
- É uma API para programação paralela em memória compartilhada.



#Pragmas

- São instruções especiais do pré-processador
 - O pré-processador é uma parte do compilador que executa operações preliminares (compilar código condicionalmente, incluindo arquivos, etc ...) em seu código antes que o compilador o veja. Essas transformações são **lexicais**, o que significa que a saída do pré-processador ainda é texto.
- Compiladores que não suportam **#pragmas** os ignoram.

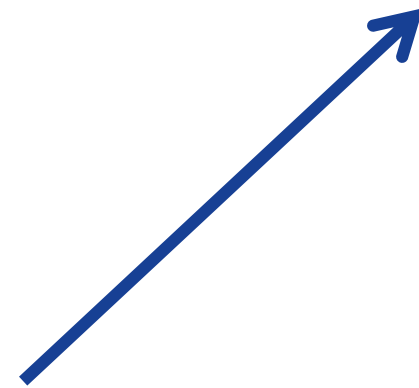
```
# pragma omp parallel num_threads( thread_count )  
Hello();
```



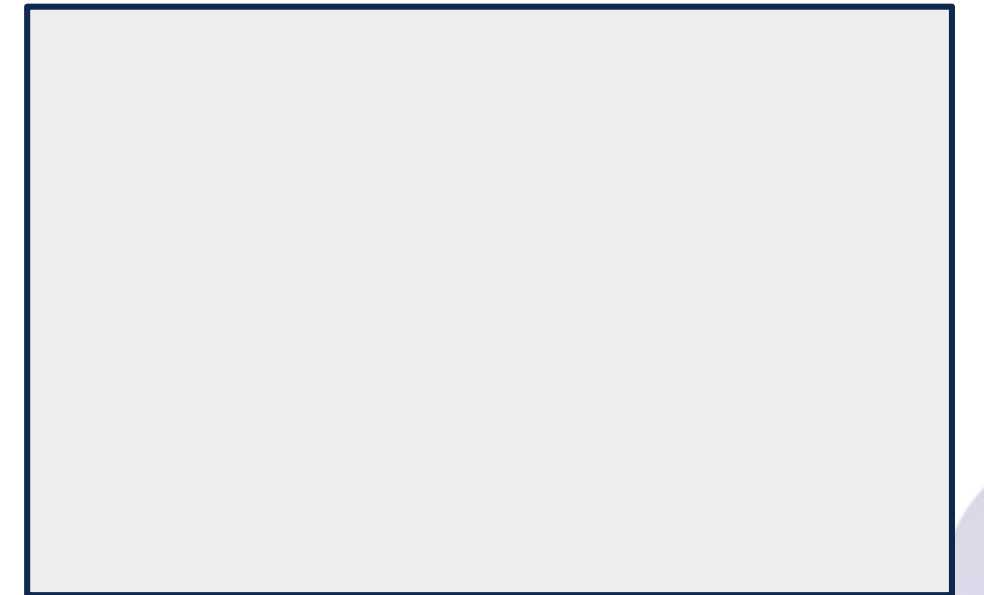
Compilar

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

```
g++ -g -Wall -fopenmp -o omp_hello omp_hello.cpp
```



Link para a biblioteca Pthreads



Executar

./omp_hello <número de threads>

./omp_hello 2

Hello from thread 0 of 2

Hello from thread 1 of 2

Depende do código
Não obrigatório

./omp_hello 4

Hello from thread 3 of 4

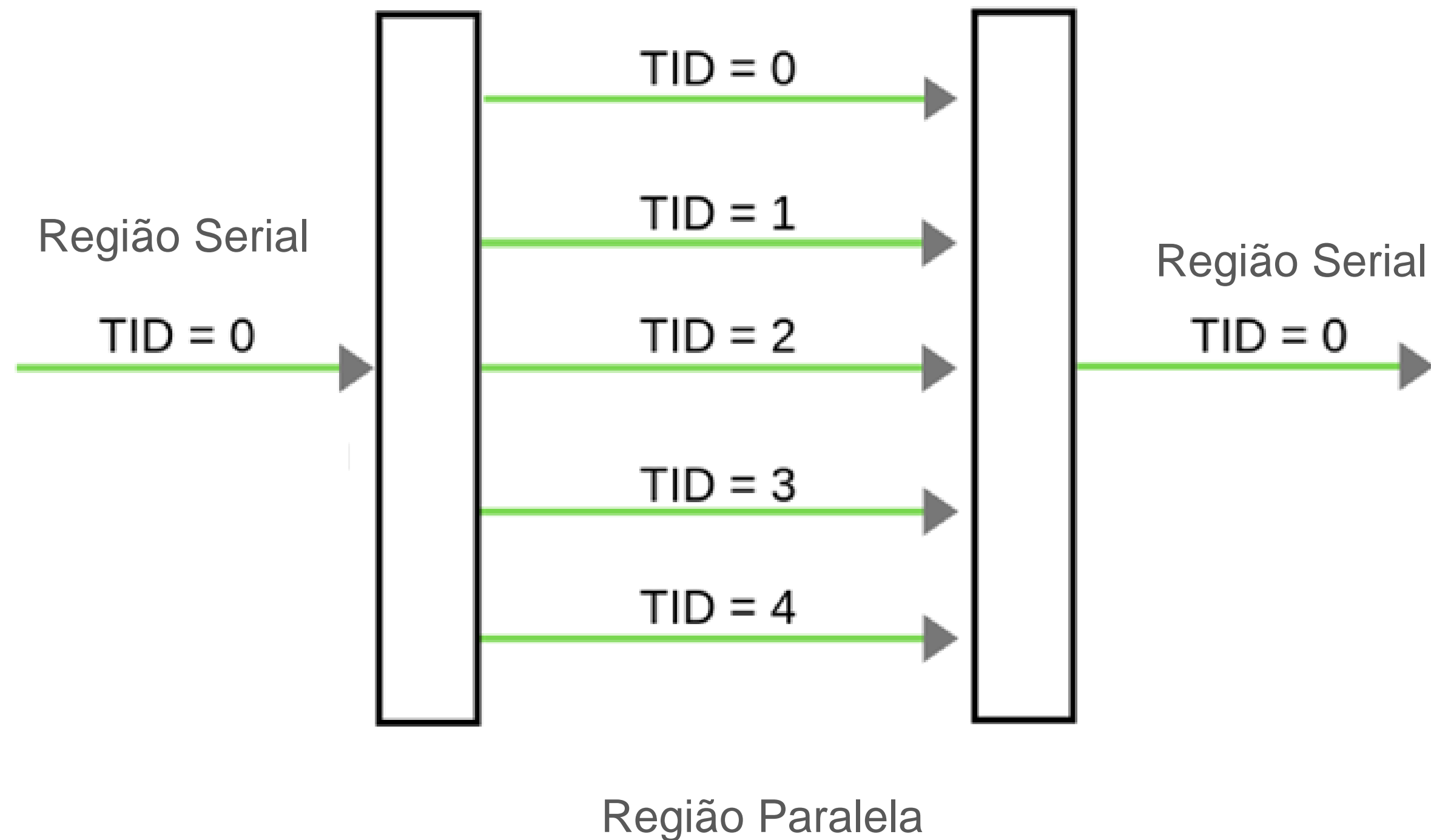
Hello from thread 1 of 4

Hello from thread 0 of 4

Hello from thread 2 of 4



Como são as threads?

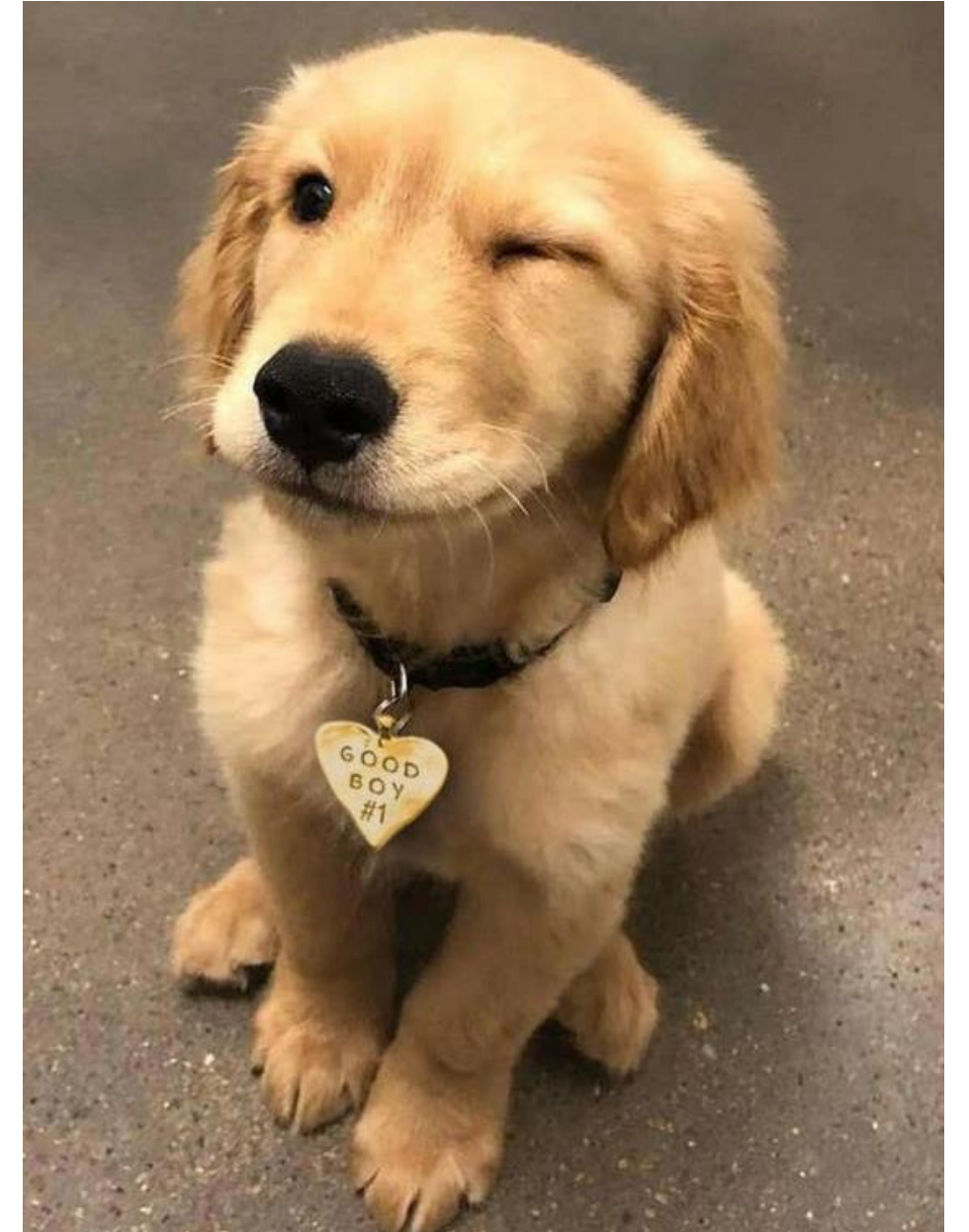


The background is a solid dark blue. It features several horizontal bars of a lighter blue color. There are two bars on the left side, one near the top and one near the bottom. On the right side, there are two bars near the top and one near the bottom. A white square is located in the bottom right corner.

Olá Mundo

Direto ao ponto...

- Você irá criar um código em C/C++
- Adicione `omp.h` no cabeçalho
- Utilize `#pragmas` para identificar a região paralela
- Você pode especificar o número de threads no terminal (antes de executar) ou no código.



Hello World (1)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  int main(int argc, char* argv[])
6  {
7
8      // Beginning of parallel region
9      #pragma omp parallel
10     {
11         printf("Hello World... from thread = %d\n", omp_get_thread_num());
12     }
13     // Ending of parallel region
14 }
```

Hello World (2)

```
akbar@ubuntu: ~/Desktop
File Edit View Search Terminal Help
akbar@ubuntu:~/Desktop$ export OMP_NUM_THREADS=5
akbar@ubuntu:~/Desktop$ gcc -o hello -fopenmp hello.c
akbar@ubuntu:~/Desktop$ ./hello
Hello World... from thread = 1
Hello World... from thread = 0
Hello World... from thread = 4
Hello World... from thread = 3
Hello World... from thread = 2
akbar@ubuntu:~/Desktop$
```

Hello World (3)

```
akbar@ubuntu: ~/Desktop
File Edit View Search Terminal Help
akbar@ubuntu:~/Desktop$ ./hello
Hello World... from thread = 1
Hello World... from thread = 0
Hello World... from thread = 4
Hello World... from thread = 3
Hello World... from thread = 2
akbar@ubuntu:~/Desktop$ ./hello
Hello World... from thread = 0
Hello World... from thread = 4
Hello World... from thread = 3
Hello World... from thread = 2
Hello World... from thread = 1
akbar@ubuntu:~/Desktop$
```

Hello World (4)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 int main(int argc, char* argv[])
6 {
7     int numThreads = atoi(argv[1]);
8
9     // Início da região paralela
10    #pragma omp parallel num_threads(numThreads)
11    {
12        int my_rank = omp_get_thread_num();
13        int num_threads = omp_get_num_threads();
14        printf("Hello World... from thread = %d of %d\n", my_rank, num_threads);
15    }
16    // Fim da região paralela
17 }
```

#Threads

GOIABINHA



Importante (1)

- Pode haver limitações definidas pelo sistema quanto ao número de threads que um programa pode iniciar.
- O padrão OpenMP não garante que isso realmente iniciará threads de `thread_count`.
- A maioria dos sistemas atuais pode iniciar centenas ou até milhares de threads.



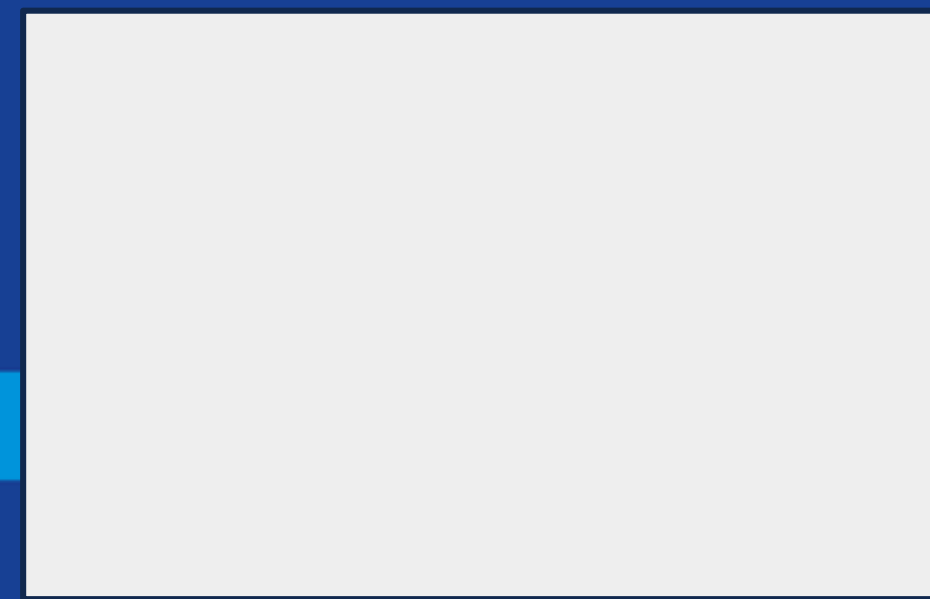
Importante (2)

No jargão OpenMP

- **Mestre:** a threads original
- **Escravos:** as threads adicionais
- **Time:** Mestre e escravos em execução



Escopo das variáveis



Escopo das variáveis

- Refere-se ao conjunto de threads que podem acessar ou não as variáveis na região paralela
- Podem ser:
 - **private**
 - **firstprivate**
 - **lastprivate**
 - **shared**
- O escopo padrão das variáveis é **shared**



Private

- As variáveis NÃO são inicializadas, i.e, recebem um valor inicial aleatório.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (void)
5  {
6      int i = 10;
7
8      #pragma omp parallel private(i)
9      {
10         printf("thread %d: i = %d\n", omp_get_thread_num(), i);
11         i = 1000 + omp_get_thread_num();
12     }
13
14     printf("i = %d\n", i);
15
16     return 0;
17 }
```

```
thread 0: i = 0
thread 3: i = 32717
thread 1: i = 32717
thread 2: i = 1
i = 10
```

(another run of the same program)

```
thread 2: i = 1
thread 1: i = 1
thread 0: i = 0
thread 3: i = 32657
i = 10
```

Firstprivate

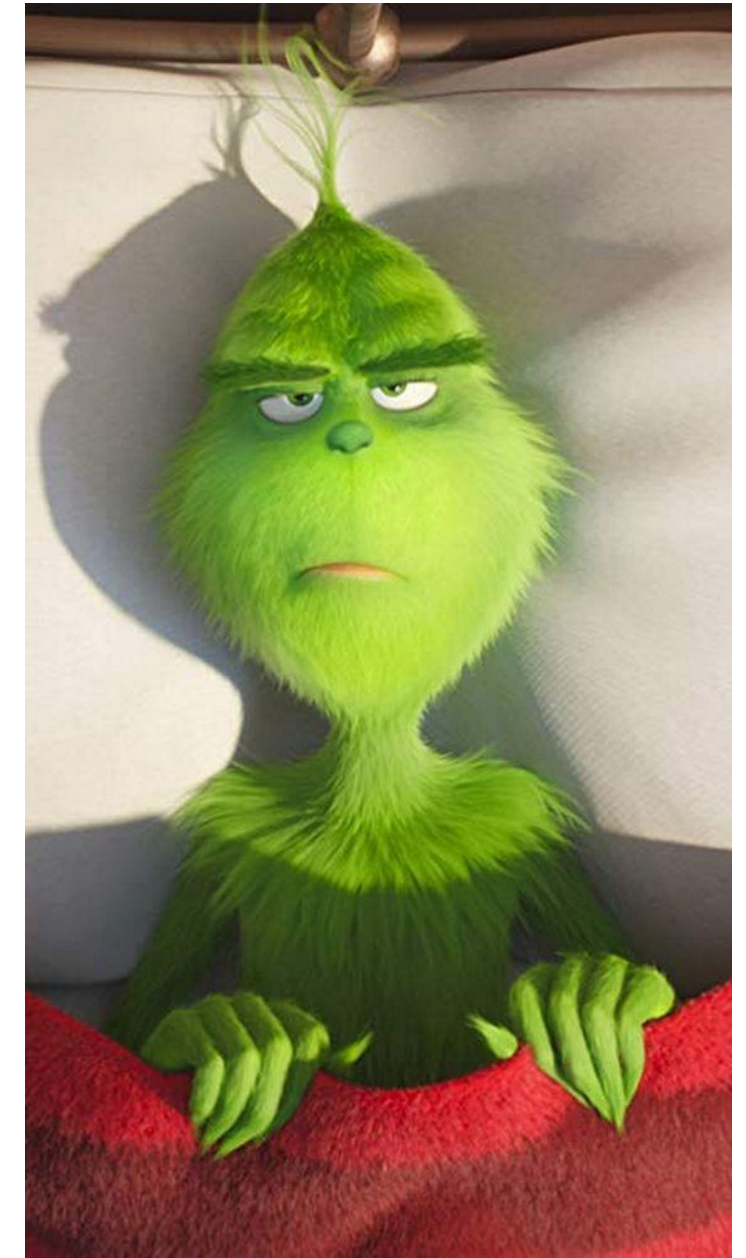
- As variáveis são inicializadas com o valor ANTES da região paralela.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (void)
5  {
6      int i = 10;
7
8      #pragma omp parallel firstprivate(i)
9      {
10         printf("thread %d: i = %d\n", omp_get_thread_num(), i);
11         i = 1000 + omp_get_thread_num();
12     }
13
14     printf("i = %d\n", i);
15
16     return 0;
17 }
```

```
thread 2: i = 10
thread 0: i = 10
thread 3: i = 10
thread 1: i = 10
i = 10
```

Lastprivate

- Utilizado em OMP for, section.
- Retorna o valor da última iteração do laço ou da última execução da seção
- **Veremos em momento oportuno**



Shared

- As variáveis são compartilhadas entre as threads

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <omp.h>
4
5  int main (void)
6  {
7      int i = 10;
8
9      #pragma omp parallel shared(i)
10     {
11         printf("thread %d: i = %d\n", omp_get_thread_num(), i);
12         if ( omp_get_thread_num == 0 ){
13             sleep(10);
14             i=32717;
15         }
16     }
17
18     printf("i = %d\n", i);
19
20     return 0;
21 }
```

```
thread 2: i = 10
thread 0: i = 10
thread 3: i = 10
thread 1: i = 10
i = 32717
```

Default (ESPECIAL) (1)

- Define qual o escopo das variáveis sem declaração explícita do seu escopo, ie., o escopo padrão.
- Pode ser **shared** ou **none**:
 - **Shared**: Todas as variáveis serão shared
 - **None**: As variáveis devem ter escopo explícito



Default (ESPECIAL) (2)

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <omp.h>
4
5  int main (void)
6  {
7      int i = 10;
8
9      #pragma omp parallel default(none) //shared(i) não existe
10     {
11         printf("thread %d: i = %d\n", omp_get_thread_num(), i);
12     }
13
14     printf("i = %d\n", i);
15
16     return 0;
17 }
```



Default (ESPECIAL) (3)

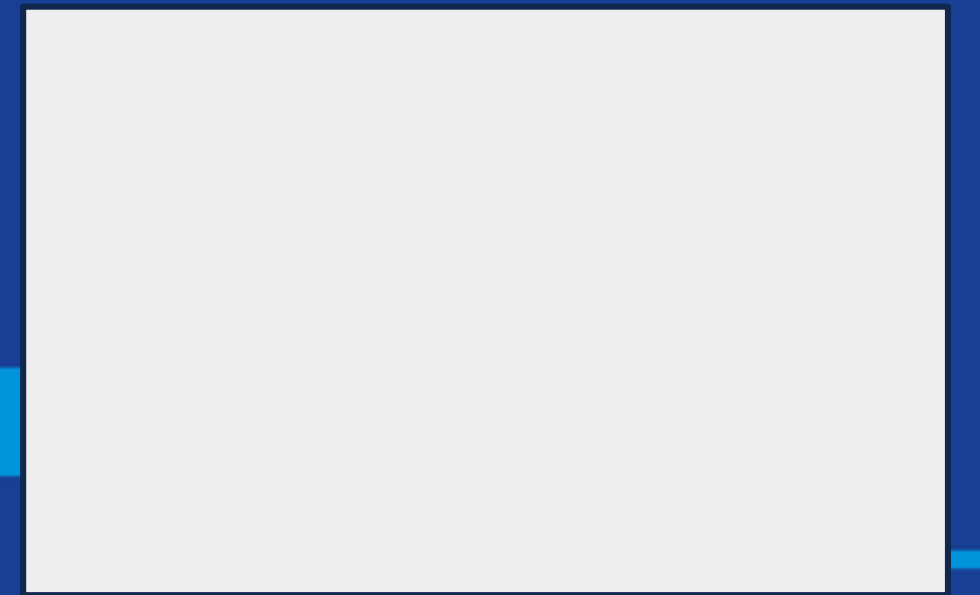
```
kayo@kayo-VirtualBox: ~/Downloads
kayo@kayo-VirtualBox:~/Downloads$ g++ -o default default.cpp -fopenmp
default.cpp: In function 'int main()':
default.cpp:10:15: error: 'i' not specified in enclosing 'parallel'
   10 |         printf("thread %d: i = %d\n", omp_get_thread_num(), i);
      |         ~~~~~^~~~~~
default.cpp:8:13: error: enclosing 'parallel'
   8 |         #pragma omp parallel default (none)
      |         ~~~
kayo@kayo-VirtualBox:~/Downloads$
```

Erro de compilação

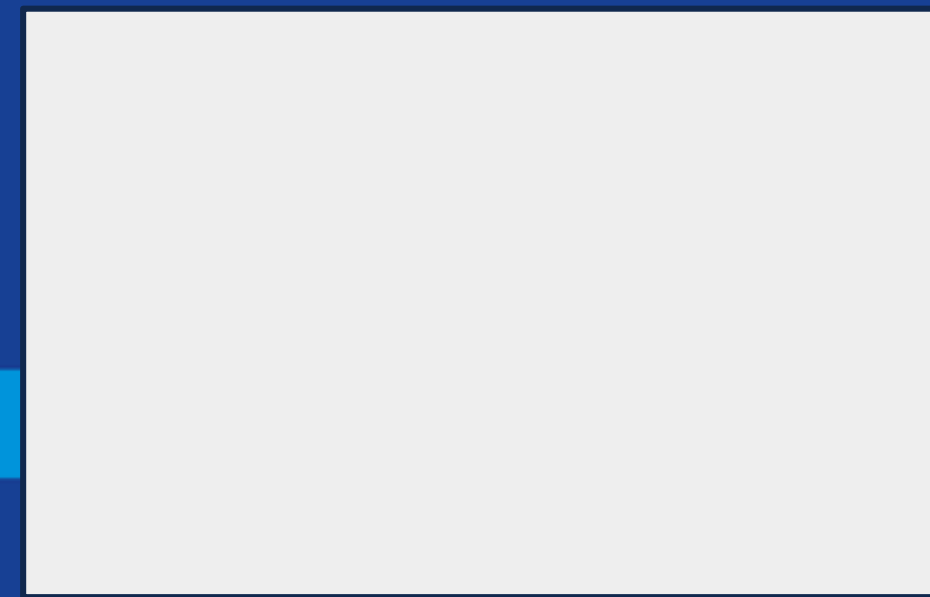
Extremamente Útil

Sempre utilize a cláusula default(*none*)

Dica de ouro, platina, diamante, mestre, grão-mestre, desafiante

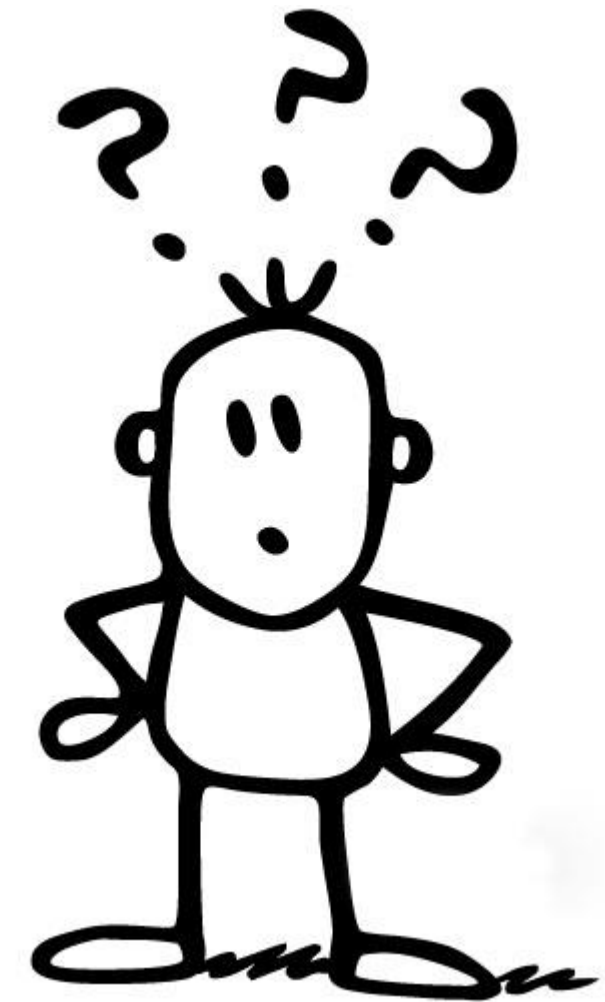


Redução



Redução (1)

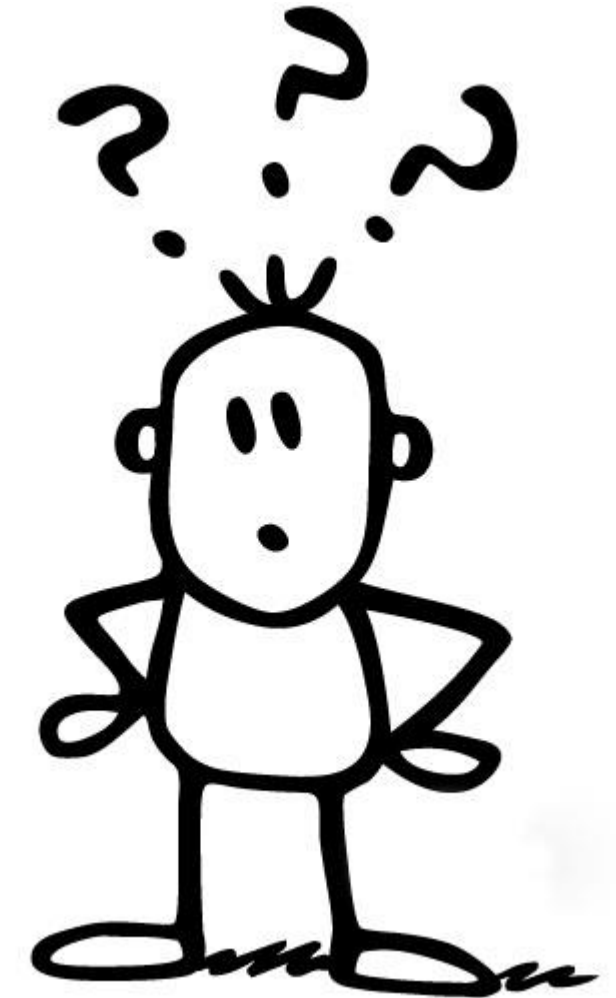
- É uma computação que se aplica repetidamente o mesmo operador de redução a uma sequência de operandos para obter um resultado único.
- Todos os resultados intermediários da operação devem ser armazenados na mesma variável: a variável de redução.
- Pode ser aplicado no escopo da região paralela (**OMP PARALLEL**) ou no **PARALLEL FOR** (estudaremos).



Redução (2)

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

→ + - * & | ^ &&



- É criado uma **cópia local da variável** em cada thread
- A **variável** é inicializado com **ZERO** se número

Exemplo 1 (1)

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (void)
5  {
6      int i = 0;
7
8      #pragma omp parallel default(none) reduction(+:i) num_threads(4)
9      {
10         printf("thread %d: i = %d\n", omp_get_thread_num(), i);
11         i = 1000;
12     }
13
14     printf("i = %d\n", i);
15
16     return 0;
17 }
```

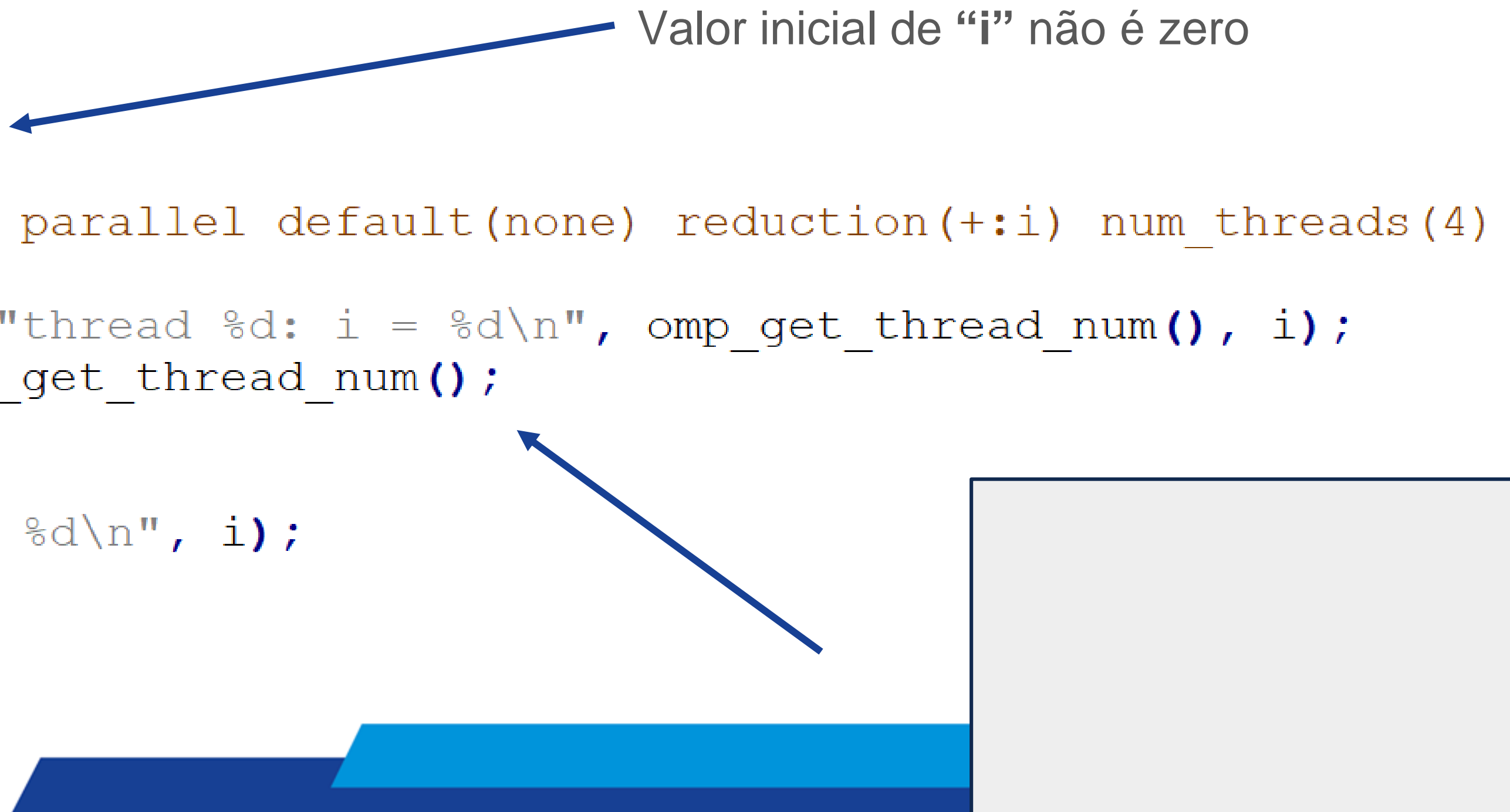
Exemplo 1 (2)

```
kayo@kayo-VirtualBox: ~/Downloads
kayo@kayo-VirtualBox:~/Downloads$ g++ -o default default.cpp -fopenmp
kayo@kayo-VirtualBox:~/Downloads$ ./default
thread 1: i = 0
thread 0: i = 0
thread 3: i = 0
thread 2: i = 0
i = 4000
kayo@kayo-VirtualBox:~/Downloads$
```


Exemplo 2 - CUIDADO (1)

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (void)
5  {
6      int i = 30;
7
8      #pragma omp parallel default(none) reduction(+:i) num_threads(4)
9      {
10         printf("thread %d: i = %d\n", omp_get_thread_num(), i);
11         i = omp_get_thread_num();
12     }
13
14     printf("i = %d\n", i);
15
16     return 0;
17 }
```

Valor inicial de "i" não é zero



Exemplo 2 - CUIDADO (2)

```
kayo@kayo-VirtualBox: ~/Downloads
kayo@kayo-VirtualBox:~/Downloads$ g++ -o default default.cpp -fopenmp
kayo@kayo-VirtualBox:~/Downloads$ ./default
thread 0: i = 0
thread 3: i = 0
thread 2: i = 0
thread 1: i = 0
i = 36
kayo@kayo-VirtualBox:~/Downloads$
```

$30+0+1+2+3$

Valor inicial antes da região paralela

Número das threads

***Garanta o valor inicial correto
da variável de redução antes
de realizar a redução***

Dica de ouro, platina, diamante, mestre, grão-mestre, desafiante

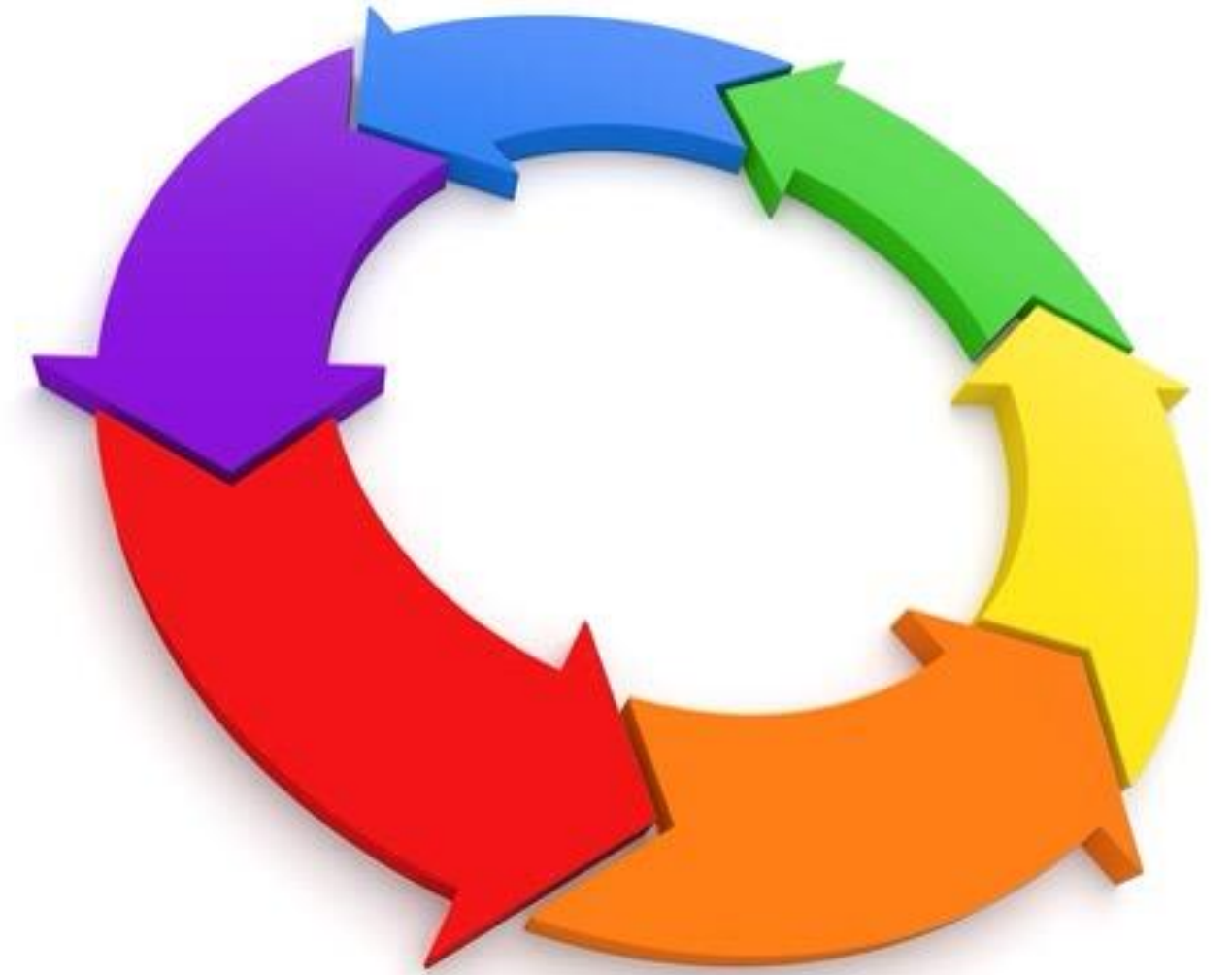


The image features a solid dark blue background. In the center, the text "OMP FOR" is displayed in a large, bold, white sans-serif font. Surrounding the text are several light blue geometric elements: horizontal bars with angled ends at the top and bottom, and a white square with a thin dark blue border in the bottom right corner.

OMP FOR

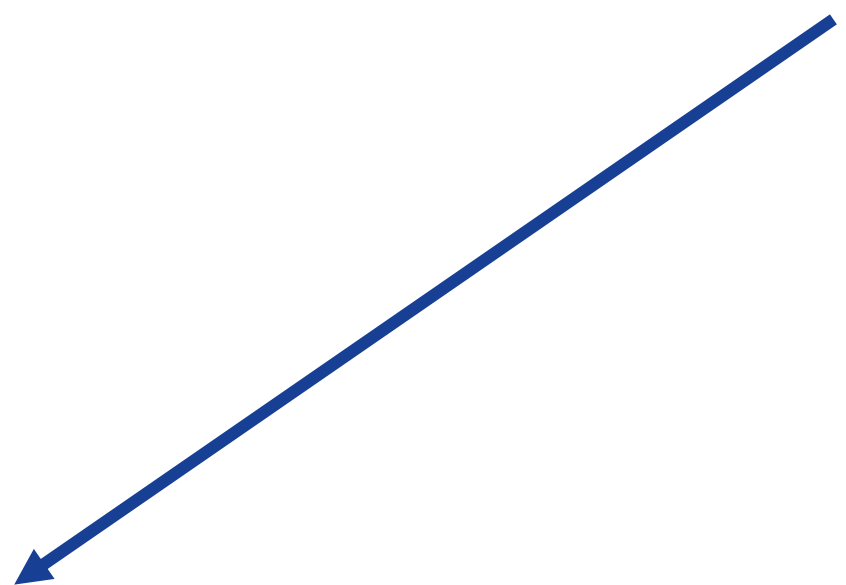
OMP FOR

- Divide o número de iterações entre as threads
- OpenMP não verifica dependências
- Um loop que os resultados dependem de um ou mais resultados de outras iterações não pode, em geral, ser corretamente paralelizado em OpenMP



Exemplo 1 (1)

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (void)
5  {
6      int i;
7      int x = 44;
8
9      #pragma omp parallel for shared(i) \
10     firstprivate(x) default(none) num_threads(10)
11     for(i=0; i<10; i++) {
12         x = i;
13         printf("Thread number: %d x: iter %d\n", omp_get_thread_num(), x);
14     }
15
16     printf("x is %d\n", x);
17
18     return 0;
19 }
```



parallel for

Exemplo 1 (2)

```
kayo@kayo-VirtualBox: ~/Downloads
kayo@kayo-VirtualBox:~/Downloads$ g++ -o for1 for1.cpp -fopenmp
kayo@kayo-VirtualBox:~/Downloads$ ./for1
Thread number: 1 x: iter 1
Thread number: 0 x: iter 0
Thread number: 7 x: iter 7
Thread number: 8 x: iter 8
Thread number: 6 x: iter 6
Thread number: 9 x: iter 9
Thread number: 5 x: iter 5
Thread number: 4 x: iter 4
Thread number: 3 x: iter 3
Thread number: 2 x: iter 2
x is 44
kayo@kayo-VirtualBox:~/Downloads$
```


Exemplo 2 (1)

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (void)
5  {
6      int i;
7      int x = 44;
8
9      #pragma omp parallel shared(i) \
10 firstprivate(x) default(none) num_threads(10)
11  {
12      #pragma omp for
13      for(i=0; i<10; i++) {
14          x = i;
15          printf("Thread number: %d x: iter %d\n", omp_get_thread_num(), x);
16      }
17  }
18  printf("x is %d\n", x);
19  return 0;
20 }
```

omp for dentro de omp parallel


Exemplo 2 (2)

```
kayo@kayo-VirtualBox: ~/Downloads
kayo@kayo-VirtualBox:~/Downloads$ g++ -o for2 for2.cpp -fopenmp
kayo@kayo-VirtualBox:~/Downloads$ ./for2
Thread number: 1 x: iter 1
Thread number: 0 x: iter 0
Thread number: 7 x: iter 7
Thread number: 8 x: iter 8
Thread number: 9 x: iter 9
Thread number: 6 x: iter 6
Thread number: 5 x: iter 5
Thread number: 4 x: iter 4
Thread number: 3 x: iter 3
Thread number: 2 x: iter 2
x is 44
kayo@kayo-VirtualBox:~/Downloads$
```

Exemplo 3 (1)

Escopo lastprivate

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (void)
5  {
6      int i;
7      int x = 44;
8
9      #pragma omp parallel for shared(i) \
10     lastprivate(x) default(none) num_threads(10)
11     for(i=0; i<10; i++) {
12         x = i;
13         printf("Thread number: %d x: iter %d\n", omp_get_thread_num(), x);
14     }
15
16     printf("x is %d\n", x);
17
18     return 0;
19 }
```



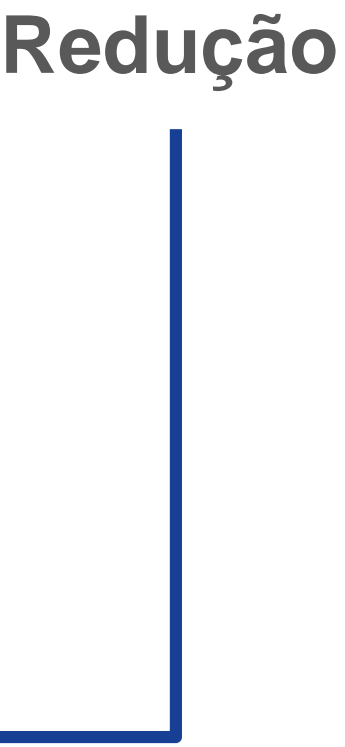
Exemplo 3 (2)

```
kayo@kayo-VirtualBox: ~/Downloads
kayo@kayo-VirtualBox:~/Downloads$ g++ -o for3 for3.cpp -fopenmp
kayo@kayo-VirtualBox:~/Downloads$ ./for3
Thread number: 1 x: iter 1
Thread number: 0 x: iter 0
Thread number: 7 x: iter 7
Thread number: 8 x: iter 8
Thread number: 6 x: iter 6
Thread number: 9 x: iter 9
Thread number: 5 x: iter 5
Thread number: 4 x: iter 4
Thread number: 3 x: iter 3
Thread number: 2 x: iter 2
x is 9
kayo@kayo-VirtualBox:~/Downloads$
```

Exemplo 4 (1)

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (void)
5  {
6      int i;
7      int x = 0;
8
9      #pragma omp parallel for shared(i) \
10     default(none) num_threads(10) reduction(+:x)
11     for(i=0; i<10; i++) {
12         x = i;
13         printf("Thread number: %d x: iter %d\n", omp_get_thread_num(), x);
14     }
15
16     printf("x is %d\n", x);
17
18     return 0;
19 }
```

Redução

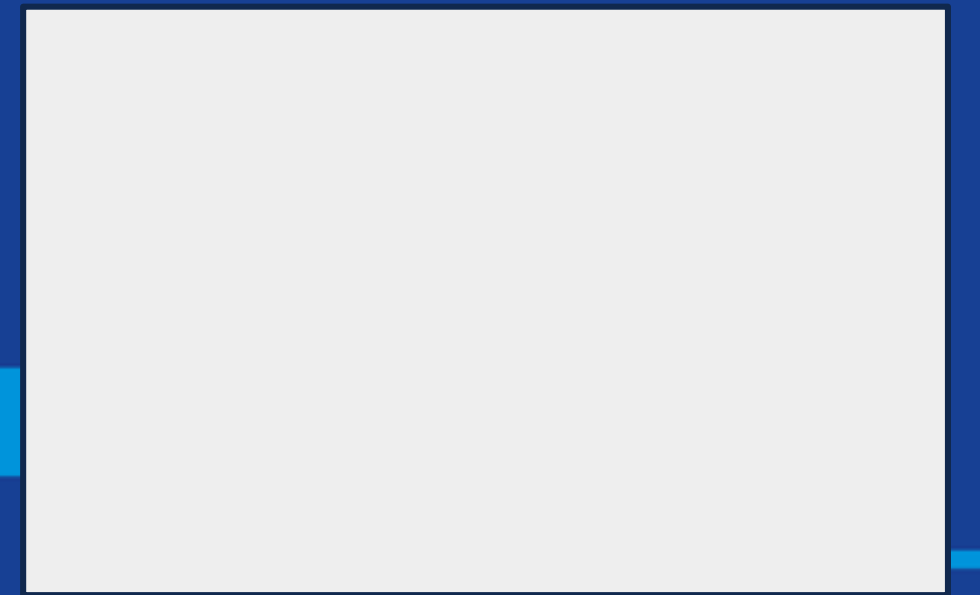


Exemplo 4 (2)

```
kayo@kayo-VirtualBox: ~/Downloads
kayo@kayo-VirtualBox:~/Downloads$ g++ -o for4 for4.cpp -fopenmp
kayo@kayo-VirtualBox:~/Downloads$ ./for4
Thread number: 1 x: iter 1
Thread number: 0 x: iter 0
Thread number: 7 x: iter 7
Thread number: 8 x: iter 8
Thread number: 6 x: iter 6
Thread number: 9 x: iter 9
Thread number: 5 x: iter 5
Thread number: 4 x: iter 4
Thread number: 3 x: iter 3
Thread number: 2 x: iter 2
x is 45
kayo@kayo-VirtualBox:~/Downloads$
```

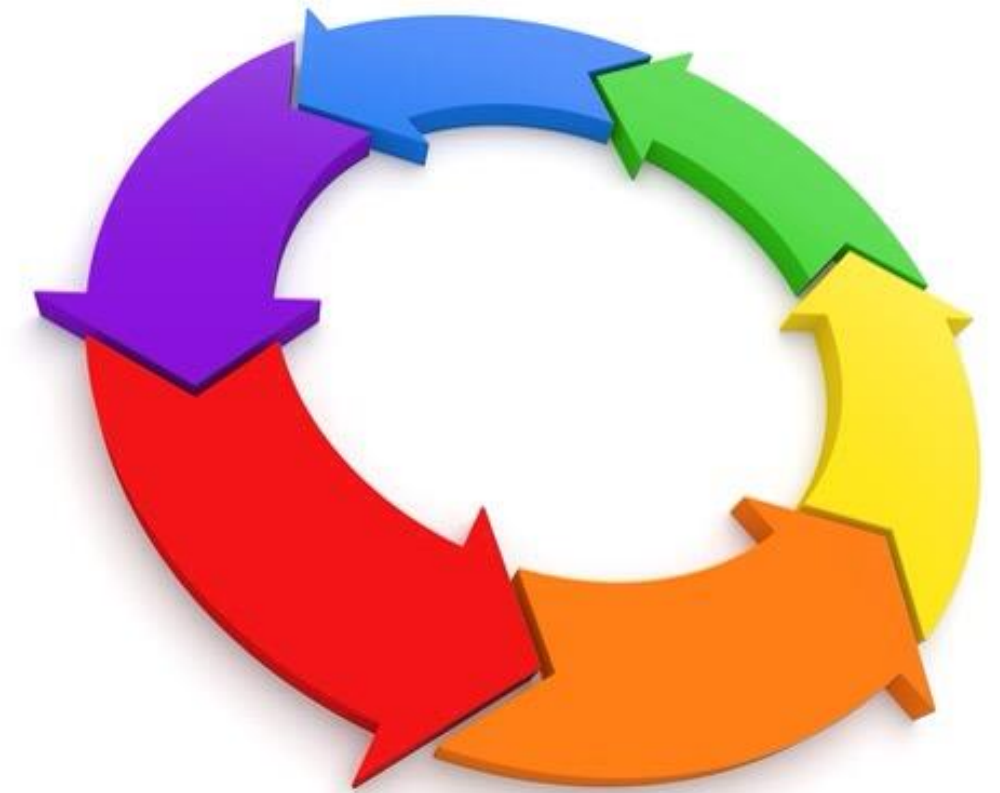
0+1+2+3+...+9

Escalonamento do FOR



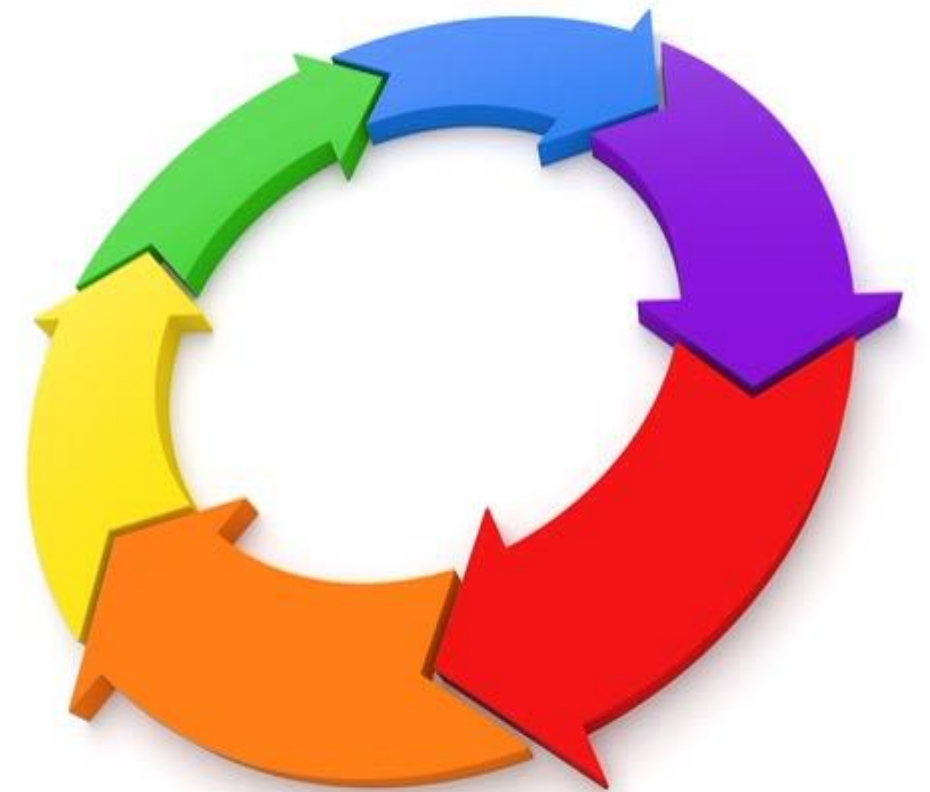
Escalonamento do FOR (1)

- **Static:** as iterações são atribuídas às threads ANTES do loop ser executado.
- **Dynamic** ou **Guided:** as iterações são atribuídas durante a execução do loop



Escalonamento do FOR (2)

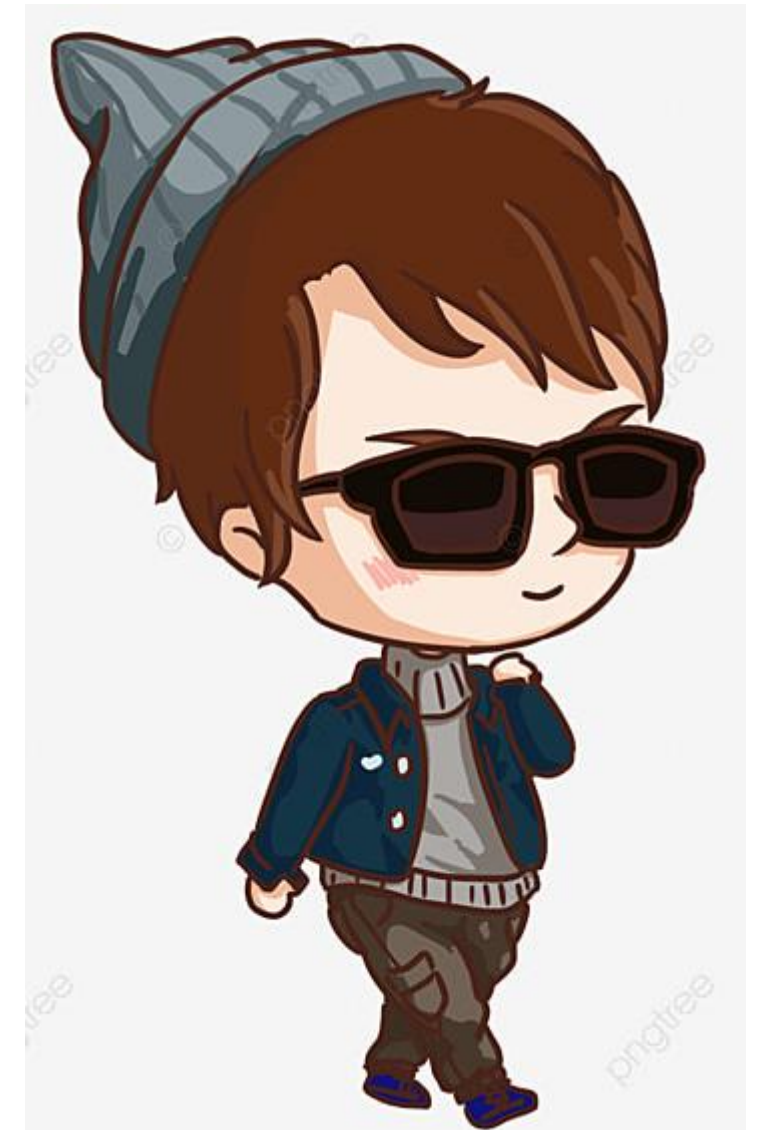
- **Auto:** delega a decisão do escalonamento ao compilador ou sistema de tempo de execução (runtime)
- **Runtime:** determinada em tempo de execução.



Static (1)

- Cada thread receberá um número de iterações igual a **chunksize**.

schedule (static, 1) 12 iterações, 3 threads	
#thread	Iterações
0	0, 3, 6, 9
1	1, 4, 7, 10
2	2, 5, 8, 11



Static (2)

- Cada thread receberá um número de iterações igual a **chunksize**.

schedule (static, 4) 12 iterações, 3 threads	
#thread	Iterações
0	0, 1, 2, 3
1	4, 5, 6, 7
2	8, 9, 10, 11



Exemplo 1 (1)

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (void)
5  {
6      int i;
7
8      #pragma omp parallel for shared(i) \
9      default(none) num_threads(3) schedule(static,1)
10     for(i=0; i<12; i++) {
11         printf("Thread number: %d x: iter %d\n", omp_get_thread_num(), i);
12     }
13
14     return 0;
15 }
```

schedule (static, 1) 12 iterações, 3 threads	
#thread	Iterações
0	0, 3, 6, 9
1	1, 4, 7, 10
2	2, 5, 8, 11

static, 1

Exemplo 1 (2)

```
kayo@kayo-VirtualBox: ~/Downloads
kayo@kayo-VirtualBox:~/Downloads$ g++ -o for_static1 for_static1.cpp -fopenmp
kayo@kayo-VirtualBox:~/Downloads$ ./for_static1
Thread number: 1 x: iter 1
Thread number: 1 x: iter 4
Thread number: 1 x: iter 7
Thread number: 1 x: iter 10
Thread number: 2 x: iter 2
Thread number: 2 x: iter 5
Thread number: 2 x: iter 8
Thread number: 2 x: iter 11
Thread number: 0 x: iter 0
Thread number: 0 x: iter 3
Thread number: 0 x: iter 6
Thread number: 0 x: iter 9
kayo@kayo-VirtualBox:~/Downloads$
```

Exemplo 2 (1)

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (void)
5  {
6      int i;
7
8      #pragma omp parallel for shared(i) \
9      default(none) num_threads(3) schedule(static,4)
10     for(i=0; i<12; i++) {
11         printf("Thread number: %d x: iter %d\n", omp_get_thread_num(), i);
12     }
13
14     return 0;
15 }
```

schedule (static, 4) 12 iterações, 3 threads	
#thread	Iterações
0	0, 1, 2, 3
1	4, 5, 6, 7
2	8, 9, 10, 11

static, 4

Exemplo 2 (2)

```
kayo@kayo-VirtualBox: ~/Downloads
kayo@kayo-VirtualBox:~/Downloads$ g++ -o for_static2 for_static2.cpp -fopenmp
kayo@kayo-VirtualBox:~/Downloads$ ./for_static2
Thread number: 1 x: iter 4
Thread number: 1 x: iter 5
Thread number: 1 x: iter 6
Thread number: 1 x: iter 7
Thread number: 2 x: iter 8
Thread number: 2 x: iter 9
Thread number: 2 x: iter 10
Thread number: 2 x: iter 11
Thread number: 0 x: iter 0
Thread number: 0 x: iter 1
Thread number: 0 x: iter 2
Thread number: 0 x: iter 3
kayo@kayo-VirtualBox:~/Downloads$
```


Dynamic

- As iterações são quebradas em **chunks** (pedaços) de **chunksize**
- Cada thread executa um **chunk**. Quando a terminar a execução deste **chunk**, a thread solicita outro **chunk** ao sistema de tempo de execução.
- O **chunksize** pode ser omitido. Quando isto ocorre, é utilizado **chunksize** igual a 1



Guided (1)

- Cada thread executa um **chunk**. Quando a terminar a execução deste **chunk**, a thread solicita outro **chunk** ao sistema de tempo de execução.
- Conforme os **chunks** são completados, o seu tamanho é reduzido.



Guided (2)

- Se **chunksize** é omitido, seu valor reduzem até 1
- Se **chunksize** é especificado, os **chunks** reduzem os seus valores até o **chunksize**. A exceção é o último **chunk**, que pode ser menor que **chunksize**.



Guided (3)

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

Iterações de regra do trapézio com **9999 iterações** usando escalonamento **GUIDED (sem chunksize)** e **2 threads**



**Barrier, atomic, critical,
single e section**



Barrier (Barreira)

- As threads que executaram a barreira permanecem bloqueados até que todas as threads tenham executado a barreira

#pragma omp barrier

- Após todas as threads terem alcançado a barreira, elas podem continuar sua execução



Atomic

- Cria uma exclusão mútua que protege uma região crítica que é formada por uma única expressão

```
#pragma omp atomic  
x++;
```

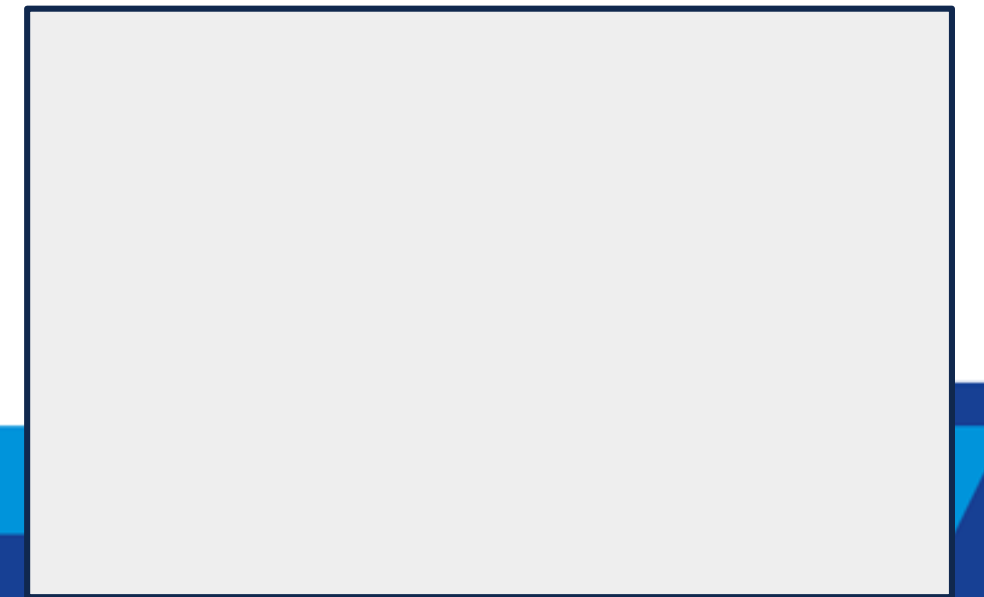
- Pode aceitar operações como: x++, x--, ++x, --x, x+=1, etc
- Aceita operadores como:

+ * - / & ^ | << >>

Critical (1)

- Cria uma exclusão mútua que protege uma região crítica **mais complexa do que atomic**.

```
# pragma omp critical
{
    ...
    global_result += my_result ;
    ...
}
```



Critical (2)

- Um bloco **critical** pode ser nomeada.
- Dois ou mais blocos **critical** com nomes diferentes podem executar simultaneamente



```
# pragma omp critical(bloco1)
{
    ...
    global_result1 += my_result1 ;
    ...
}
```

```
# pragma omp critical(bloco2)
{
    ...
    global_result2 += my_result1 ;
    ...
}
```



Single

- Especifica um bloco que DEVE ser executado por SOMENTE UMA thread.

```
# pragma omp single
{
    ...
    global_result1 += my_result1 ;
    ...
}
```

Barreira implícita

Retira a barreira

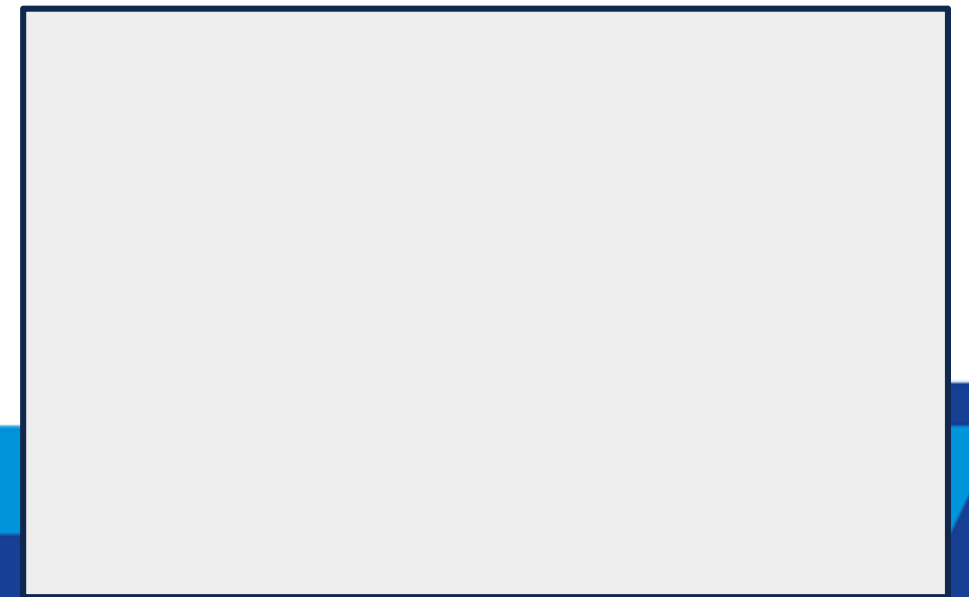
- Cláusulas opcionais: copyprivate, firstprivate, private, **nowait**



Section (1)

- Especifica blocos independentes que DEVEM ser executados por SOMENTE UMA thread.
- Forma simples de paralelizar tarefas distintas que não possuem dependência uma com as outras.

Olha como sou independente



Section (2)

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        a();
    }

    #pragma omp section
    {
        b();
    }

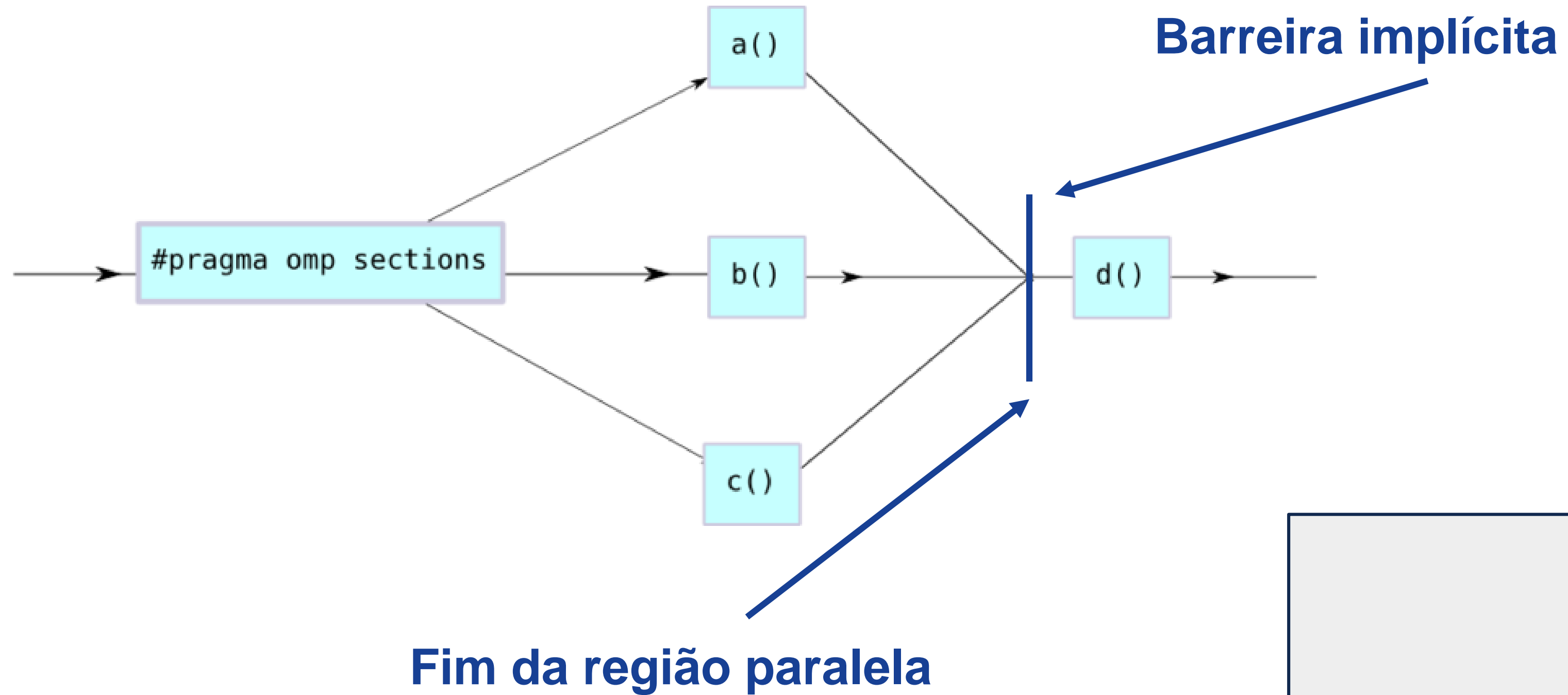
    #pragma omp section
    {
        c();
    }
}
d();
```

Barreira implícita

Fim da região paralela



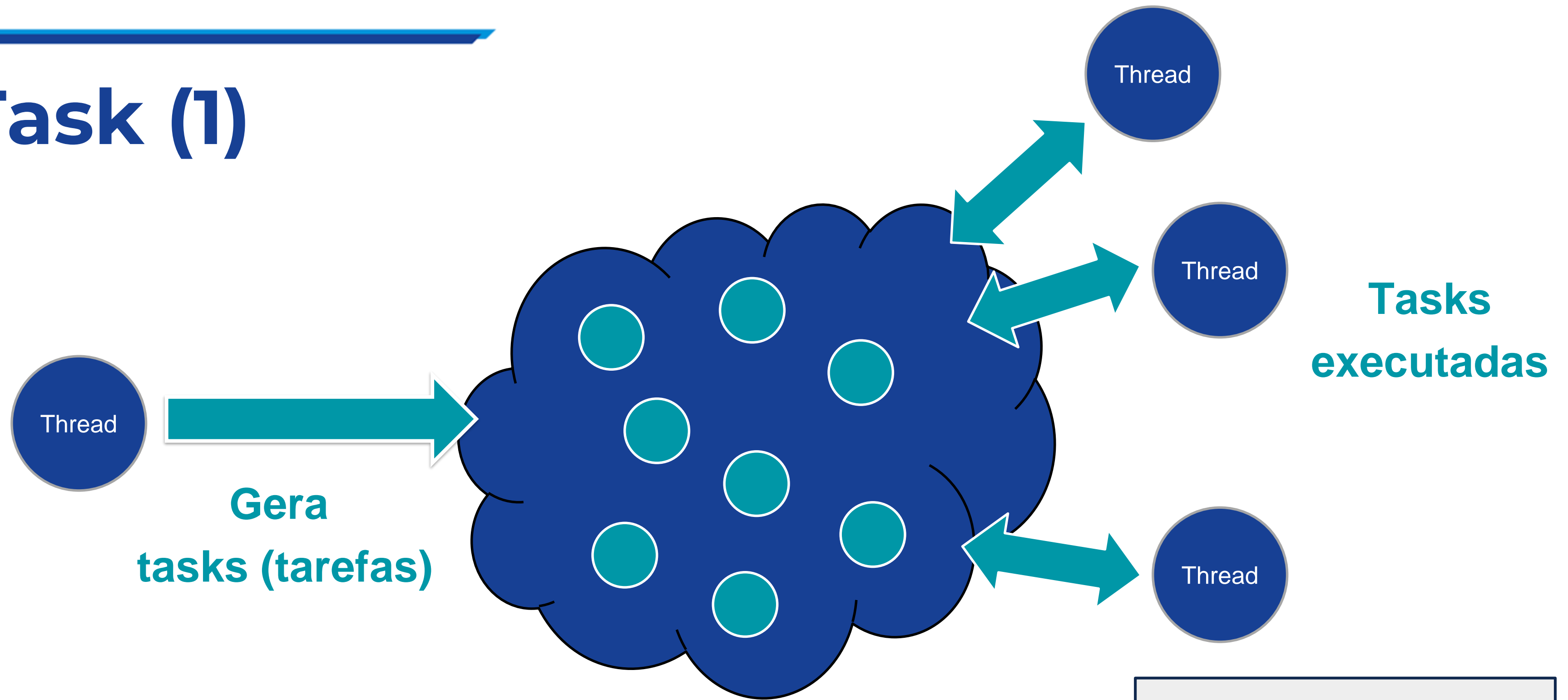
Section (3)



Task

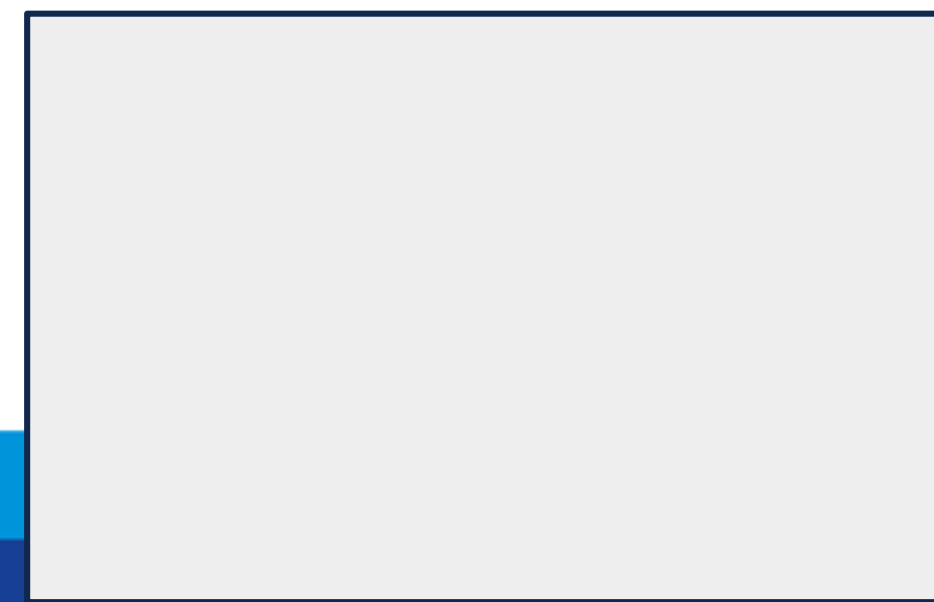


Task (1)



Task (2)

- Quando uma thread encontra um construtor de **task**, uma nova **tarefa** é criada
- O momento de execução de uma **tarefa** fica por conta do sistema de tempo de execução
- Execução de uma **tarefa** pode ser imediata ou atrasada.
- A conclusão de uma **tarefa** pode ser imposta por meio da **sincronização de tarefas**

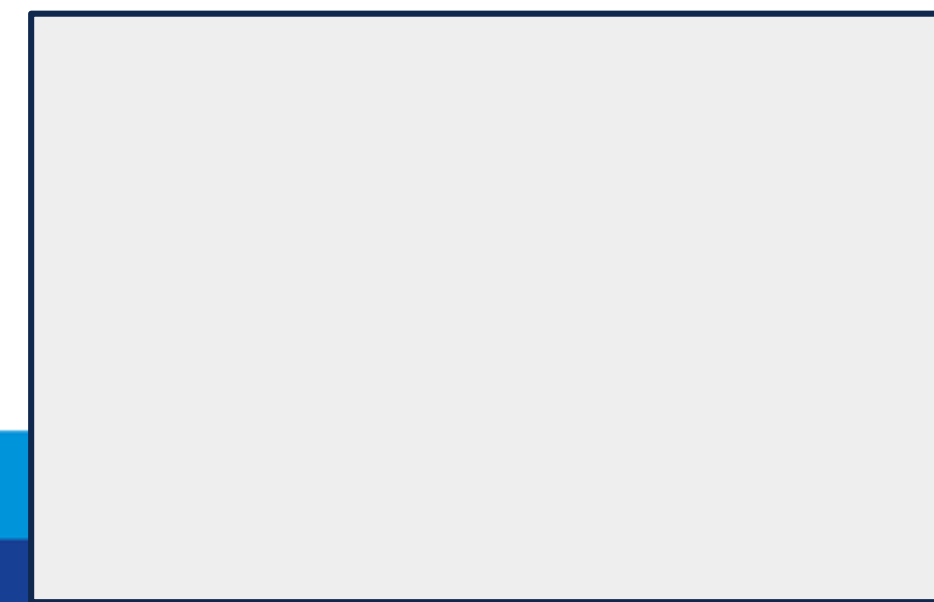


Task (3)

- Sincronização de tarefas

pragma omp taskwait

- Cada **tarefa** tem que ser finalizada
- Não explora o princípio da localidade em memórias (pode reduzir, e muito, o desempenho paralelo)
- Vamos para exemplos incrementais!!!!



Exemplo 1

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

```
int main(int argc, char *argv[]) {
```

```
    printf("A ");
    printf("race ");
    printf("car ");
```

```
    printf("\n");
    return(0);
```

```
}
```

```
$ cc -fast hello.c
$ ./a.out
A race car
$
```

O que o programa vai imprimir?

Exemplo 2 (1)

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

int main(int argc, char *argv[]) {

    #pragma omp parallel
    {
        printf("A ");
        printf("race ");
        printf("car ");

    } // End of parallel region

    printf("\n");
    return(0);
}
```

O que o programa vai imprimir com 2 threads?

Exemplo 2 (2)

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car A race car
```

Note que o programa também poderia imprimir:

“A A race race car car”

“A race A car race car ”

“A race A race car car”

...

Exemplo 3 (1)

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

int main(int argc
```

O que o programa vai
imprimir com 2 threads?

```
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            printf("race ");
            printf("car ");
        }
    } // End of parallel region

    printf("\n");
    return(0);
}
```

Exemplo 3 (2)

```
$ cc -xopenmp -fast hello.c  
$ export OMP_NUM_THREADS=2  
$ ./a.out  
A race car
```

Evidente, agora somente 1 thread
executa o código!

Exemplo 4 (1)

```
int main(int argc, char *argv[]) {  
    #pragma omp parallel  
    {  
        #pragma omp single  
        {  
            printf("A ");  
            #pragma omp task  
            {printf("race ");}  
            #pragma omp task  
            {printf("car ");}  
        }  
    } // End of parallel region  
  
    printf("\n");  
    return(0);  
}
```

O que o programa vai
imprimir com 2 threads?

Exemplo 4 (2)

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
$ ./a.out
A race car
$ ./a.out
A car race
$
```

**As tarefas podem ser
executadas aleatoriamente**

Exemplo 5 (1)

```
int main(int argc, char *argv[]) {  
    #pragma omp parallel  
    {  
        #pragma omp single  
        {  
            printf("A ");  
            #pragma omp task  
            {printf("race ");}  
            #pragma omp task  
            {printf("car ");}  
            printf("is fun to watch ");  
        }  
    } // End of parallel region  
  
    printf("\n");  
    return(0);  
}
```

O que o programa vai imprimir com 2 threads?

Exemplo 5 (2)

```
$ cc -xopenmp -fast hello.c  
$ export OMP_NUM_THREADS=2  
$ ./a.out
```

A is fun to watch race car

```
$ ./a.out
```

A is fun to watch race car

```
$ ./a.out
```

A is fun to watch car race

```
$
```

Exemplo 6 (1)

```
int main(int argc, char
#pragma omp parallel
{
    #pragma omp single
    {
        printf("A ");
        #pragma omp task
        {printf("car ");}
        #pragma omp task
        {printf("race ");}
        #pragma omp taskwait
        printf("is fun to watch ");
    }
} // End of parallel region

printf("\n");return(0);
}
```

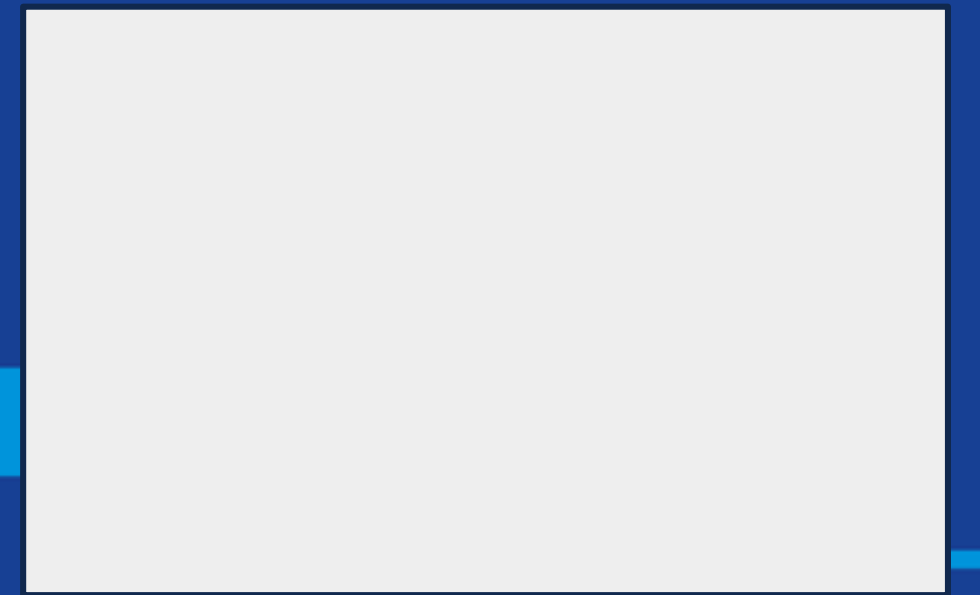
O que o programa vai imprimir com 2 threads?

Exemplo 6 (2)

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
$
A car race is fun to watch
$ ./a.out
A car race is fun to watch
$ ./a.out
A race car is fun to watch
$
```

As tarefas são executadas
primeiro

Exclusão Mútua com Locks



Locks (1)

- Um **lock** consiste de uma estrutura de dados e funções que permitem ao programador impor explicitamente a exclusão mútua em uma região crítica.
- Você tem acesso a **lock** *bloqueante* e *não-bloqueante*.



Locks (2)

```
1  #include <omp.h>
2
3  int main( int argc, const char* argv[] ){
4
5      omp_lock_t my_lock;      //Declara LOCK
6      omp_init_lock(&my_lock); //Inicializa o LOCK
7      int i;
8
9      #pragma omp parallel shared(my_lock, i) \
10 num_threads(10) default(none)
11  {
12      omp_set_lock(&my_lock); //LOCK
13      i++;
14      omp_unset_lock(&my_lock); //UNLOCK
15  }
16
17      omp_destroy_lock(&my_lock); //Destroi LOCK
18
19      return(0);
20 }
```



Lock bloqueante



Locks (3)

- Lock não-bloqueante

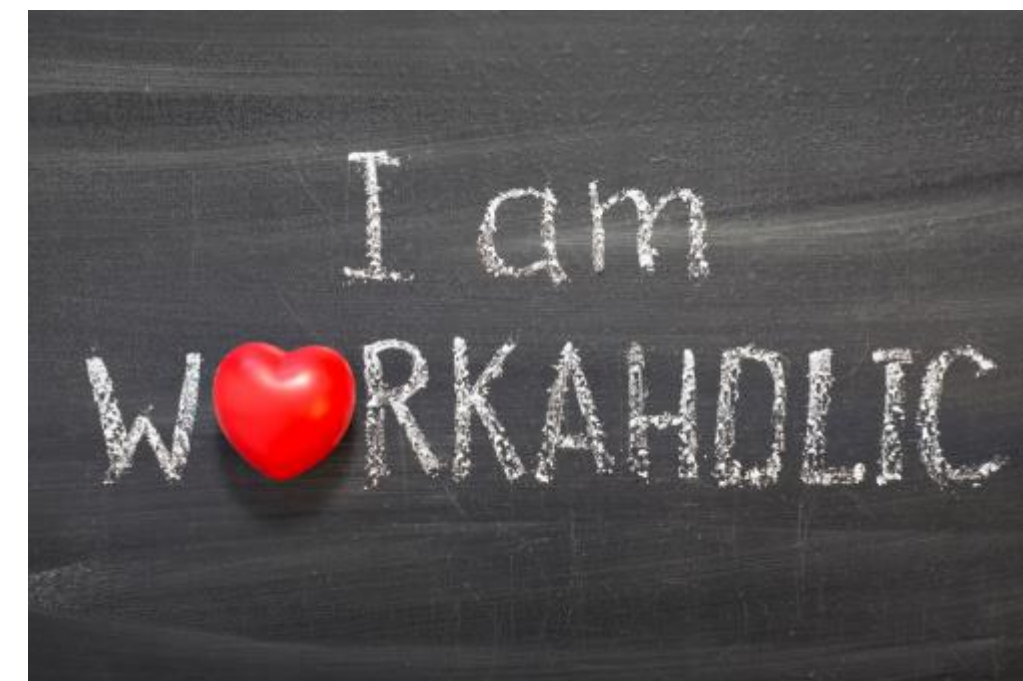
omp_try_lock(&my_lock)

- Tenta Lock. Se bem sucedido, trava a **my_lock** e retorna **true**. Caso contrário, não trava **my_lock** e retorna **false**.
- try_lock não bloqueia a execução da thread.



Locks (4)

```
1  #include <omp.h>
2
3  int main( int argc, const char* argv[] ){
4
5      omp_lock_t my_lock;      //Declara LOCK
6      omp_init_lock(&my_lock); //Inicializa o LOCK
7      int i
8
9      #pragma omp parallel shared(my_lock, i) \
10     num_threads(10) default(none)
11     {
12         if (omp_test_lock(&my_lock)){ //LOCK
13             i++;
14             omp_unset_lock(&my_lock); //UNLOCK
15         }
16         else{
17             printf("Não consegui o Lock")
18         };
19     }
20
21     omp_destroy_lock(&my_lock); //Destrói LOCK
22
23     return(0);
24 }
25
26 }
```



Lock não-bloqueante

