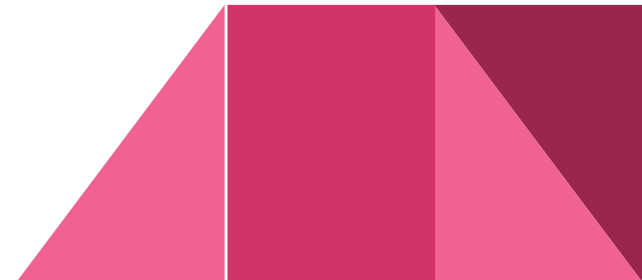


Material Suplementar

# Programação Paralela e Distribuída – Parte 2

**Gabriel P. Silva**



# Programação Paralela e Distribuída

com MPI, OpenMP e OpenACC  
para computação de alto desempenho

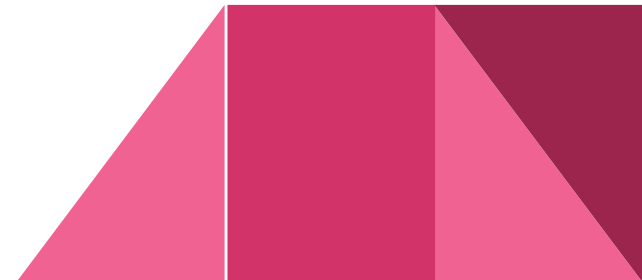


 Casa do Código | **alura**

GABRIEL P. SILVA  
CALEBE P. BIANCHINI  
EVALDO B. COSTA

Os códigos fontes utilizados neste material  
estão disponíveis em:

<https://github.com/Programacao-Paralela-e-Distribuida/OPENMP>



# Breve História do OpenMP

- Existe uma falta histórica de padronização nas diretivas para compartilhamento de memória. Cada fabricante fazia a sua própria.
- O fórum OpenMP foi iniciado pela Digital, IBM, Intel, KAI e SGI. Agora inclui todos os grandes fabricantes.
- O padrão OpenMP para Fortran foi liberado em Outubro de 1997. A versão 2.0 foi liberada em Novembro de 2000.
- O padrão OpenMP C/C++ foi liberado em Outubro de 1998. A versão 2.0 foi liberada em Março de 2002.



# Breve História do OpenMP

- A versão 3.0 C/C++ e Fortran foi liberada em Novembro de 2008.
- A versão 3.1 C/C++ e Fortran foi liberada em Setembro de 2011.
- A versão 4.0 C/C++ e Fortran foi liberada em Outubro de 2013.
- A versão 4.5 C/C++ e Fortran foi liberada em Novembro de 2015.
- A versão 5.0 está em discussão/desenvolvimento pela comunidade.



# Breve História do OpenMP

## **MPI vs. OpenMP**

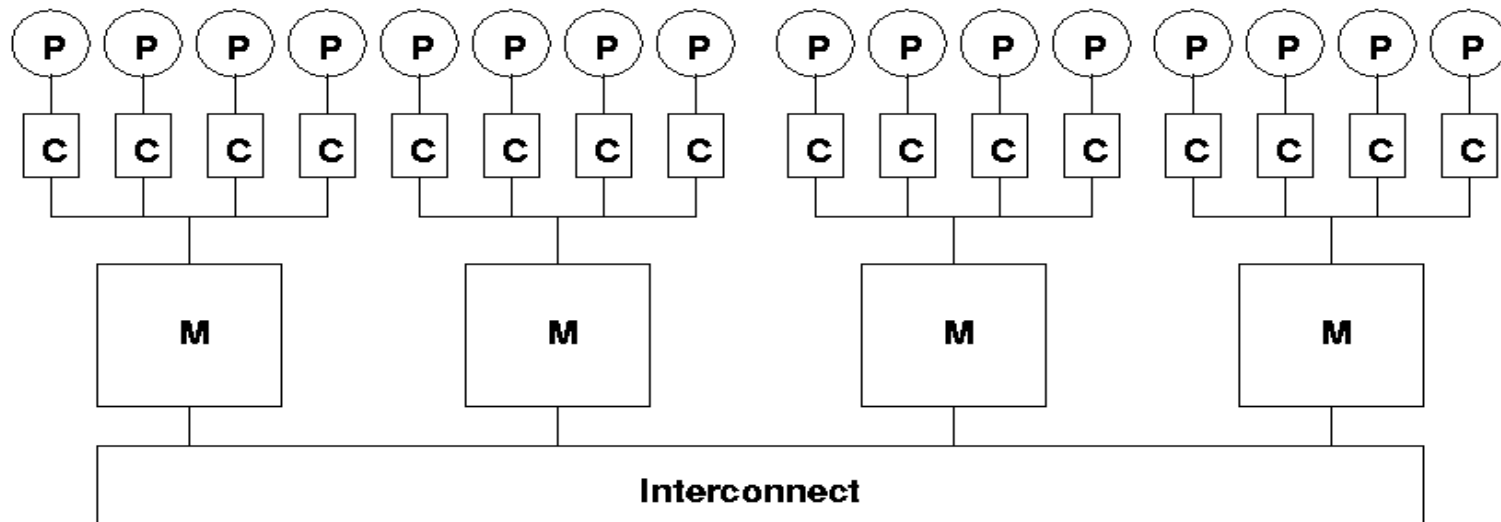
MPI	OpenMP
Distributed memory model	Shared memory model
on Distributed network	on Multi-core processors
Message based	Directive based
Flexible and expressive	Easier to program and debug

# Sistemas de Memória Compartilhada

- O OpenMP foi projetado para a programação de computadores paralelos com memória compartilhada.
- A facilidade principal é a existência de um único espaço de endereçamento através de todo o sistema de memória.
- Cada processador pode ler e escrever em todas as posições de memória.
- Um espaço único de memória
- Dois tipos de arquitetura:
  - Memória Compartilhada Centralizada
  - Memória Compartilhada Distribuída



# Sistemas de Memória Compartilhada Distribuída

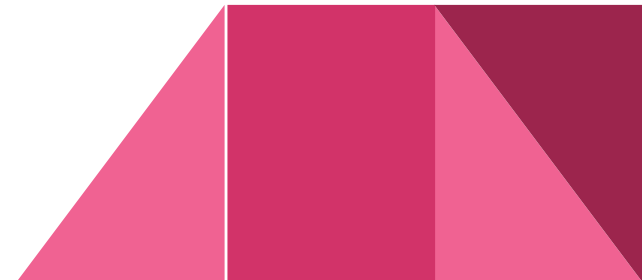


**Clustered distributed shared memory**

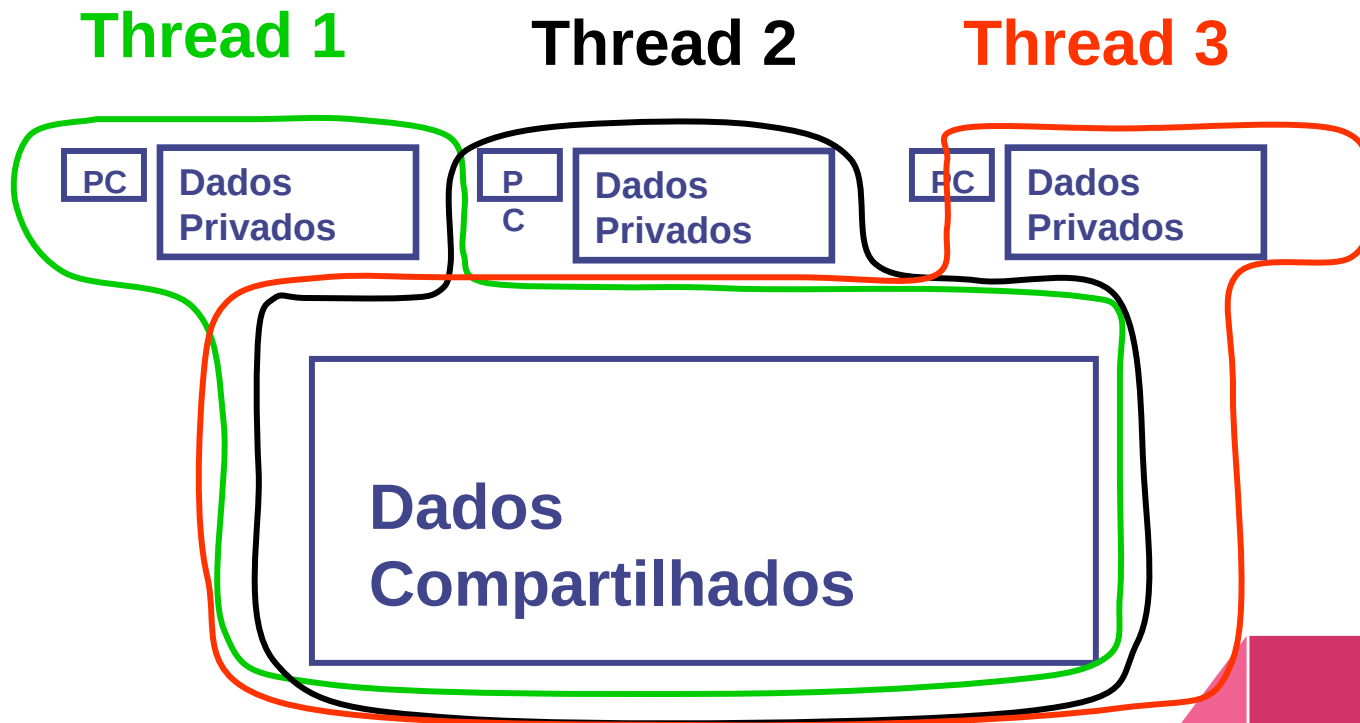


# Threads

- Uma *thread* é um processo “peso leve”.
- Cada *thread* pode ser seu próprio fluxo de controle em um programa.
- As *threads* podem compartilhar dados com outras *threads*, mas também têm dados privados.
- As *threads* se comunicam através de uma área de dados compartilhada.
- Uma equipe de *threads* é um conjunto de *threads* que cooperam em uma tarefa.
- A “*thread master*” é responsável pela coordenação da equipe de *threads*.

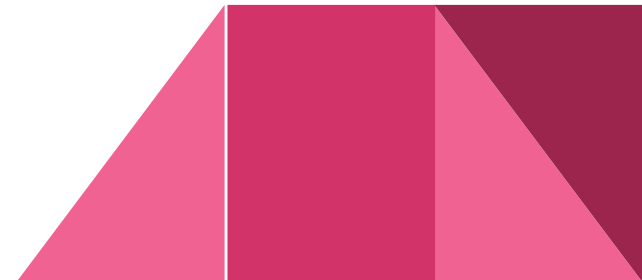


# Threads



# Paralelismo

- O paralelismo no OpenMP é obtido pela execução simultânea de diversas threads dentro do que são chamadas de regiões paralelas.
- Haverá ganho real de desempenho se houver processadores disponíveis na arquitetura para efetivamente executar essas regiões em paralelo.
- As diversas iterações de um laço **for** também podem ser compartilhadas entre as diversas threads e, se não houver dependências de dados entre as iterações do laço, poderão também ser executadas em paralelo.

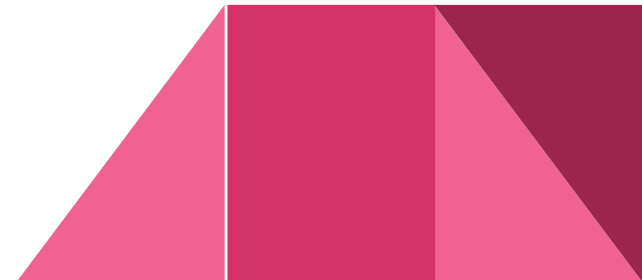


# Laços Paralelos

- Os laços são a principal fonte de paralelismo em muitas aplicações.
- Se as iterações de um laço são independentes (podem ser executadas em qualquer ordem), então podemos compartilhar as iterações entre *threads* diferentes.
- Por exemplo, se tivermos duas *threads* e o laço:

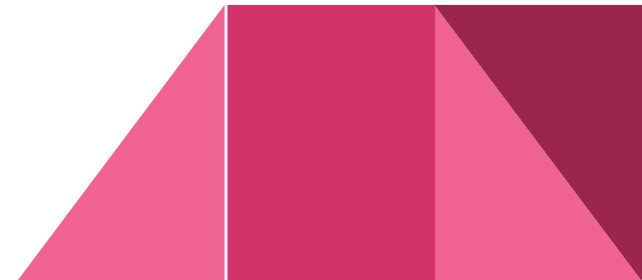
```
for (i = 0; i<100; i++)  
    a[i] = a[i] + b[i];
```

- As iterações 0-49 podem ser feitas em uma *thread* e as iterações 50-99 na outra.

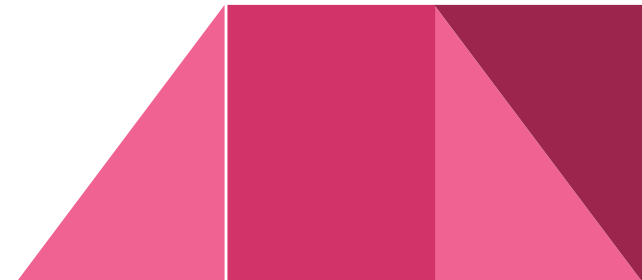


# Diretivas e Sentinelas


- O OpenMP faz uso conjunto de diretivas passadas para o compilador, assim como de algumas funções, para explorar o paralelismo no código em linguagem C ou FORTRAN.
- Uma **diretiva** é uma linha especial de código fonte com significado especial apenas para determinados compiladores.
- Uma diretiva se distingue pela existência de uma **sentinela** no começo da linha.
- As sentinelas do OpenMP são:
  - **C/C++:**  
#pragma omp
  - **Fortran:**  
!\$OMP (ou C\$OMP ou \*\$OMP)



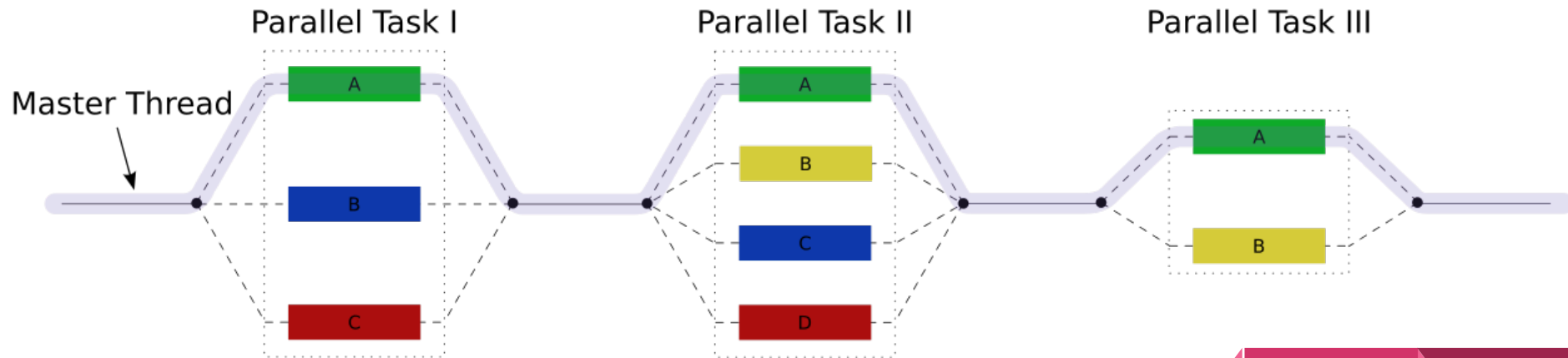
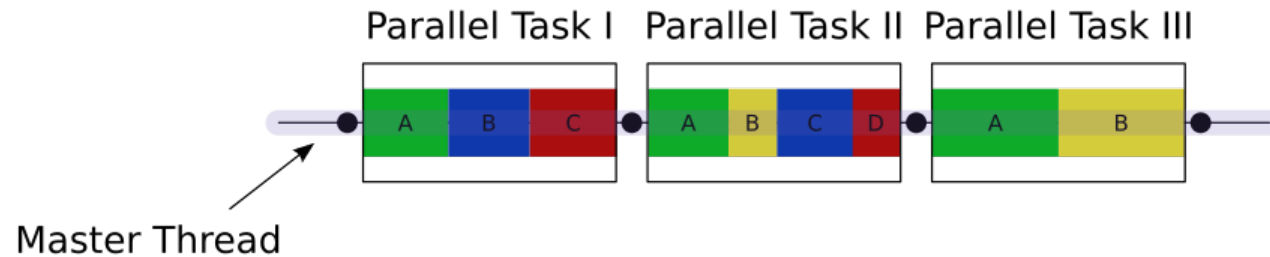
# Regiões Paralelas



# Região Paralela

- A região paralela é a estrutura básica de paralelismo no OpenMP.
  - Uma região paralela define uma seção do programa.
  - Os programas começam a execução com uma única *thread* (a “thread master”).
  - Quando a primeira região paralela é encontrada, a *thread master* cria uma equipe de *threads* (modelo *fork/join*).
  - Cada *thread* executa as sentenças que estão dentro da região paralela.
  - No final da região paralela, a “*thread master*” espera pelo término das outras *threads* e continua então a execução de outras sentenças.
- 

# Região Paralela



Source: <http://en.wikipedia.org/wiki/OpenMP>



# Número de Threads


- Fora do programa, pode ser especificado o número de threads que serão disparadas, com o uso da variável de ambiente **OMP\_NUM\_THREADS**
- Dentro do programa, podem ser utilizadas as seguintes opções:
  - Cláusula **omp\_numthreads(n)**
  - Função **omp\_set\_num\_threads( n)**
- **C/C++:**

```
#include <omp.h>
```

```
void omp_set_num_threads(int num_threads);
```

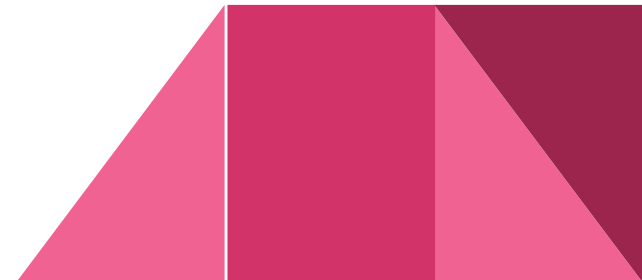


# OMP\_NUM\_THREADS

- **OMP\_NUM\_THREADS** – especifica o número de threads para serem usadas durante a execução de regiões paralelas.
  - O valor padrão para esta variável é 1.
  - **OMP\_NUM\_THREADS** threads serão usadas para executar o programa independente do número de processadores físicos disponíveis no sistema.
  - Como resultado, você pode executar programas com mais threads do que o número de processadores físicos e eles vão executar corretamente. Contudo, o desempenho da execução dos programas nesta maneira poderá ser ineficiente.
- 

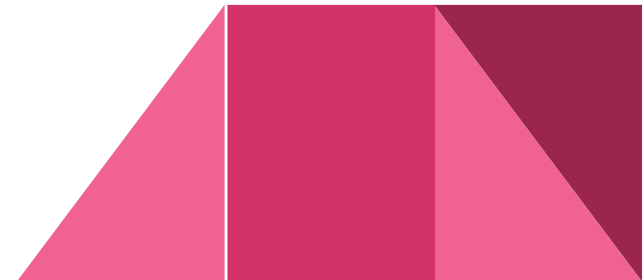
# Precedência

- A variável de ambiente **OMP\_NUM\_THREADS** (se presente) especifica inicialmente o número de threads.
- As chamadas para a função **omp\_set\_num\_threads()** se sobrepõe ao valor especificado em **OMP\_NUM\_THREADS**;
- A presença da cláusula **num\_threads()** em uma região paralela se sobrepõe a todos outros dois valores.



# Número de Threads

- Um das funções mais utilizadas no OpenMP retorna o número de *threads* que estão sendo utilizadas naquela região paralela.
- **C/C++:**  
`#include <omp.h>`  
`int omp_get_num_threads(void);`
- **Nota importante:** retorna 1 se a chamada é fora de uma região paralela.



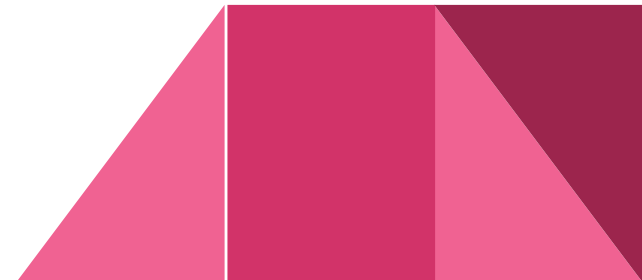
# Identificador da Thread

- Também são utilizadas para encontrar o número atual da *thread* em execução.

## C/C++:

```
#include <omp.h>  
int omp_get_thread_num(void);
```

- Retorna valores entre 0 e `omp_get_num_threads() - 1`



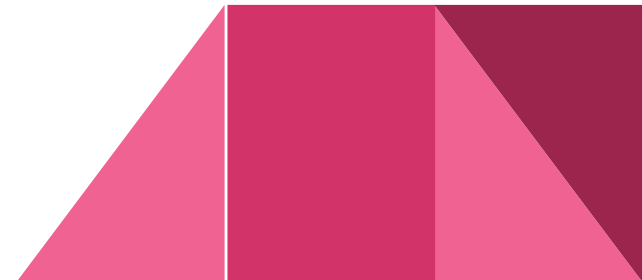
# Região Paralela

- Existe uma função para identificar se a execução atual é dentro de uma região paralela.

## C/C++:

```
#include <omp.h>  
int omp_in_parallel();
```

- Retorna um valor diferente de 0 se a chamada for dentro de uma região paralela

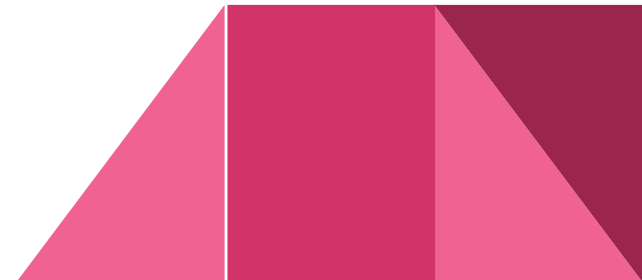


# Diretiva para Regiões Paralelas

- Um código dentro da região paralela é executado por todas as *threads*.
- Sintaxe:

**C/C++:**

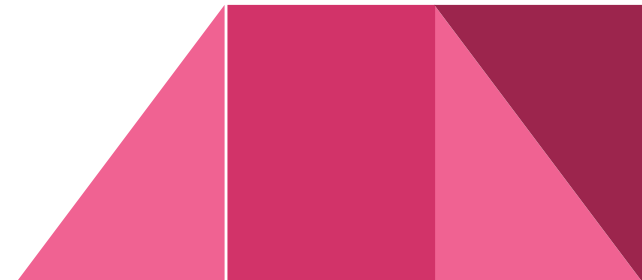
```
#pragma omp parallel  
    {  
        block  
    }
```



# Laços for paralelos

- Laços são a maior fonte de paralelismo na maioria dos códigos. Diretivas paralelas de laços são portanto muito importantes!
- Um laço for paralelo divide as iterações do laço entre as *threads*.
- Apresentaremos aqui apenas a forma básica.
- Sintaxe C/C++:

```
#pragma omp for [clausulas]  
for loop
```

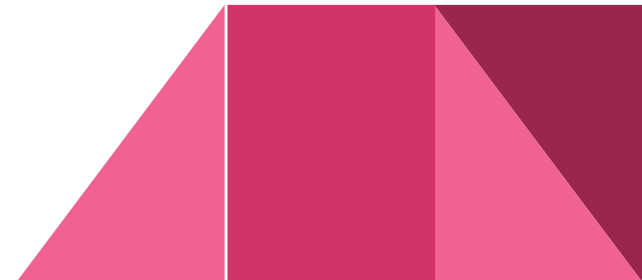




# Laços for paralelos

- Sem cláusulas adicionais, a diretiva **for** usualmente particionará as iterações o mais igualmente possível entre as *threads*.
- Contudo, isto é dependente de implementação e ainda há alguma ambigüidade:

**Ex.: 7 iterações, 3 *threads*. Pode ser particionado como 3+3+1 ou 3+2+2**

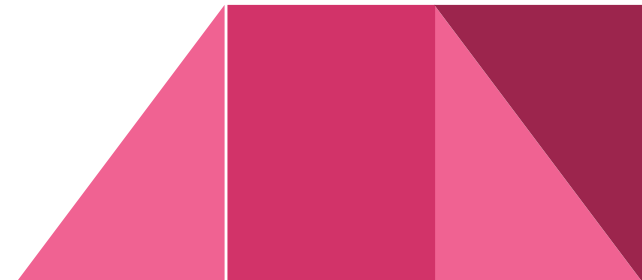


# A diretiva for paralela

- Esta construção é tão comum que existe uma forma que combina a região paralela e a diretiva for:
- Sintaxe C/C++:

```
#pragma omp parallel for [clausulas]  
    for loop
```

- Veremos depois mais detalhes da diretiva for



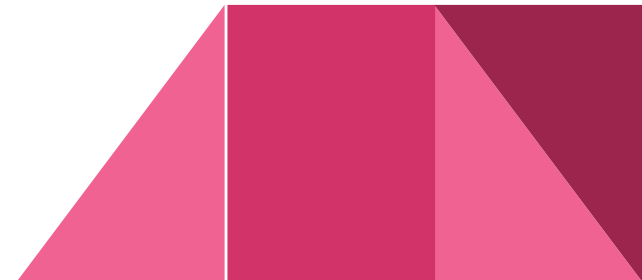
# Cláusulas

- Especificam informação adicional na diretiva de região paralela:

## C/C++:

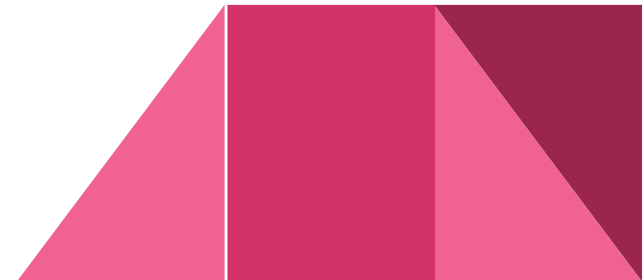
```
#pragma omp parallel [clausulas]  
#pragma omp parallel for [clausulas]
```

- Cláusulas são separadas por vírgula ou espaço no Fortran, e por espaço no C/C++.



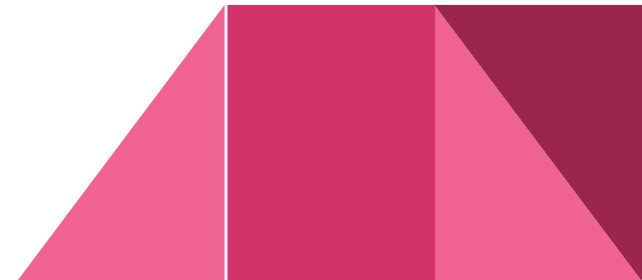
# Cláusulas

- A diretiva **parallel** admite as seguintes cláusulas:
  - **num\_threads**
  - **default, private, shared**
  - **reduction**
  - **firstprivate**
  - **if**
  - **copyin (não abordado)**



# Cláusula num\_threads


- A cláusula **num\_threads** tem a mesma funcionalidade da função **omp\_set\_num\_threads**.
- A cláusula **num\_threads** se aplica às seguintes diretivas:
  - **parallel**
  - **for**
  - **sections (veremos depois)**



# Exemplo - Cláusula num\_threads

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

int main() {
    printf("Fora = %d\n", omp_in_parallel( ));
    #pragma omp parallel num_threads(4)
    {
        int i = omp_get_thread_num();
        printf("Olá da thread %d\n", i);
        printf("Dentro = %d\n",
omp_in_parallel( ));
    }
}
```



# Executando um Programa

- Antes de compilar o programa definir a quantidade de threads que serão utilizadas.

```
# export OMP_NUM_THREADS=4
```

- Compilando o programa.

```
# gcc omp_hello.c -fopenmp -o teste
```

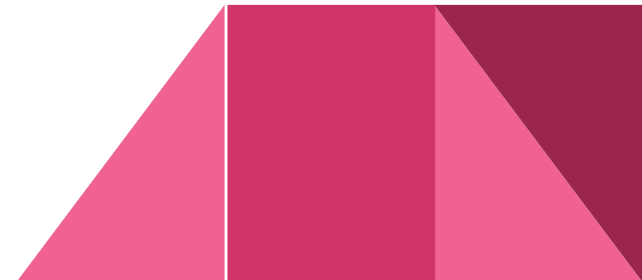
- Executando o programa.

```
# ./teste
```



# Dados Privados e Compartilhados

- Dentro de uma região paralela, as variáveis podem ser privadas ou compartilhadas.
- Todas as *threads* vêm **a mesma cópia** das variáveis compartilhadas.
- Todas as *threads* podem ler ou escrever nas variáveis compartilhadas.
- Cada *thread* tem a **sua própria cópia** de variáveis privadas: essas são invisíveis para as outras *threads*.
- Uma variável privada pode ser lida ou escrita apenas pela sua própria *thread*.





# Cláusulas shared, private e default

- Dentro de uma região paralela as variáveis podem ser **compartilhadas** (todas as *threads* vêm a mesma cópia) ou **privadas** (cada *thread* tem a sua própria cópia).
- Cláusulas **shared**, **private** e **default**

## C/C++:

`shared(list)`

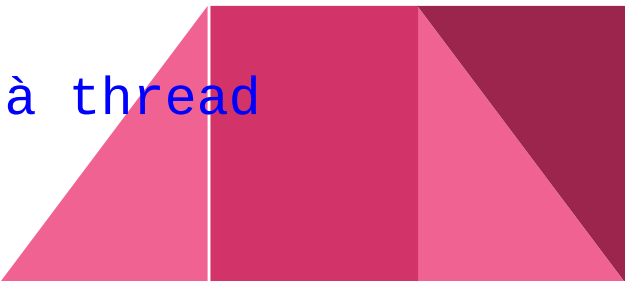
`private(list)`

`default(shared|none)`



# Exemplo - Alô Mundo

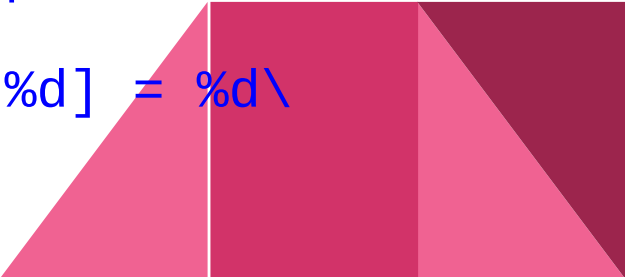
```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
    int nthreads, tid;
    #pragma omp parallel private(nthreads, tid)
    {
        /* Obtém o número da thread */
        tid = omp_get_thread_num();
        printf("Alô mundo da thread = %d\n", tid);
        /* Apenas a thread master faz isto */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Número de threads = %d\n",
nthreads);
        }
    } /* Todas as threads se juntam à thread
master e terminam */
}
```



# Exemplo - Cláusula shared

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


int main(int argc, char *argv[]) {
    int i, n = 7;
    int a[n];
    (void) omp_set_num_threads(4);
    for (i=0; i<n; i++)
        a[i] = i+1;
    #pragma omp parallel for shared(a)
    for (i=0; i<n; i++) {
        a[i] += i;
    } /*-- End of parallel for --*/
    printf("No programa principal depois do\n");
    for (i=0; i<n; i++)    printf("a[%d] = %d\n", i, a[i]);
    return(0);
}
```



# Exemplo - Cláusula private

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

int main(int argc, char *argv[]) {
    int i, n = 5;
    int a;
    (void) omp_set_num_threads(3);
    #pragma omp parallel for private(i,a)
    for (i=0; i<n; i++)
    {
        a = i+1;
        printf("Thread %d tem um valor de a = %d\n",
               omp_get_thread_num(), a, i);
    } /* -- End of parallel for -- */
    return(0);
}
```

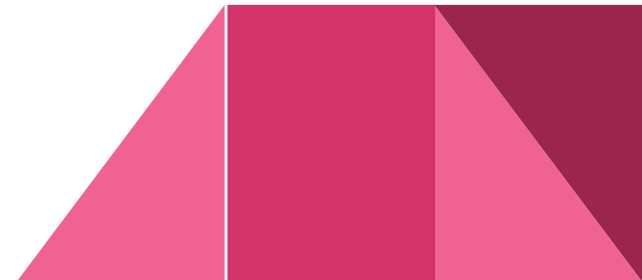


# Reduções

- Uma redução produz um único valor a partir de operações associativas como soma, multiplicação, máximo, mínimo, e , ou. Por exemplo:

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

- Permitindo que apenas uma *thread* por vez atualize a variável **b** removeria todo o paralelismo.
- Ao invés disto, cada *thread* pode acumular sua própria cópia privada, então essas cópias são reduzidas para dar o resultado final.




# Reduções

- Uma redução produz um único valor a partir de operações associativas como adição, multiplicação, máximo, mínimo, e, ou.
- É desejável que cada thread faça a redução em uma cópia privada e então reduzam todas elas para obter o resultado final.
- Uso da cláusula **reduction**:

**C/C++:**

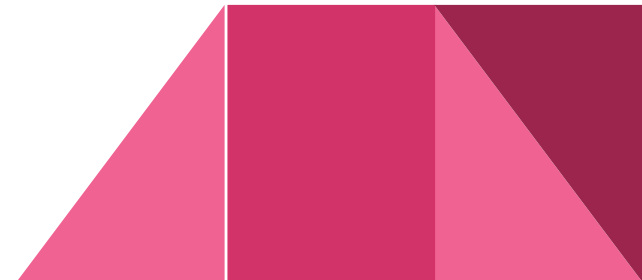
`reduction(op: list)`



# Reduções


- Onde **op** pode ser:

Operação	Valor Inicial
<b>+</b> --> soma	0
<b>-</b> --> subtração	0
<b>*</b> --> multiplicação	1
<b>&amp;</b> --> e	todos os bits em 1
<b> </b> --> ou	0
<b>^</b> --> equiv. lógica	0
<b>&amp;&amp;</b> --> not equiv. lógica	1
<b>  </b> --> máx	0



# Exemplo - Cláusula reduction

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define SUM_INIT 0
int main(int argc, char *argv[]) {
    int i, n = 25;
    int sum, a[n];
    int ref = SUM_INIT + (n-1)*n/2;
    (void) omp_set_num_threads(3);
    for (i=0; i<n; i++)
        a[i] = i;
    #pragma omp parallel
    {
        #pragma omp single
        printf("Número de threads é %d\n", \
            omp_get_num_threads());
    }
}
```





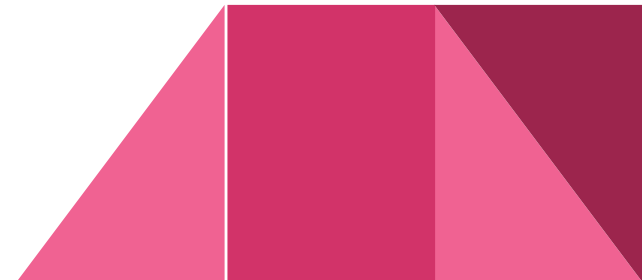
# Exemplo - Cláusula reduction

```
sum = SUM_INIT;
printf("Valor da soma antes da região paralela: %d\n", sum);

#pragma omp parallel for default(none)
shared(n,a) \ reduction(+:sum)
    for (i=0; i<n; i++)
        sum += a[i];

/*-- Fim da redução paralela --*/

printf("Valor da soma depois da região paralela: %d\n", sum);
printf("Verificação do resultado: soma = %d (deveria ser %d)\n", sum, ref);
return(0);
}
```

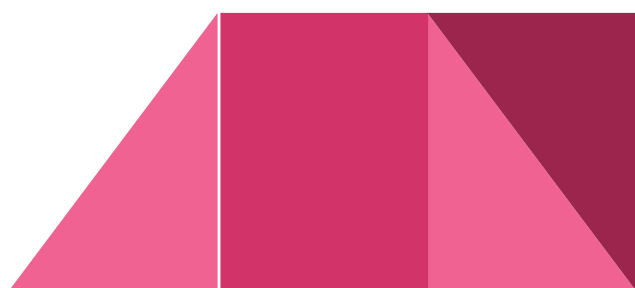


# Exemplo Método do Trapézio

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

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

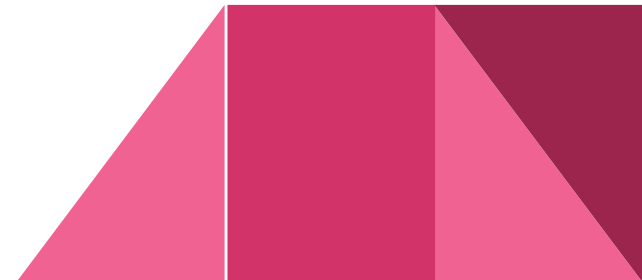
void main() {
    double integral;          /* Armazena resultado em integral */
    double a, b;              /* Limite esquerdo e direito */
    long i,n;                 /* Número de Trapezóides */
    double h;                 /* Largura da base do Trapezóide */
    double x;
    a = 0.0;
    b = 1.0;
    n = 8000000000;
    h = (b-a)/n;
    integral = (f(a) + f(b))/2.0;
    omp_set_num_threads(8);
    double start = omp_get_wtime();
```



# Exemplo Método do Trapézio

```
#pragma omp parallel for reduction(+:integral) shared(n, h)
    for (i = 1; i < n-1; i++) {
        integral += f(a + i*h);
    }

    integral *= h ;
    printf("Com n = %ld trapezoides, a estimativa \n", n);
    printf("da integral de %f ate %f = %lf \n", a, b,
integral);
    printf("Tempo: \t %f \n", omp_get_wtime()-start);
}
```



# Erros Comuns

- Esquecer a cláusula **parallel**, o que resulta em execução sequencial:

```
// no "parallel" directive
{
  #pragma omp for
    for ( size_t i = 0; i < n; ++i ) {
      ...
    }
}
```

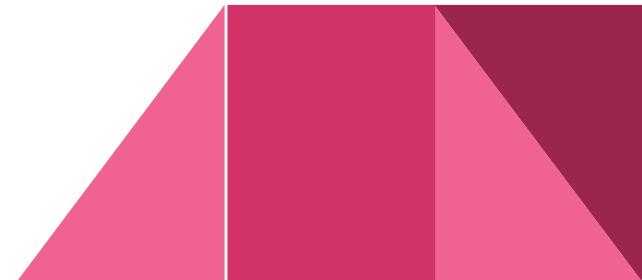
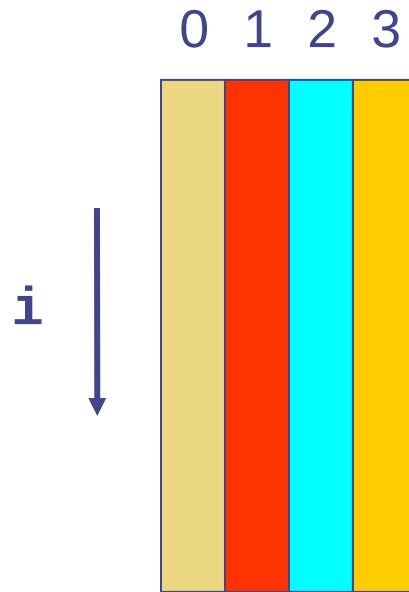
- Esquecer a palavra **for**, quando todas as threads vão executar o laço:

```
#pragma omp parallel
{
  #pragma omp // no "for" directive
    for ( size_t i = 0; i < n; ++i ) {
      ...
    }
}
```

- Infelizmente o compilador não vai avisar sobre estes erros.

# Variáveis Privadas e Compartilhadas

Exemplo: cada *thread* inicia a sua própria coluna de uma **matriz** compartilhada:



# Exemplo: Valor Inicial em uma Matriz

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char *argv[]) {
    int i, myid, n = 100000;
    float a[n][4];

    #pragma omp parallel default(none) private (i, myid)
        shared(a,n)
        myid = omp_get_thread_num();
        for (i = 0; i < n; i++){
            a[i][myid] = 1.0;
        }
    /* end parallel */
}
```

# Variáveis Privadas e Compartilhadas

- Como decidir quais variáveis devem ser compartilhadas e quais privadas?
  - A maioria das variáveis são compartilhadas
  - O índices dos laços são privados.
  - Variáveis temporárias dos laços são compartilhadas.
  - Variáveis apenas de leitura – compartilhadas
  - Matrizes principais – compartilhadas
- Às vezes a decisão deve ser baseada em fatores de desempenho.

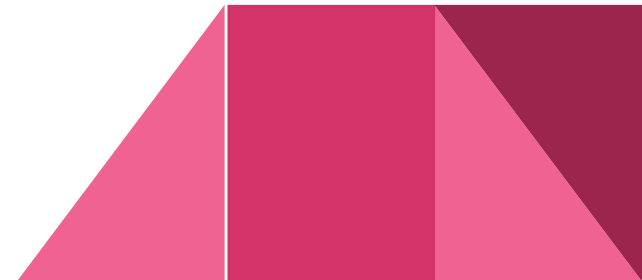


# Cláusula firstprivate

- Variáveis privadas não tem valor inicial no início da região paralela.
- Para dar um valor inicial deve-se utilizar a cláusula **firstprivate**:

C/C++:

**firstprivate**(*list*)





# Exemplo - Cláusula firstprivate


```
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1
#define FALSE 0
#include <omp.h>

int main(int argc, char *argv[]) {
{
    int *a;
    int n = 2, nthreads, vlen, indx, offset = 4,
    i, TID;
    int failed;
    (void) omp_set_num_threads(3);
    indx = offset;

/*  Prepara os parâmetros para a computação e
    aloca memória */
```


# Exemplo - Cláusula firstprivate

```
#pragma omp parallel firstprivate(indx) \
shared(a,n,nthreads,failed)
{
    #pragma omp single
    {
        nthreads = omp_get_num_threads();
        vlen = indx + n*nthreads;
        if ( (a = (int *)
malloc(vlen*sizeof(int))) == NULL )
            failed = TRUE;
        else
            failed = FALSE;
    }
} /*-- End of parallel region --*/
for(i=0; i<vlen; i++) a[i] = -i-1;
/* Cada thread acessa o vetor com a variável indx
*/
```



# Exemplo - Cláusula firstprivate

```
printf("Comprimento do segmento por thread é %d\n", \ n);  
printf("O offset do vetor a é %d\n",indx);  
#pragma omp parallel default(none)  
firstprivate(indx) \ private(i,TID) shared(n,a)  
{  
    TID = omp_get_thread_num();  
    indx += n*TID;  
    for(i=indx; i<indx+n; i++)  
        a[i] = TID + 1;  
} /*-- Final da região paralela --*/  
printf("Depois da região paralela:\n");  
for (i=0; i<vlen; i++)  
    printf("a[%d] = %d\n",i,a[i]);  
free(a);  
return(0);  
}
```

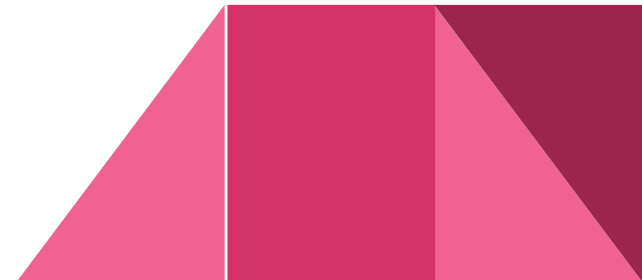


# Cláusula if

## Sintaxe

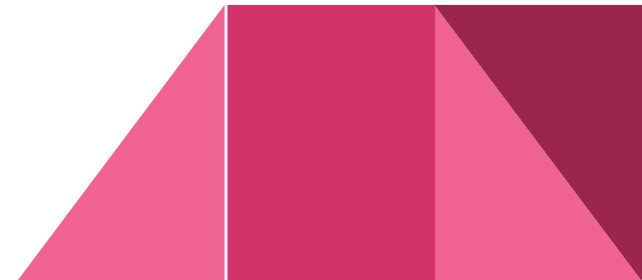
**if(expressao)**

- Onde, expressao é uma expressão inteira que, se avaliada como verdadeira (não nula), faz com que o código na região paralela seja executado em paralelo. Se a expressão for avaliada como falsa (zero) a região paralela é executada sequencialmente (por uma única thread).




# Exemplo - Cláusula if

```
double dot ( const size_t n, double * x, double * y ) {  
    double f = 0;  
    #pragma omp parallel for reduction(+:f) if(n>=1000)  
    for ( size_t i = 0; i < n; ++i )  
        f += x[i] * y[i];  
    return f;  
}
```

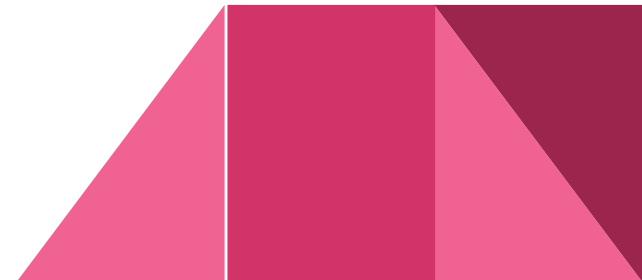


# Exemplo - Cláusula if

```
#include <stdio.h>
#include <omp.h>
void test(int val) {
    #pragma omp parallel if (val)
    if (omp_in_parallel()) {
        #pragma omp single
        printf("Valor = %d, paralelizada com %d threads\n", val, omp_get_num_threads());
    }
    else {
        printf("Valor = %d, serializada\n", val);
    }
}
int main( ) {
    omp_set_num_threads(2);
    test(0);
    test(2);
}
```

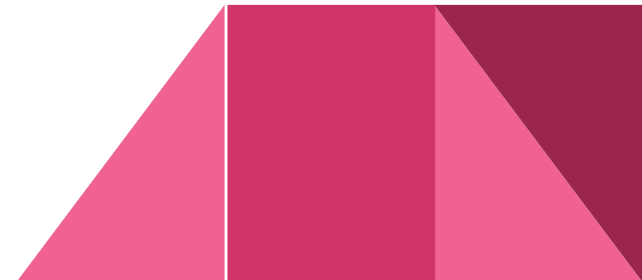


# Diretivas para Compartilhamento de Trabalho



# Diretivas para Compartilhamento de Trabalho

- Diretivas que aparecem dentro de uma região paralela e indicam como o trabalho deve ser compartilhado entre as threads.
  - Laços do/for paralelos
  - Seções paralelas
  - Diretivas MASTER e SINGLE

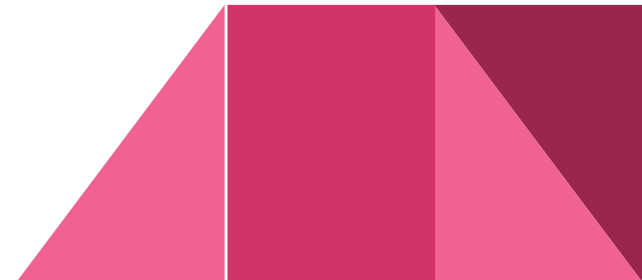




# Laços for paralelos

- Laços são a maior fonte de paralelismo na maioria dos códigos. Diretivas paralelas de laços são portanto muito importantes!
- Um laço do/for paralelo divide as iterações do laço entre as *threads*.
- Apresentaremos aqui apenas a forma básica.
- Sintaxe C/C++:

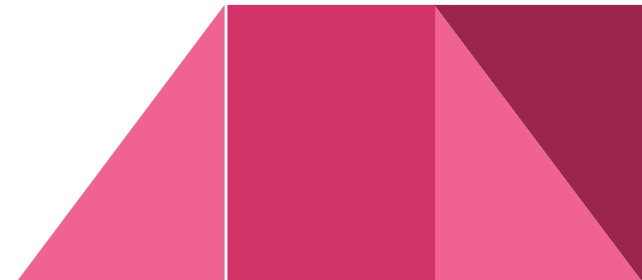
```
#pragma omp for [clausulas]  
for loop
```



# Laços for paralelos

- Sem cláusulas adicionais, a diretiva **for** usualmente particionará as iterações o mais igualmente possível entre as *threads*.
- Contudo, isto é dependente de implementação e ainda há alguma ambigüidade:

**Ex.: 7 iterações, 3 *threads*. Pode ser particionado como 3+3+1 ou 3+2+2**

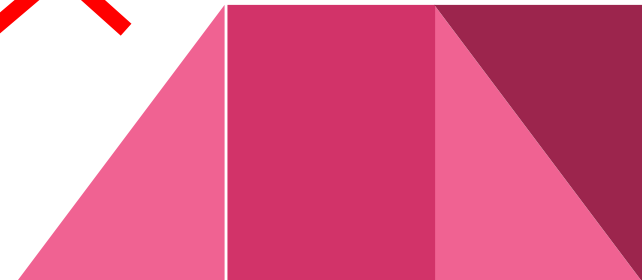


# Laços for paralelos

- Como você pode dizer se um laço é paralelo ou não?
- Teste: se o laço dá o mesmo resultado se executado na ordem inversa então ele é quase certamente paralelo.
- Desvios para fora do laço não são permitidos.
- Exemplos:

1.

```
for (i=1; i < n; i++)  
{  
    a[i] = 2*a[i-1];  
}
```



# Laços for paralelos

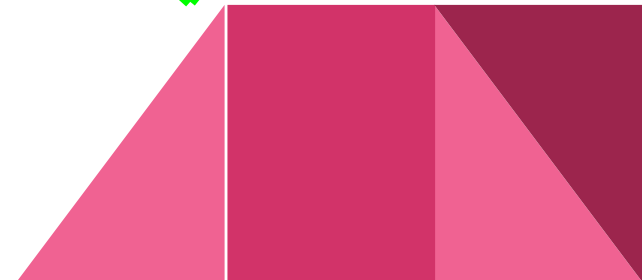
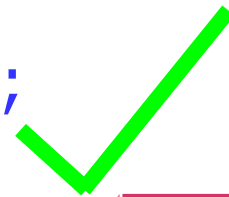
2.

```
ix = base;  
for (i=0; i < n; i++) {  
    a[ix] = a[ix]* b[i];  
    ix  = ix + stride;  
}
```



3.

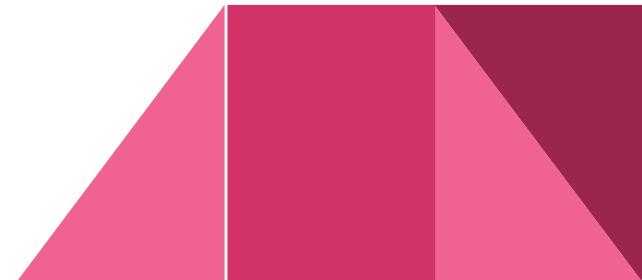
```
for (i=0; i<n; i++){  
    b[i]= (a[i] - a[i-1])*0.5;  
}
```



# Exemplo - Laços Paralelos

## Exemplo:

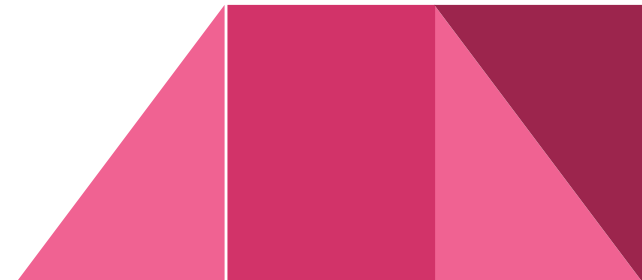
```
#pragma omp parallel
#pragma omp for
    for (i=1; i <=n; i++){
        b[i] = (a[i]- a[i-1])*0.5;
    }
/* end parallel for */
```



# A diretiva for paralela

- Esta construção é tão comum que existe uma forma que combina a região paralela e a diretiva do/for:
- Sintaxe C/C++:

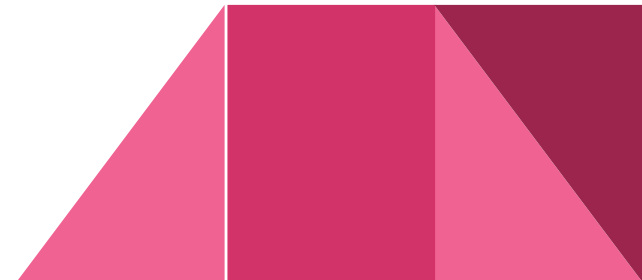
```
#pragma omp parallel for [clausulas]  
    for loop
```



# Exemplo (saxpy)

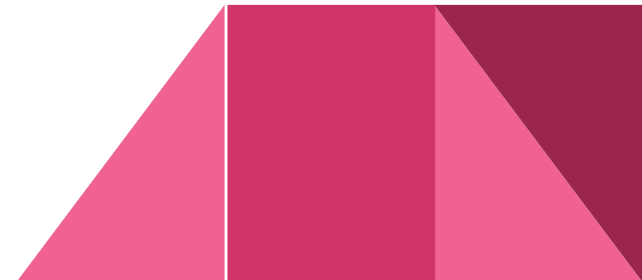
```
#pragma omp parallel for
for (i=0; i < n; i++)
{
    z[i] = a * x[i] + y;
}
```

- Este exemplo realiza a operação “multiply-add”, que é uma multiplicação de um vetor por um valor que em seguida é somado a uma variável ou constante.



# Cláusulas

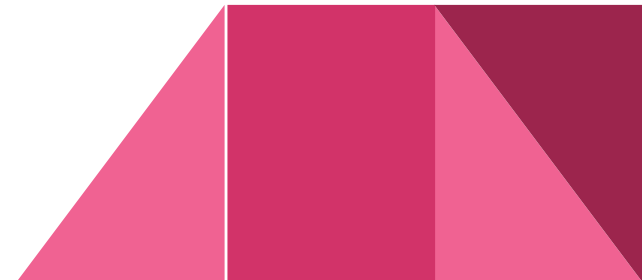
- A diretiva **for** pode ter cláusulas **private** e **firstprivate** as quais se referem ao escopo do laço.
- Note que a variável de índice do laço paralelo é **private** por padrão (mas outros índices de laços não são).
- A diretiva **parallel for** pode usar todas as cláusulas disponíveis para a diretiva **parallel**.
- Relembrando:
  - **num\_threads**
  - **default, private, shared**
  - **reduction**
  - **firstprivate**
  - **if**
  - **copyin** (não abordado)





# Seções Paralelas

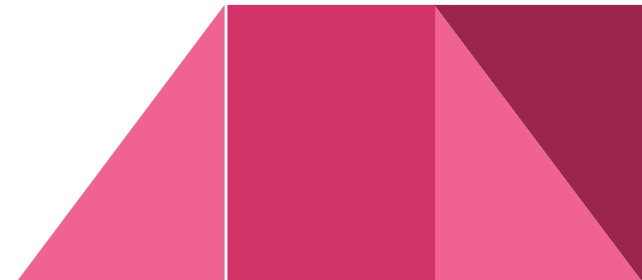
- Permitem que blocos separados de código sejam executados em paralelo (ex. diversas subrotinas independentes)
- Não é escalável: o código fonte deve determinar a quantidade de paralelismo disponível.
- Raramente utilizada, exceto com paralelismo aninhado (que não será abordado aqui).



# Seções Paralelas

**C/C++:**

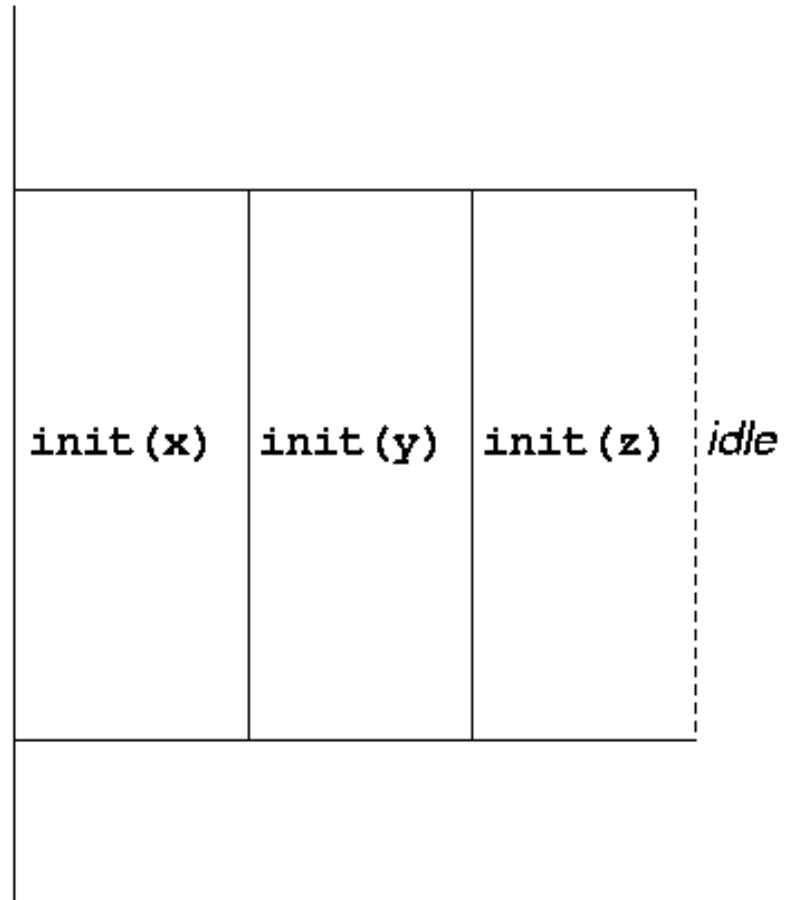
```
#pragma omp sections [clausulas]  
{  
    [ #pragma omp section ]  
        structured-block  
    [ #pragma omp section  
        structured-block  
        . . . ]  
}
```



# Seções Paralelas

## Exemplo:

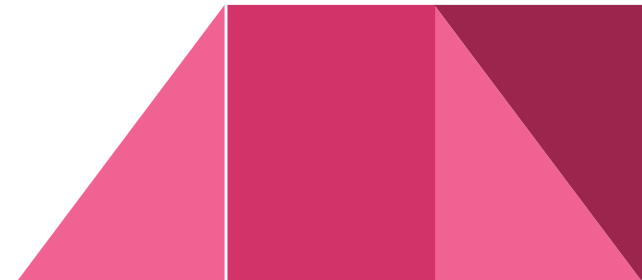
```
#pragma omp parallel
#pragma omp sections
{
  #pragma omp section
    init(x);
  #pragma omp section
    init(y);
  #pragma omp section
    init(z);
}
```



# Seções Paralelas

- Diretivas **sections** podem ter as cláusulas **private**, **firstprivate** e **lastprivate**
- Cada seção deve conter um bloco estruturado: não pode haver desvio para dentro ou fora de uma seção.
- Forma abreviada C/C++:

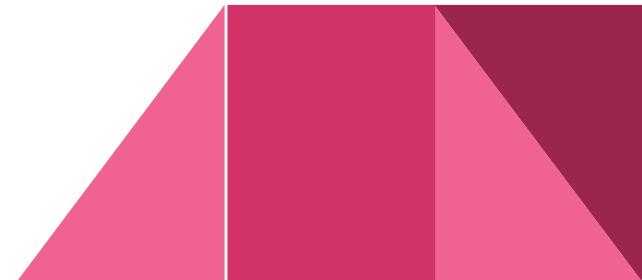
```
#pragma omp parallel sections [clausulas]
{
    . . .
}
```



# Diretiva single

- Indica que um bloco de código deve ser executado apenas por uma *thread*.
- A primeira *thread* que alcançar a diretiva **single** irá executar o bloco.
- Outras *threads* devem esperar até que o bloco seja executado.
- Sintaxe C/C++:

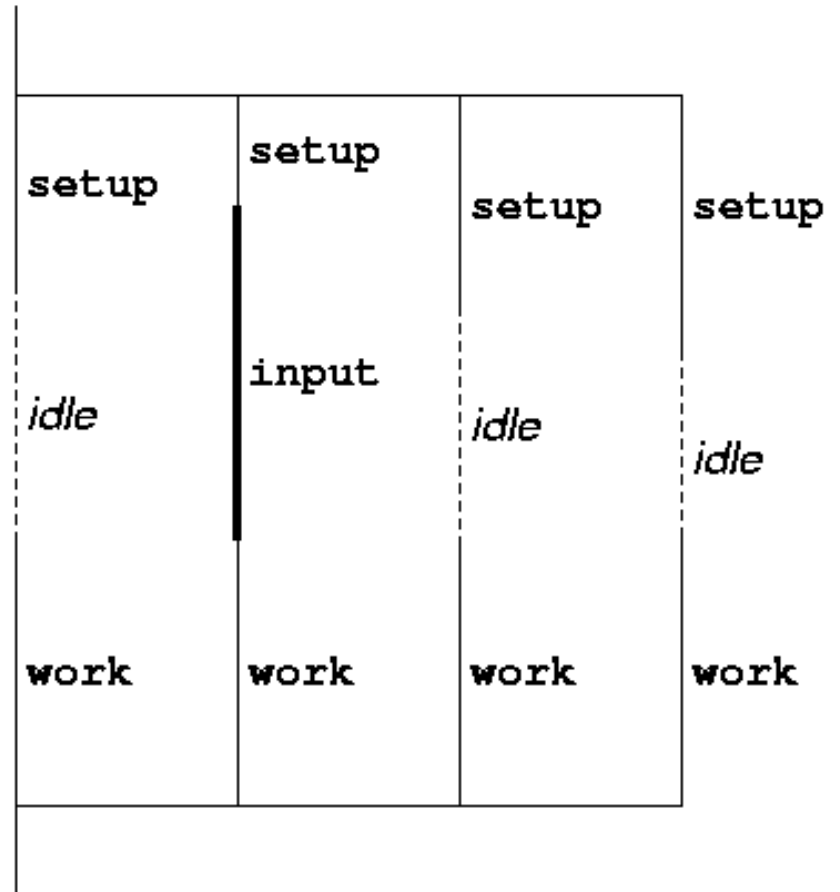
```
#pragma omp single [clausulas]  
    structured block
```



# Diretiva single

Exemplo:

```
#pragma omp parallel
{
    setup(x);
    #pragma omp single
    {
        input(y);
    }
    work(x,y);
}
```



# Diretiva single

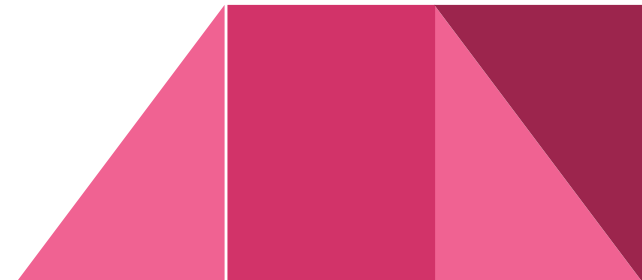
- A diretiva **single** pode ter cláusulas **private** e **firstprivate**.
- Existe uma barreira implícita no final do bloco estruturado. As demais threads esperam pela thread que está executando o bloco.
- A diretiva deve conter um bloco estruturado: não pode haver desvio dentro ou para fora do dele.



# Diretiva master

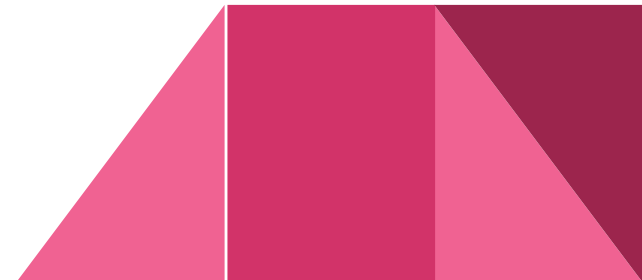
- Indica que um bloco deve ser executado apenas pela thread master (thread 0).
- Outras threads pulam o bloco e continuam a execução: é diferente da diretiva **single** neste aspecto.
- Na maior parte das vezes utilizada para E/S.
- Sintaxe C/C++:

```
#pragma omp master  
    structured block
```





# Mais sobre laços paralelos do/for



# Cláusula lastprivate

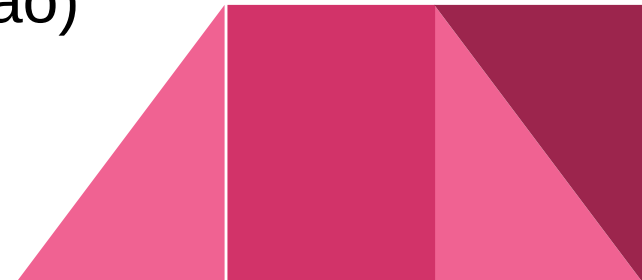
- Algumas vezes é necessário que se saiba o valor que uma variável privada terá na saída de um laço (normalmente é indefinido)

**Sintaxe:**

**C/C++:**

**lastprivate(list)**

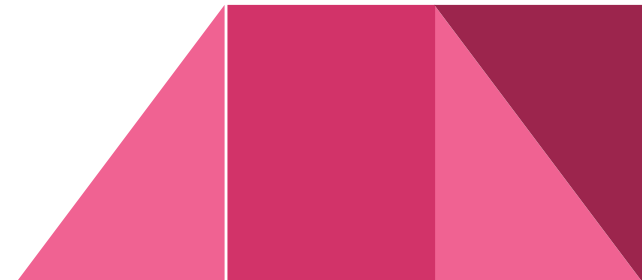
- Também se aplica à diretiva **sections** (a variável tem um valor atribuído a ela a última seção)



# Cláusula lastprivate

## Exemplo:

```
#pragma omp parallel
#pragma for lastprivate (i)
    for (i=0,func(l,m,n)){
        d[i]=d[i]+e*f[i];
    }
    ix = i-1;
    . . .
/* pragma end for */
```



# Cláusula `schedule`

- A cláusula **`schedule`** permite uma variedade de opções por especificar quais iterações dos laços são executadas por quais *threads*.
- Sintaxe:

C/C++:

**`schedule (kind[, chunksize])`**

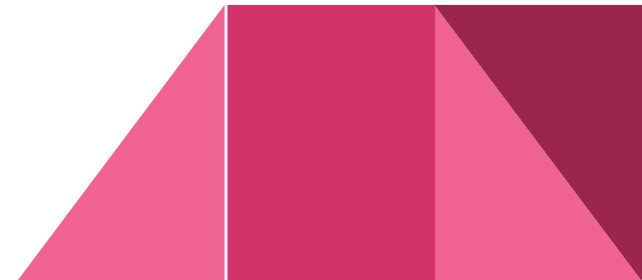
onde *kind* pode ser **STATIC**, **DYNAMIC**, **GUIDED** ou **RUNTIME** e *chunksize* é uma expressão inteira com valor positivo.

**Ex.: `#pragma for schedule(DYNAMIC, 4)`**

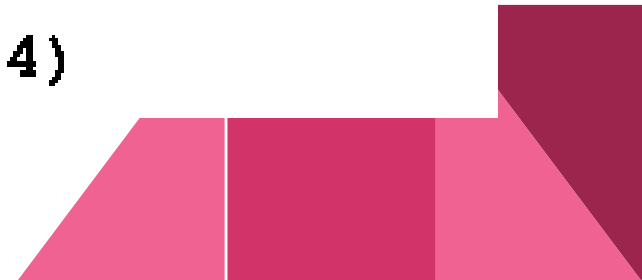
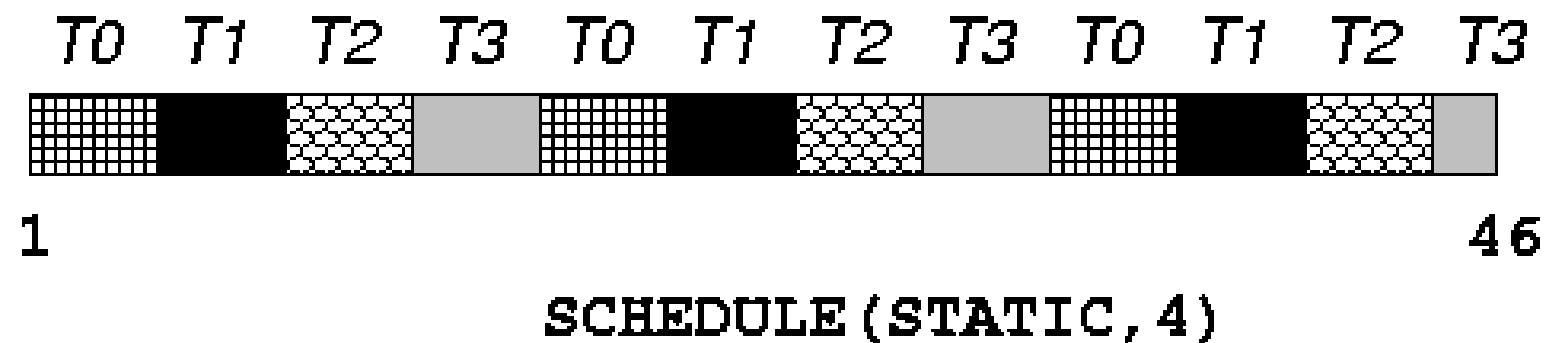
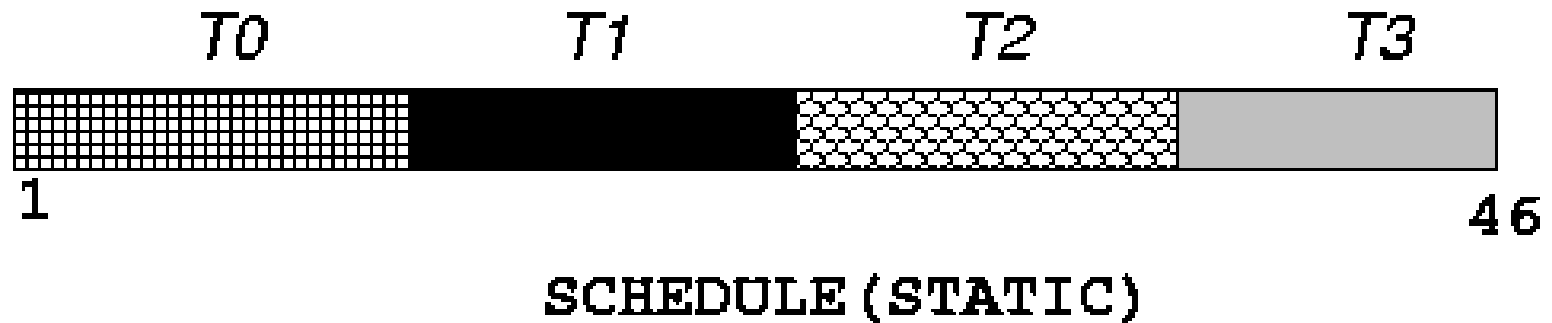


# Escalonamento static

- Sem a especificação de **chunksize**, o espaço de iteração é dividido em pedaços (aproximadamente) iguais e cada pedaço é atribuído a cada *thread* (escalonamento em bloco).
- Se o valor de **chunksize** é especificado, o espaço de iteração é dividido em pedaços, cada um com **chunksize** iterações, e os pedaços são atribuídos ciclicamente a cada *thread* (escalonamento block cyclic)



# Escalonamento static



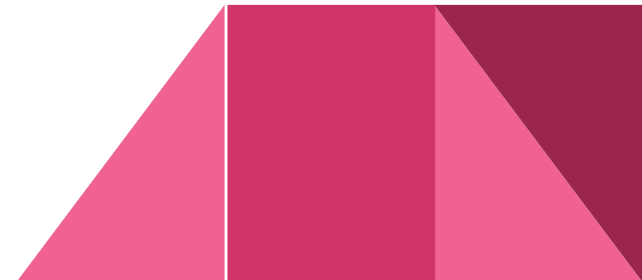
# Escalonamento dynamic

- O escalonamento **dynamic** divide o espaço de iteração em pedaços de tamanho **chunksize**, e os atribui para as *threads* com uma política first-come-first-served.
- i.e. se uma *thread* terminou um pedaço, ela recebe o próximo pedaço na lista.
- Quando nenhum valor de **chunksize** é especificado, o valor padrão é 1.



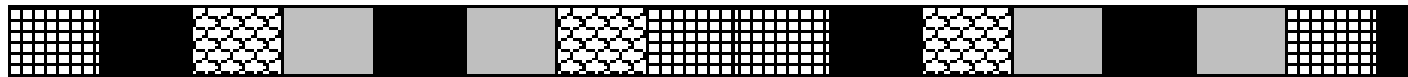
# Escalonamento guided

- O escalonamento **guided** é similar ao **dynamic**, mas os pedaços iniciam grandes e se tornam menores exponencialmente.
- O tamanho do próximo pedaço é (a grosso modo) o número de iterações restantes dividido pelo número de *threads*.
- O valor **chunksize** especifica o tamanho mínimo dos pedaços.
- Quando nenhum valor de **chunksize** é especificado, o padrão é 1.





# Escalonamentos dynamic e guided



1

**SCHEDULE (DYNAMIC, 3)**

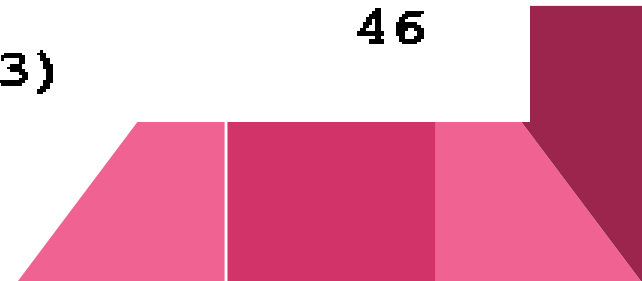
46



1

**SCHEDULE (GUIDED, 3)**

46

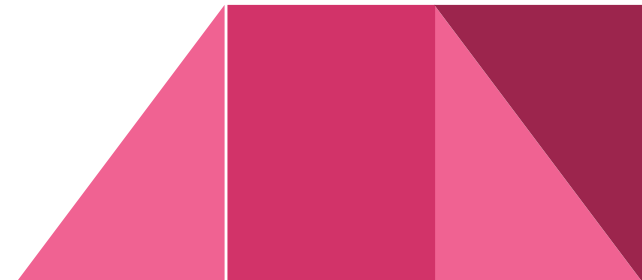


# Escalonamento runtime

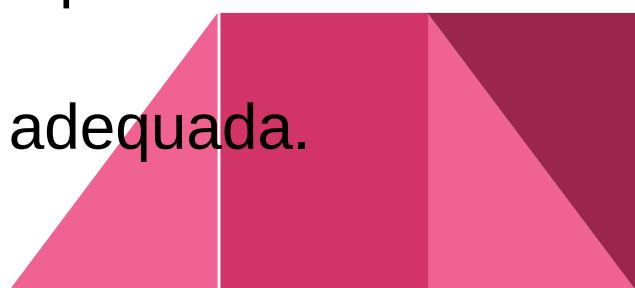
- O escalonamento RUNTIME delega a escolha do escalonamento para a execução, quando é determinado pelo valor da variável de ambiente OMP\_SCHEDULE.

```
$ export OMP_SCHEDULE="guided,4"
```

- É ilegal especificar um valor de chunksize com o escalonamento RUNTIME.



# Escolhendo um Escalonamento

- Quando utilizar um escalonamento?
  - STATIC: melhor para laços balanceados – menor sobrecarga.
  - STATIC,n : melhor para laços com desbalanceamento suave.
  - DYNAMIC : útil se as iterações tem grande variação de carga, mas acaba com a localidade espacial dos dados.
  - GUIDED: freqüentemente menos cara que DYNAMIC, mas tenha cuidado com laços onde as primeiras iterações são as mais caras!
  - Use RUNTIME para experimentação adequada.
- 

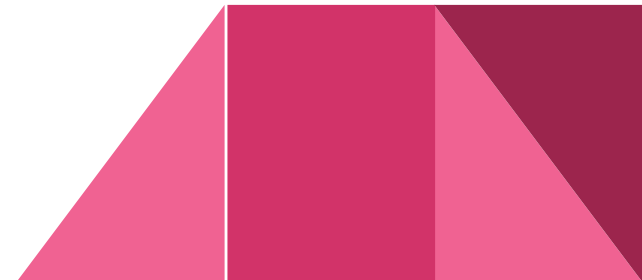
# Diretiva ordered

- Pode especificar o código dentro de um laço que deverá ser executado na ordem em que seria se executado seqüencialmente.

- Sintaxe:

C/C++: `#pragma omp ordered`  
*structured block*

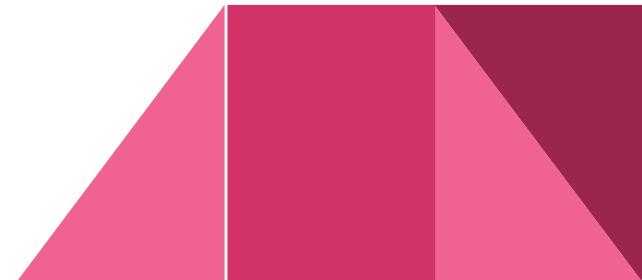
- Pode aparecer dentro de uma diretiva DO/FOR que tiver a cláusula ORDERED especificada.



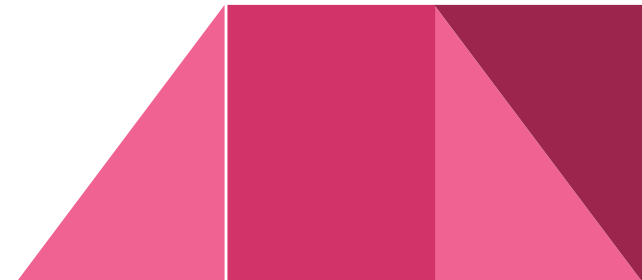
# Diretiva ordered

- Exemplo:

```
#pragma omp parallel for ordered
    for (j =0; j < n; j++)
        . . .
#pragma omp ordered
    printf ("%d %d \n", j, count[j])
    . . .
```



# Sincronização

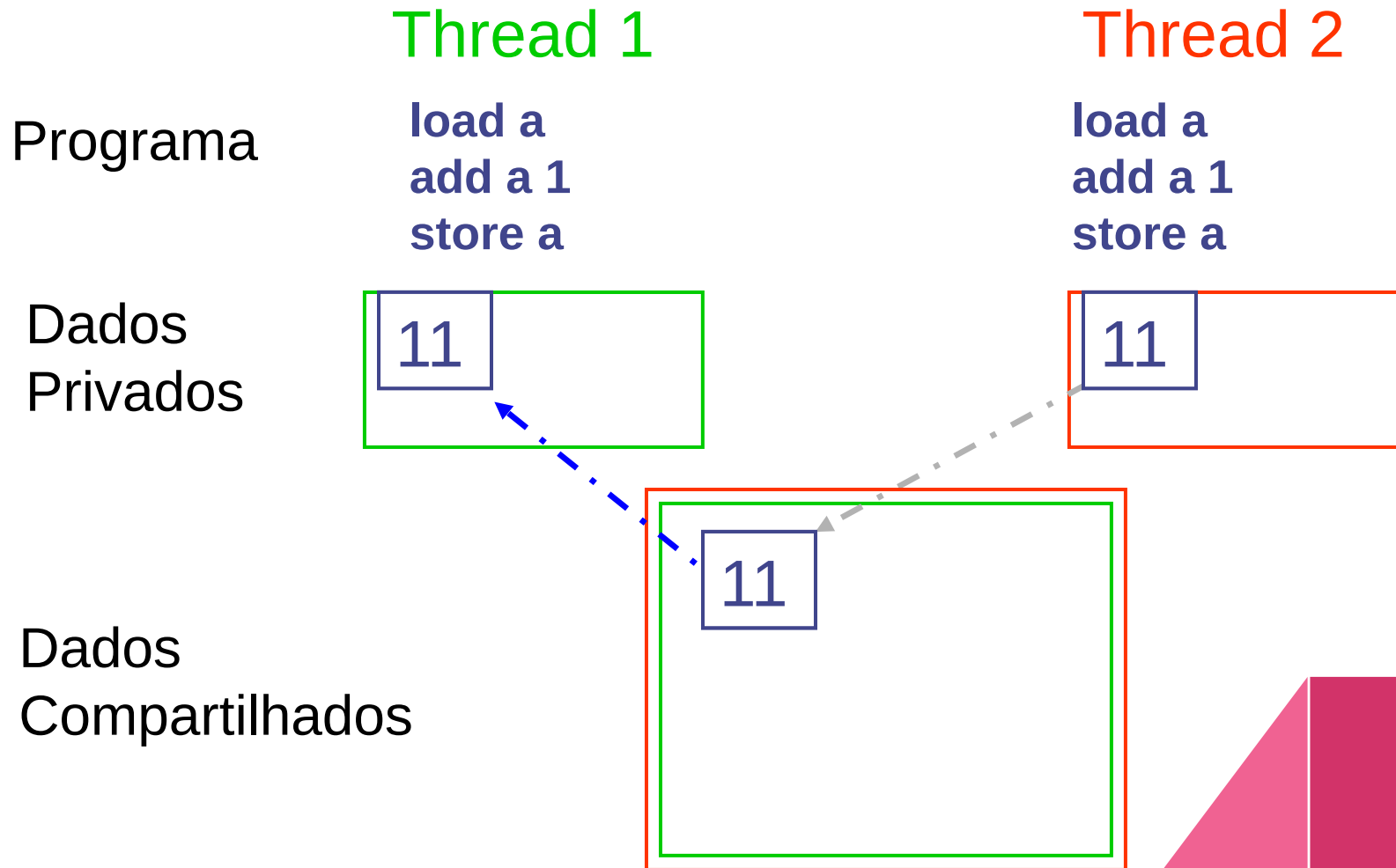


# Sincronização

- Há necessidade de assegurar que as ações nas variáveis compartilhadas ocorram na maneira correta: por ex.: a *thread* 1 deve escrever na variável **A** antes da *thread* 2 faça a sua leitura, ou a *thread* 1 deve ler a variável **A** antes que a *thread* 2 faça sua escrita.
- Note que as atualizações para variáveis compartilhadas (p.ex.  $a = a + 1$ ) não são atômicas! Se duas *threads* tentarem fazer isto ao mesmo tempo, uma das atualizações pode ser perdida.



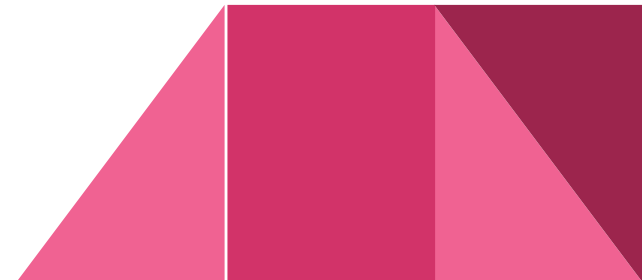
# Exemplo de Sincronização





# O que é necessário?

- É necessário sincronizar ações em variáveis compartilhadas.
- É necessário assegurar a ordenação correta de leituras e escritas.
- É necessário proteger a atualização de variáveis compartilhadas (não atômicas por padrão).



# Diretiva barrier

- Nenhuma thread pode prosseguir além de uma barreira até que todas as outras threads cheguem até ela.
- Note que há uma barreira implícita no final das diretivas **for**, **sections** e **single**.
- Sintaxe:

**C/C++:**

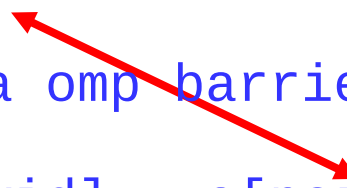
```
#pragma omp barrier
```

- Ou nenhuma ou todas as threads devem encontrar a barreira: senão **DEADLOCK!!**

# Diretiva barrier

## Exemplo:

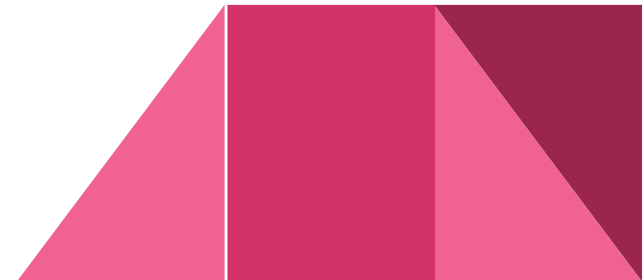
```
#pragma omp parallel private(i,myid,neighb)
{
    myid = omp_get_thread_num();
    neighb = myid - 1;
    if (myid == 0) neighb = omp_get_num_threads()-
1;
    ...
    a[myid] = a[myid]*3.5;
    #pragma omp barrier
    b[myid] = a[neighb] + c
    ...
}
```



- Barreira requerida para forçar a sincronização em no vetor **a**

# Cláusula nowait


- A cláusula **nowait** pode ser usada para suprimir as barreiras implícitas no final das diretivas **for**, **sections** e **single**. (Barreiras são caras!)
- Sintaxe:  
    **C/C++:**  
    **#pragma omp for nowait**  
                    *for loop*
- Igualmente para **sections** e **single**.



# Cláusula nowait

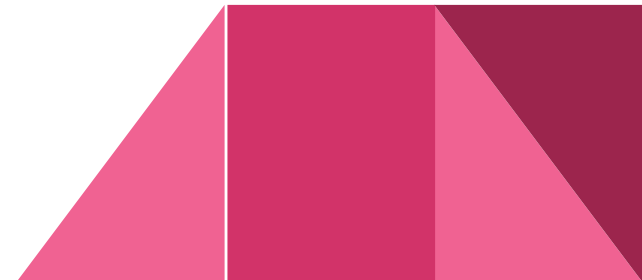
Exemplo: Dois laços sem dependências

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (j=0; j < n; j++){
    a[j] = c * b[j];
  }
  #pragma omp for nowait
  for (i=0; i < m; i++){
    x[i] = sqrt(y[i]) * 2.0;
  }
}
```



# Cláusula nowait

- Use com **EXTREMO CUIDADO!**
- É muito fácil remover uma barreira que é necessária.
- Isto resulta no pior tipo de erro: comportamento não-determinístico da aplicação (às vezes o resultado é correto, às vezes não, o comportamento se altera no depurador, etc.).
- Pode ser um bom estilo de codificação colocar a cláusula **nowait** em todos os lugares e fazer todas as barreiras explicitamente.

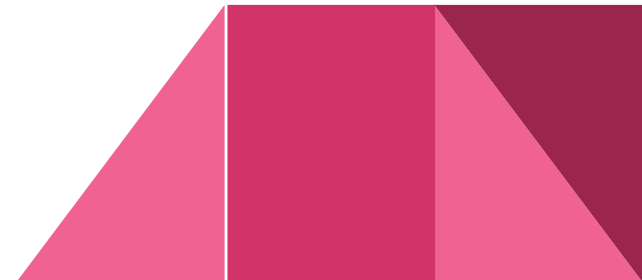


# Cláusula nowait

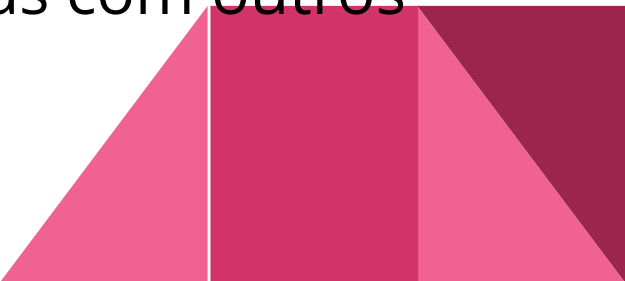
## Exemplo:

```
#pragma omp for
    for (j =0; j < n; j++){
        a[j] = b[j] + c[j];
    }
#pragma omp for
    for (j =0; j < n; j++){
        d[j] = e[j] * f;
    }
#pragma omp for
    for (j =0; j < n; j++){
        z[j] = (a[j]+a[j+1]) * 0.5;
    }
```

**Pode-se remover a primeira barreira, OU a segunda, mas não ambas, já que há uma dependência em **a****



# Seções Críticas

- Uma seção crítica é um bloco de código que só pode ser executado por uma *thread* por vez.
  - Pode ser utilizado para proteger a atualização de variáveis compartilhadas.
  - A diretiva **critical** permite que as seções críticas recebam nomes.
  - Se uma thread está em uma seção crítica com um dado nome, nenhuma outra *thread* pode estar em uma seção crítica com o mesmo nome ( embora elas possam estar em seções críticas com outros nomes).
- 



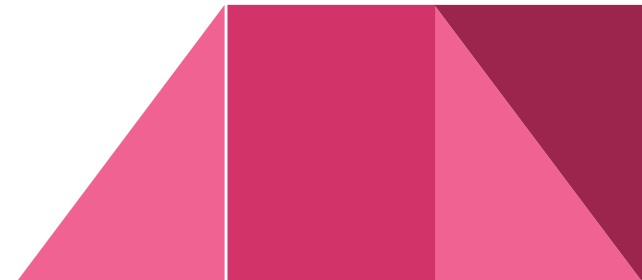
# Diretiva critical

- Sintaxe:

**C/C++:**

```
#pragma omp critical [( name )]  
    structured block
```

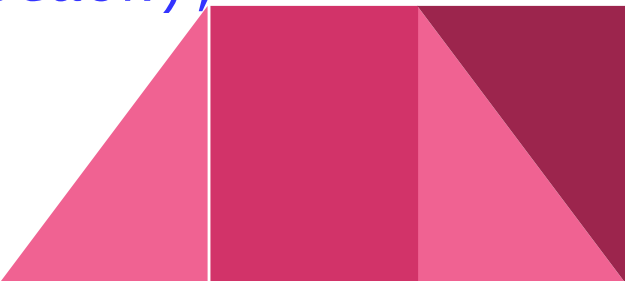
- Se o nome é omitido, um nome nulo é assumido (todas as seções críticas sem nome tem efetivamente o mesmo nome).



# Diretiva critical

Exemplo: colocando e retirando de uma pilha

```
#pragma omp parallel shared(stack),  
    private(inext,inew)  
    ...  
#pragma omp critical (stackprot)  
{  
    inext = getnext(stack};  
}  
    work(inext,inew);  
#pragma omp critical (stackprot)  
    if (inew > 0) putnew(inew,stack);  
}  
    ...
```



# Diretiva atomic

- Usada para proteger uma atualização única para uma variável compartilhada.
- Aplica-se apenas a uma única sentença.
- Sintaxe:

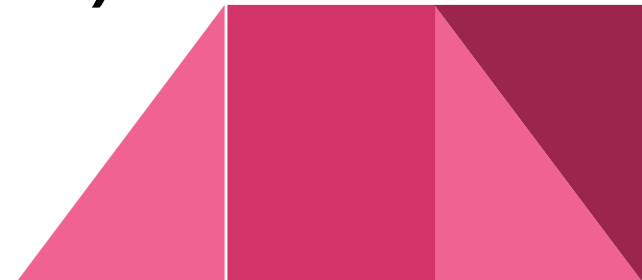
**C/C++:**

```
#pragma omp atomic  
statement
```

- Onde *statement* só pode conter os seguintes operadores:

**Unários:** ++, -- (prefixado e pós-fixado)

**Binários:** +, -, \*, /, ^, &, |, <<, >>

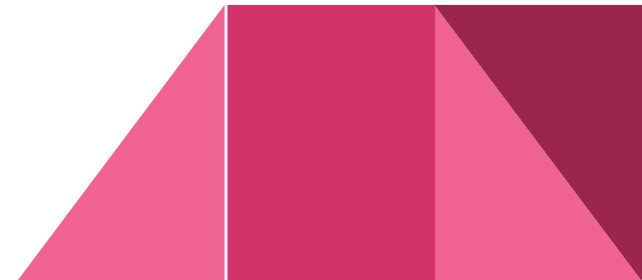


# Diretiva atomic

- `++x ; --x ; x++; x--;`
- `x += expr ; x -= expr ;`
- `x = x + expr ; x = x - expr ;`
- `x = expr + x ; x = expr - x ;`
- `x |= expr ; x ^= expr ;`
- `x = x | expr ; x = x ^ expr ;`
- `x = expr | x ; x = expr ^ x ;`
- `x*= expr ; x = x*expr ; x= expr*x ;`
- `x<<= expr ; x = x << expr ; x= expr << x ;`
- `x/= expr ; x &= expr ;`
- `x= x / expr ; x = x & expr ;`
- `x= expr / x ; x = expr & x ;`
- `x>>= expr ; x= x >> expr ;`
- `x= expr >> x ;`

# Diretiva atomic

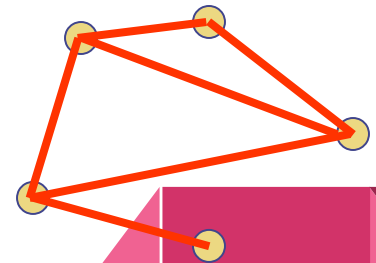
- Note que a avaliação da expressão não é atômica.
- Pode ser mais eficiente que usar diretivas **critical**, por exemplo, se diferentes elementos do vetor ou matriz podem ser protegidos separadamente.



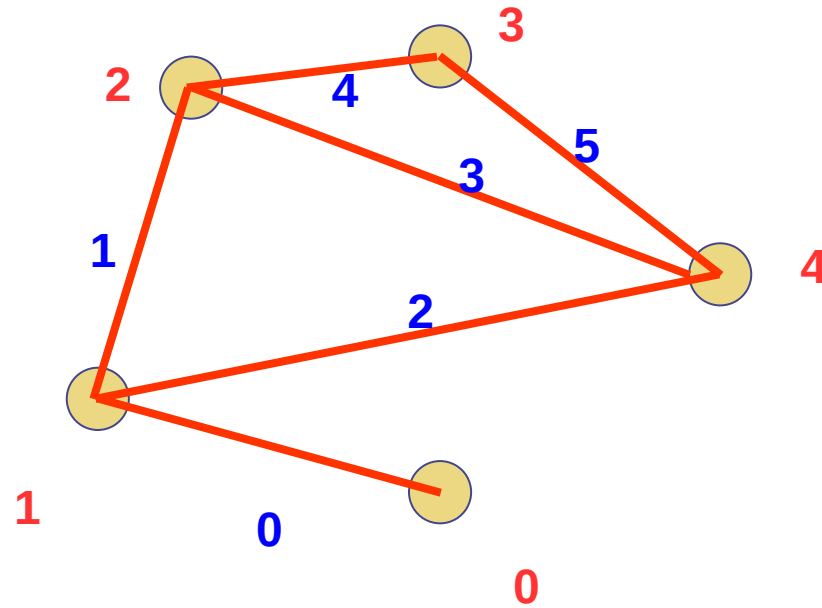
# Diretiva atomic

- Exemplo (computar o grau de cada vértice em um grafo):

```
#pragma omp parallel for
    for (j=0; j<nedges; j++){
#pragma omp atomic
        degree[edge[j].vertex1]++;
#pragma omp atomic
        degree[edge[j].vertex2]++;
    }
```



# Diretiva atomic



# Rotinas lock

- Ocasionalmente pode ser necessário mais flexibilidade que a fornecida pelas diretivas **critical e atomic**.
- Um **lock** é uma variável especial que pode ser marcada por uma *thread*. Nenhuma outra *thread* pode marcar o **lock** até que a *thread* que o marcou o desmarque.
- Marcar um **lock** pode tanto pode ser bloqueante como não bloqueante.
- Um **lock** deve ter um valor inicial antes de ser usado e pode ser destruído quando não for mais necessário.
- Variáveis de **lock** não devem ser usadas para qualquer outro propósito.



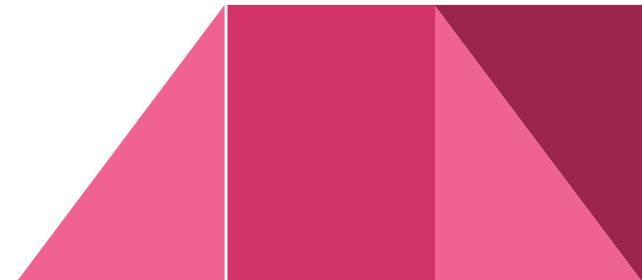


# Rotinas de lock – Sintaxe

## C/C++:

```
#include <omp.h>
void omp_init_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

- Existem também rotinas de **lock** aninháveis que permitem a uma mesma *thread* ativar um **lock** múltiplas vezes antes de liberá-lo o mesmo número de vezes.

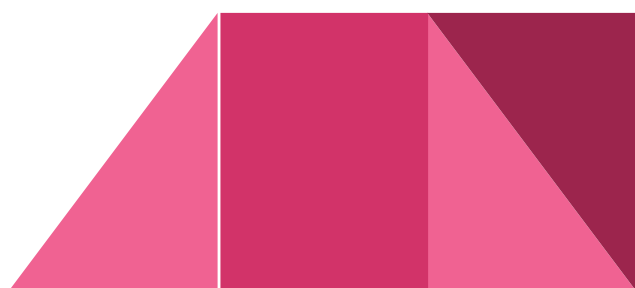


# Rotinas de lock – Exemplo

## Exemplo:

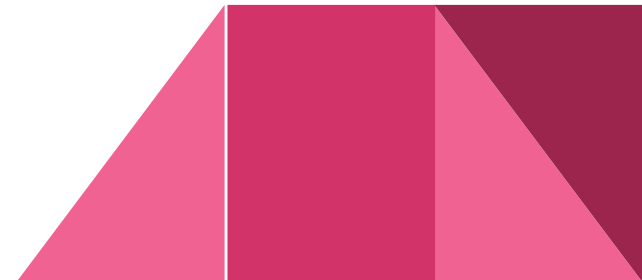
```
call omp_init_lock(ilock)
#pragma omp parallel shared(ilock)
...
do {
    do_something_else(); }
while ( ~ omp_test_lock(ilock))

work();
omp_unset_lock(ilock);
...
```

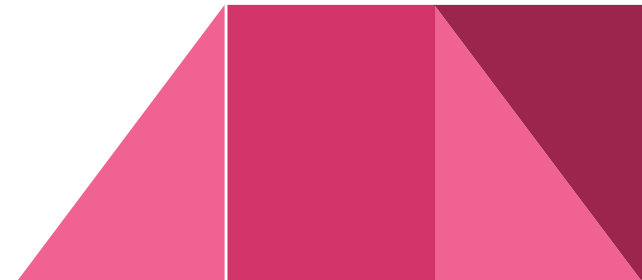


# Escolhendo a Sincronização

- Como uma regra simples, use a diretiva **atomic** sempre que possível, já que permite o máximo de otimização.
- Se não for possível use a diretiva **critical**. Tenha cuidado de usar diferentes nomes sempre que possível.
- Como um último recurso você pode ter que usar as rotinas de **lock**, mas isto deve ser uma ocorrência muito rara.



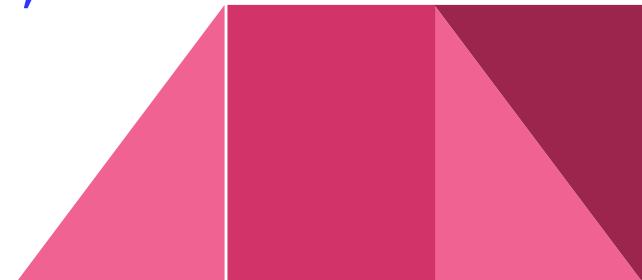
# Funcionalidades Adicionais



# Reduções em Arranjos

- Arranjos podem ser usados como variáveis de redução (anteriormente só escalares e elementos de um arranjo).
- Exemplo:

```
#pragma omp parallel for private (i)
reduction (+:b)
    for (j = 0; j < N; j++)
        for (i=0; i < M; i++)
            b(i) = b(i) + b(i,j);
```

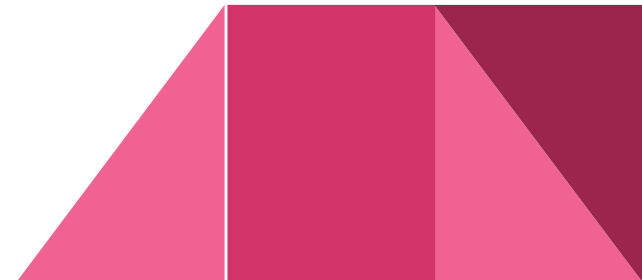


# Cláusula copyprivate

- Difunde o valor de uma variável privada para todas as *threads* no final de uma diretiva **single**.
- Talvez o uso mais importante seja a leitura de valores de variáveis privadas.
- Sintaxe:

**C/C++:**


```
#pragma omp single copyprivate(list)
```



# Cláusula copyprivate

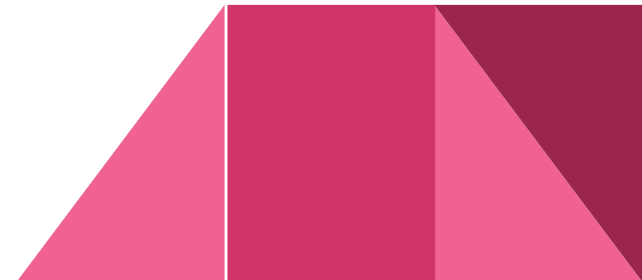
## Exemplo:

```
#pragma omp parallel private (a,b)
{
    ...
    #pragma omp single copyprivate(a)
    {
        scanf ("Entre com o valor = %d", a);
    }
    b = a * a;
    ...
}
```



# Controle de Afinidade de Threads

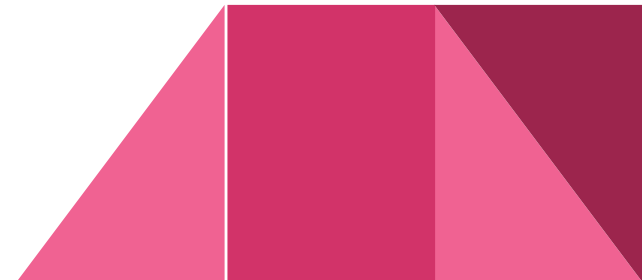
- A afinidade de *threads* se torna importante em sistemas com grande número de núcleos.
- OMP\_PLACES define os lugares aos quais as *threads* são atribuídas.
  - **threads**: cada local corresponde a uma *thread* em *hardware*.
  - **cores**: cada local corresponde a um único núcleo (consistindo de uma ou mais *threads*).
  - **sockets**: cada local corresponde a um único soquete (chip) (consistindo de um ou mais núcleos).





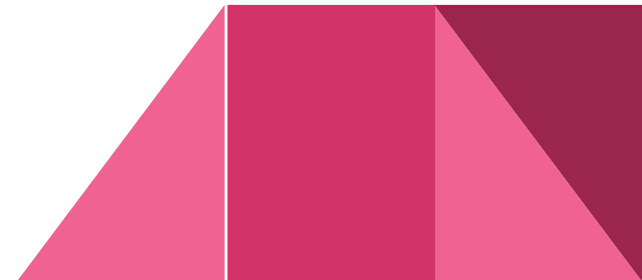
# Controle de Afinidade de Threads

- **OMP\_PROC\_BIND**
  - **false:** afinidade de *thread* desativada, o ambiente de execução pode mover threads entre locais.
  - **true:** trava as *threads* ao núcleos.
  - **spread:** distribui as *threads* igualmente entre os locais.
  - **close:** empacota as *threads* perto da *thread* master na lista de locais.
  - **master:** coloca as *threads* junto com a *thread* master.



# Usando o OpenMP

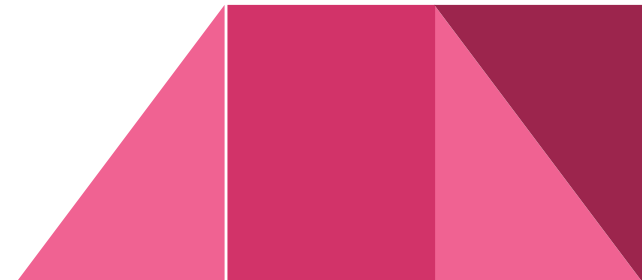
- Comparações com Troca de Mensagem:
  - Algoritmo de decomposição de domínio é o mesmo, mas a implementação é mais simples:
    - Nenhuma necessidade de troca de mensagens, células fantasma ou buffers de sombra.
    - Dados globais, variáveis de campo compartilhadas: leitura por qualquer thread, escrita pode ser compartilhada.
  - Paraleliza apenas partes do código que são significativas, não há necessidade de converter o código todo.
    - Pré-processamento, pós-processamento pode ser deixado à parte.



# Usando o OpenMP

- **OMP\_SCHEDULE** – especifica o tipo de escalonamento para divisão das iterações do laço entre as threads, para uso pelas cláusulas “**omp for**” e “**omp parallel for**” com uso da cláusula de escalonamento “**runtime**”.
- O valor padrão para esta variável é “**static**”.
- Se o tamanho do **chunk** não for especificado, um valor de 1 é assumido, exceto no caso de escalonamento estático.
- Exemplos do uso da cláusula **OMP\_SCHEDULE** são os seguintes:

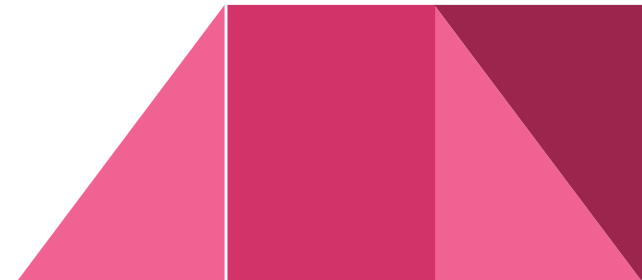
```
$ setenv OMP_SCHEDULE "static, 5"  
$ setenv OMP_SCHEDULE "guided, 8"  
$ setenv OMP_SCHEDULE "dynamic"
```



# Usando o OpenMP

- **MPSTKZ** – aumenta o tamanho das pilhas utilizadas pelas threads executando regiões paralelas. Para uso com programas que utilizam grandes quantidades de variáveis locais às threads nas rotinas chamadas nas regiões paralelas.
- O valor deve ser um inteiro <n> concatenado com **M** ou **m** para especificar o tamanho da pilha em Megabytes:

```
$ setenv MPSTKZ 8M
```



# Usando o OpenMP

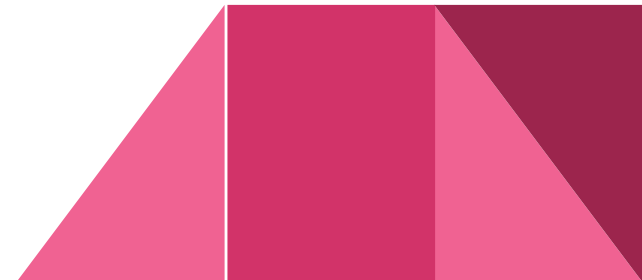
- Compilando código OpenMP paralelizado para C/C++ usando o compilador Intel.
- C:  

```
$ icc -o myprog myprog.c -openmp -openmp_report2  
$ gcc -o myprog myprog.c -fopenmp
```
- C++:  

```
$ icc -o myprog myprog.C -openmp -openmp_report2  
$ gcc -o myprog myprog.C -fopenmp
```
- Habilitando `-openmp_report2` oferece como saída diagnóstico de paralelização durante o tempo de compilação.

# Referências

- 1) Gabriel P. Silva, Calebe Bianchini e Evaldo B. Costa  
“Programação Paralela – Um Curso Introdutório” Editora Casa do Código, 2022
- 2) Book: “Parallel Programming in OpenMP”, Chandra et. al., Morgan Kaufmann, ISBN 1558606718.
- 3) Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, 2nd ed., Englewood Cliffs, NJ, Prentice--Hall, 1988.
- 4) <http://www.openmp.org>
- 5) <http://www.compunity.org>
- 6) <http://scv.bu.edu/SCV/Tutorials/OpenMP/>



# Obrigado!

Gabriel P. Silva

[gabriel@ic.ufrj.br](mailto:gabriel@ic.ufrj.br)

<http://github.com/gpsilva2003>

