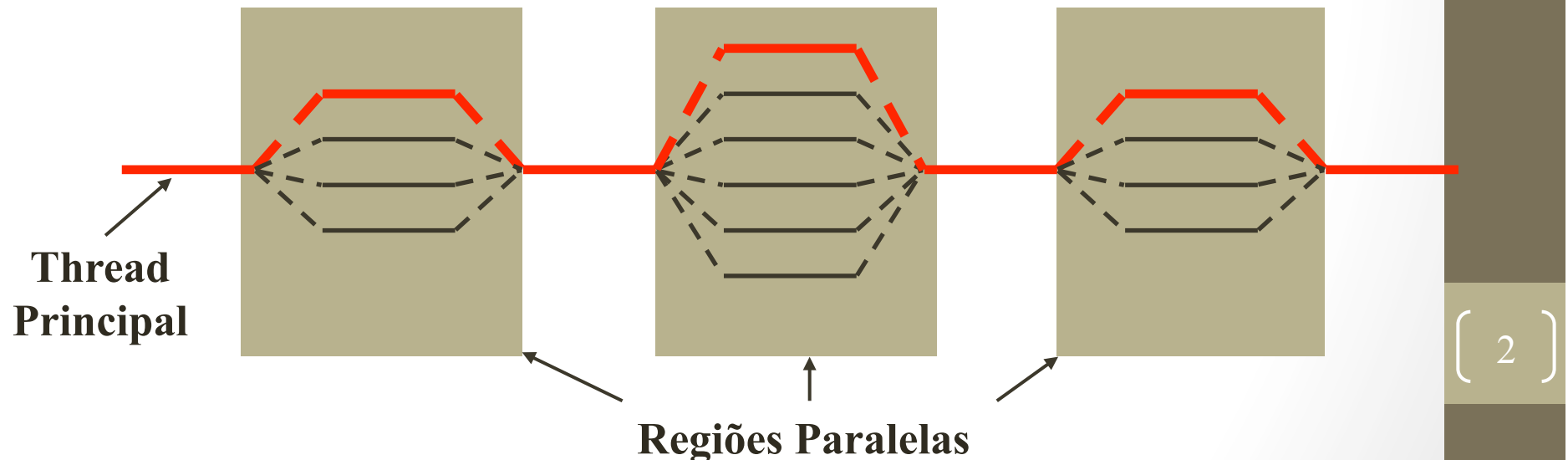


# Introdução a programação paralela com OpenMP

Física Computacional  
2012.2

# Modelo de Programação

- Thread principal divide-se em um conjunto de threads quando necessário
  - Gerenciado de maneira quase transparente
  -
- Paralelismo é adicionado de maneira incremental: Um programa sequencial evolui para um programa paralelo.



# OpenMP: Biblioteca com mais de 20 rotinas

- Rotinas de acesso ao ambiente:
  - Modificar/verificar número de threads
    - `omp_[set|get]_num_threads()`
    - `omp_get_thread_num()`
    - `omp_get_max_threads()`
  - Está em uma região paralela?
    - `omp_in_parallel()`
  - Quantos processadores tem o sistema?
    - `omp_get_num_procs()`
  - Travamentos explícitos
    - `omp_[set|unset]_lock()`
  - E muito mais...

# Alguns detalhes

- A maioria dos construtores são formadas por diretivas de compilação

- C e C++:

`#pragma omp construct [clause [clause]...]`

- Fortran:

`C$OMP construct [clause [clause]...]`

`!$OMP construct [clause [clause]...]`

`*$OMP construct [clause [clause]...]`

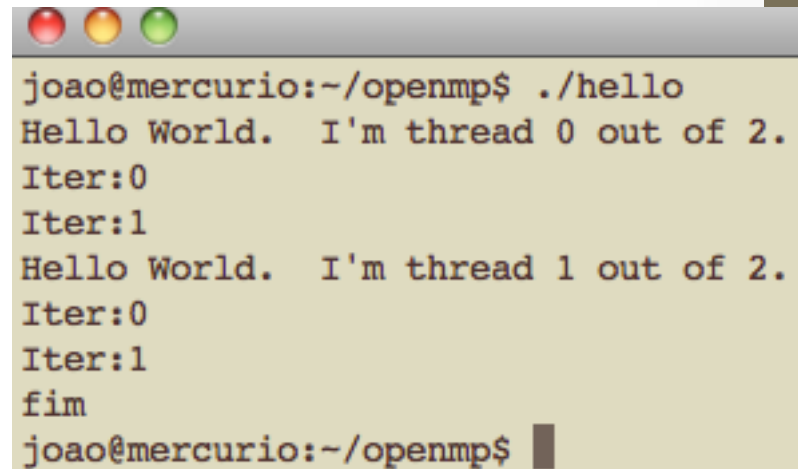
# Exemplo: Hello World

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

int main() {
#pragma omp parallel
{
    int myId = omp_get_thread_num();
    int nThreads = omp_get_num_threads();

    printf("Hello World. I'm thread %d out of %d.\n", myId, nThreads);
    for( int i=0; i<2 ;i++ )
        printf("Iter:%d\n",i);
}
printf("GoodBye World\n");
}
```

- Compile com
- `g++ -fopenmp -o hello hello.c -lm`
- Máquina com dois núcleos



```
joao@mercurio:~/openmp$ ./hello
Hello World. I'm thread 0 out of 2.
Iter:0
Iter:1
Hello World. I'm thread 1 out of 2.
Iter:0
Iter:1
fim
joao@mercurio:~/openmp$
```

# Compartilhamento de tarefas

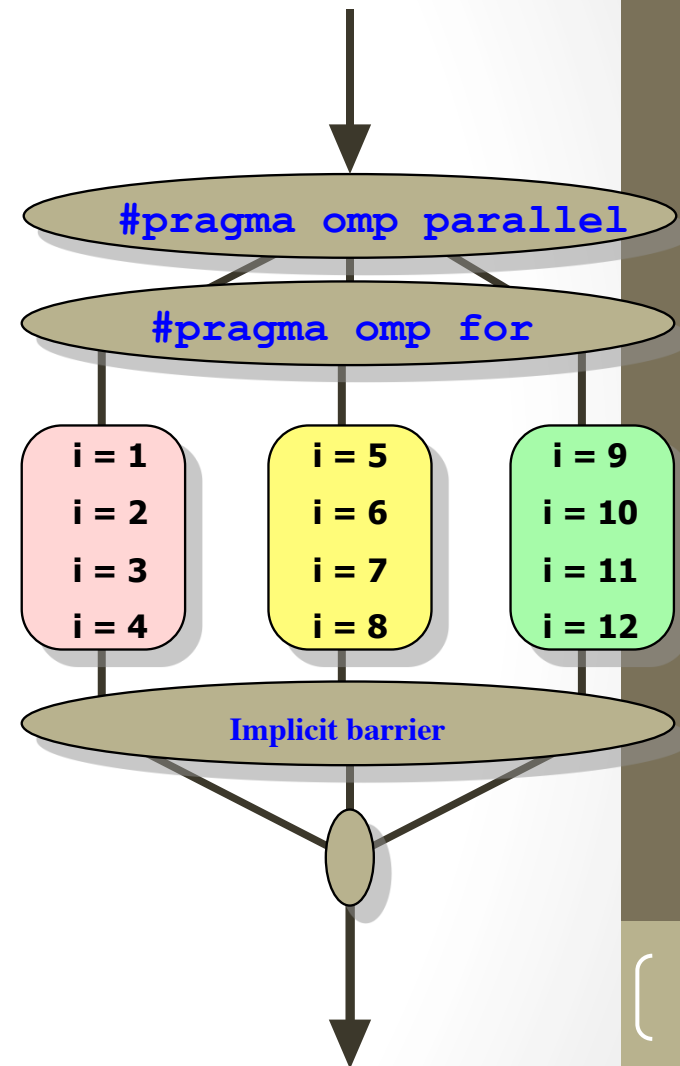
- **Descreve a distribuição de tarefas**
- Três categorias:
  - Contrutor “omp for”
  - Construtor “omp sections”
  - Contrutor “omp task”

Automaticamente divide o trabalho entre as threads

# “omp for”

```
// assume N=12
#pragma omp parallel
#pragma omp for
    for(i = 1, i < N+1, i++)
        c[i] = a[i] + b[i];
```

- Threads executam iterações diferentes
- Threads devem aguardar no final do construtor



# Combinando Construtores

- Os dois códigos abaixo são equivalentes

```
#pragma omp parallel
{
    #pragma omp for
    for ( int i=0;i< MAX; i++) {

        res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (int i=0;i< MAX; i++) {
        res[i] = huge();
    }
```



# Cláusula private

- Reproduz a variável para cada thread
  - Variáveis não são inicializadas
  - Qualquer valor for a da região paralela é indefinida
  - Declarando uma variável como private, significa que cada thread irá ter uma cópia dela.
    - O valor da variável x na thread 1 é diferente do valor da variável x da thread 2.

```
void* work(float* c, int N) {  
    float x, y; int i;  
    #pragma omp parallel for private(x,y)  
    for(i=0; i<N; i++) {  
        x = a[i]; y = b[i];  
        c[i] = x + y;  
    }  
}
```

# A cláusula `schedule`

- Afeta como os loops são distribuídos entre as threads

## `schedule(static [, chunk])`

- Blocos de iterações de tamanho “chunk” de threads
- Distribuição “Round robin”
- Baixo “overhead” Pode causar desbalanceamento de carga

## `schedule(dynamic [, chunk])`

- Quando uma thread acaba seu trabalho, requisita mais
- Alto “overhead”, pode reduzir o desbalanceamento de carga

## `schedule(guided[, chunk])`

- Schedule dinâmico com bloco grande
- Tamanho dos bloco diminui até valores não menores que “chunk”.

# Exemplo schedule

```
#pragma omp parallel for schedule (static, 8)
for( int i = start; i <= end; i += 2 )
{
    if ( TestForPrime(i) ) gPrimesFound++;
}
```

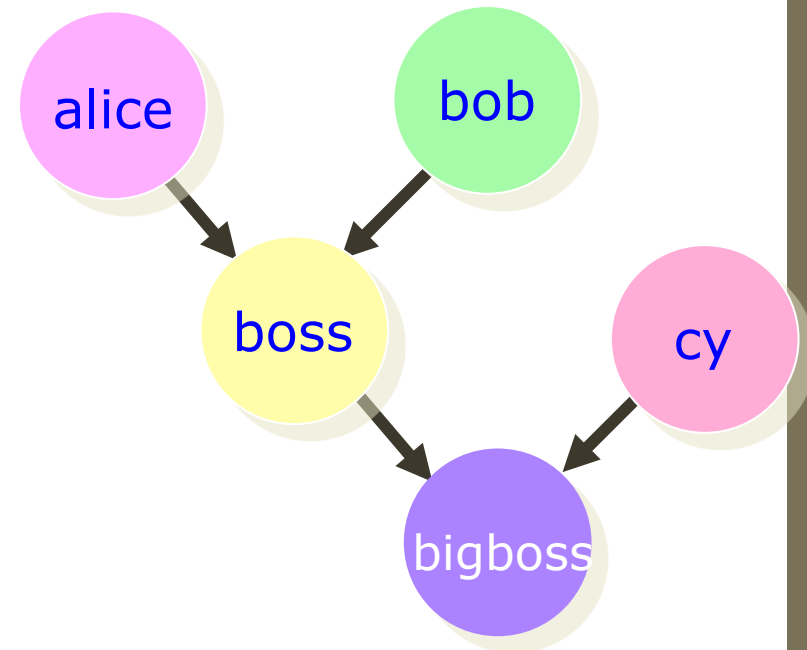
- Iterações são divididas em “pedaços” de tamanho 8.
- Se start = 3, então o primeiro pedaço é:

**i**={3,5,7,9,11,13,15,17}

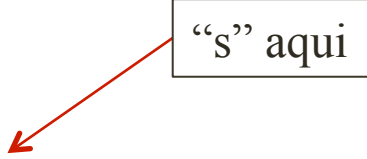
# Paralelismo de Funções


```
a = alice();  
b = bob();  
s = boss(a, b);  
c = cy();  
printf ("%6.2f\n", bigboss(s,c));
```

alice, bob, e cy  
podem ser calculados  
em paralelo



# omp sections

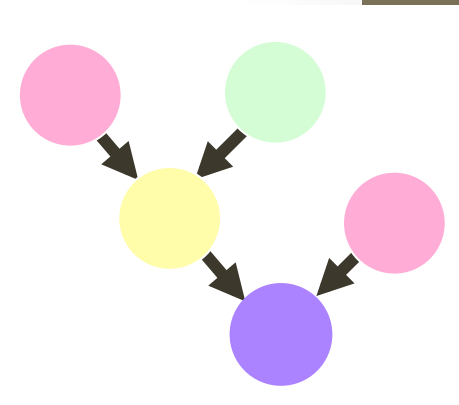
- **#pragma omp sections** 
- Deve estar dentro de uma região paralela.
- Precede um bloco contendo N sub-blocos de código que podem se executados por N threads.
- Engloba cada “omp section”

- **#pragma omp section** 
- Precede cada sub-bloco de código
- Segmentos são distribuídos entre as threads disponíveis.

# Paralelismo funcional usando `omp sections`

```
#pragma omp parallel sections
{
    #pragma omp section
        double a = alice();
    #pragma omp section
        double b = bob();
    #pragma omp section
        double c = cy();
}

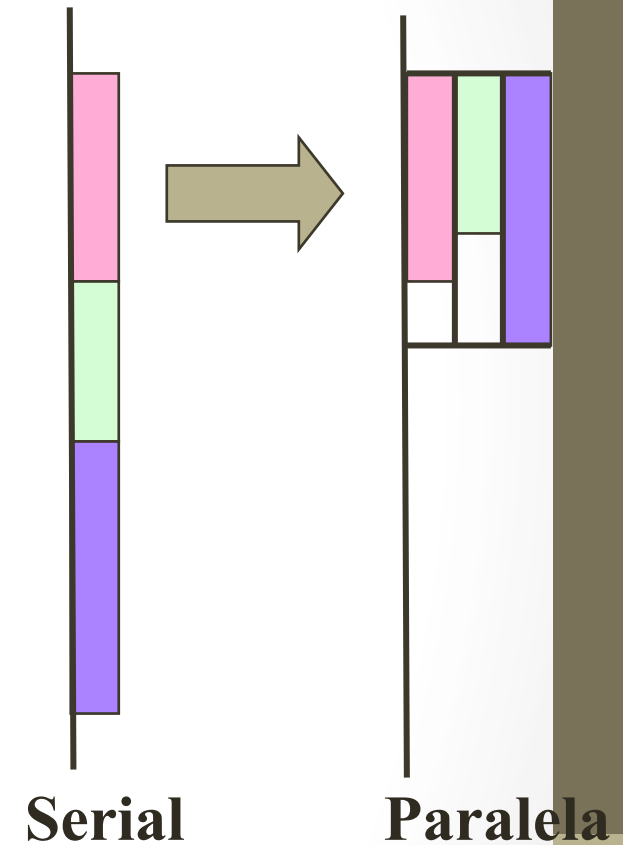
double s = boss(a, b);
printf ("%6.2f\n", bigboss(s,c));
```



# Vantagens de seções paralelas

- Seções independentes são executadas paralelamente. Redução do tempo de execução.

```
#pragma omp parallel sections
{
  #pragma omp section
    phase1();
  #pragma omp section
    phase2();
  #pragma omp section
    phase3();
}
```



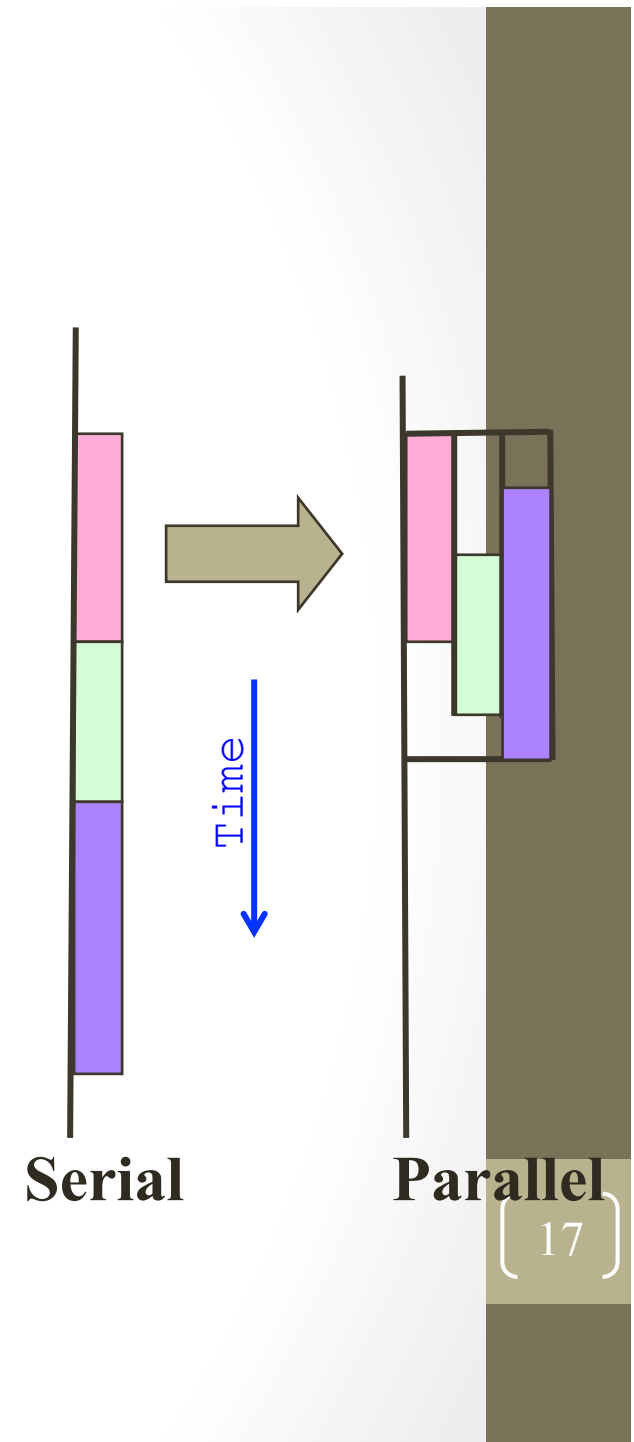
# Nova funcionalidade do OpenMP

- Tasks – Maior mudança na versão 3.0 do OpenMP
- Permite paralelização de problemas irregulares.
  - Loops sem bordas
  - Algoritmos recursivos
  - Produtor/Consumidor



# O Que são tasks?

- Tasks são unidades independentes
- Cada thread faz uma tarefa específica
- Uma tarefa pode ser executada imediatamente ou depois
  - O sistema decide, durante a execução
- Tasks são compostas de
  - **Código para execução**
  - **dados**
  - **Variáveis de controle interno (ICV)**



# Exemplo simples de Tasks

```
#pragma omp parallel
// considere 8 threads
{
    #pragma omp single private(p)
    {
        // algum código aqui...
        node *p = head_of_list;
        while( p != end_of_list ) {
            #pragma omp task
            {
                processwork(p);
            }
            p = p->next;
        }
    }
}
```

Um conjunto de 8 threads criadas

Somente uma thread executa o loop

A thread que executa o loop, cria uma tarefa para cada instância de processwork()

# Construtor de Tasks –

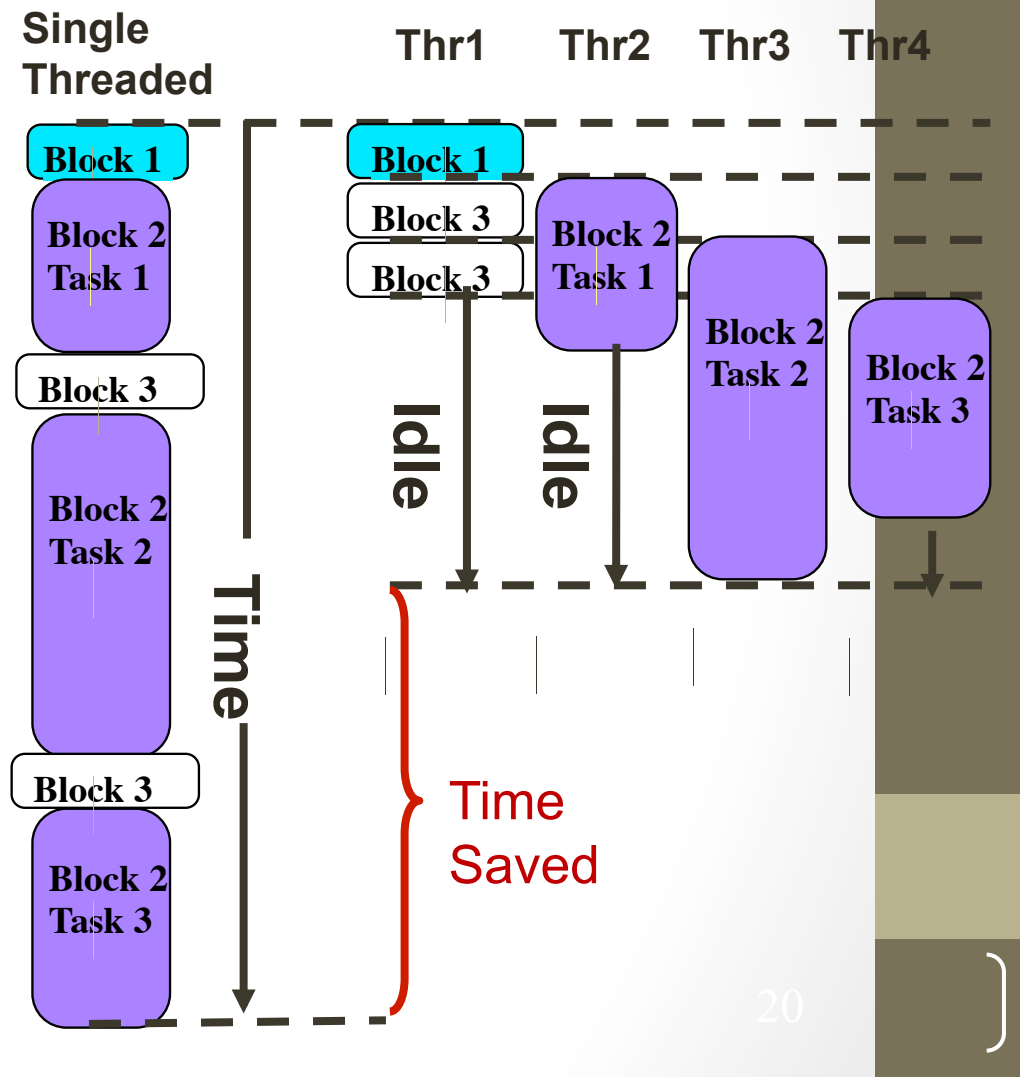
- Um conjunto de threads é criado no construtor `omp parallel`
- Uma thread é selecionada para executar o loop, considere thread “L”
- Thread L opera o loop while, cria tarefas e obtém o próximo ponteiro
- A cada vez que a thread L cruza o construtor `omp task` cria uma nova tarefa
- Cada tarefa executa sua própria thread.

```
#pragma omp parallel
{
    #pragma omp single
    { // block 1
        node *p = head_of_list;
        while (p) { //block 2
            #pragma omp task private(p)
            process(p);
            p = p->next; //block 3
        }
    }
}
```

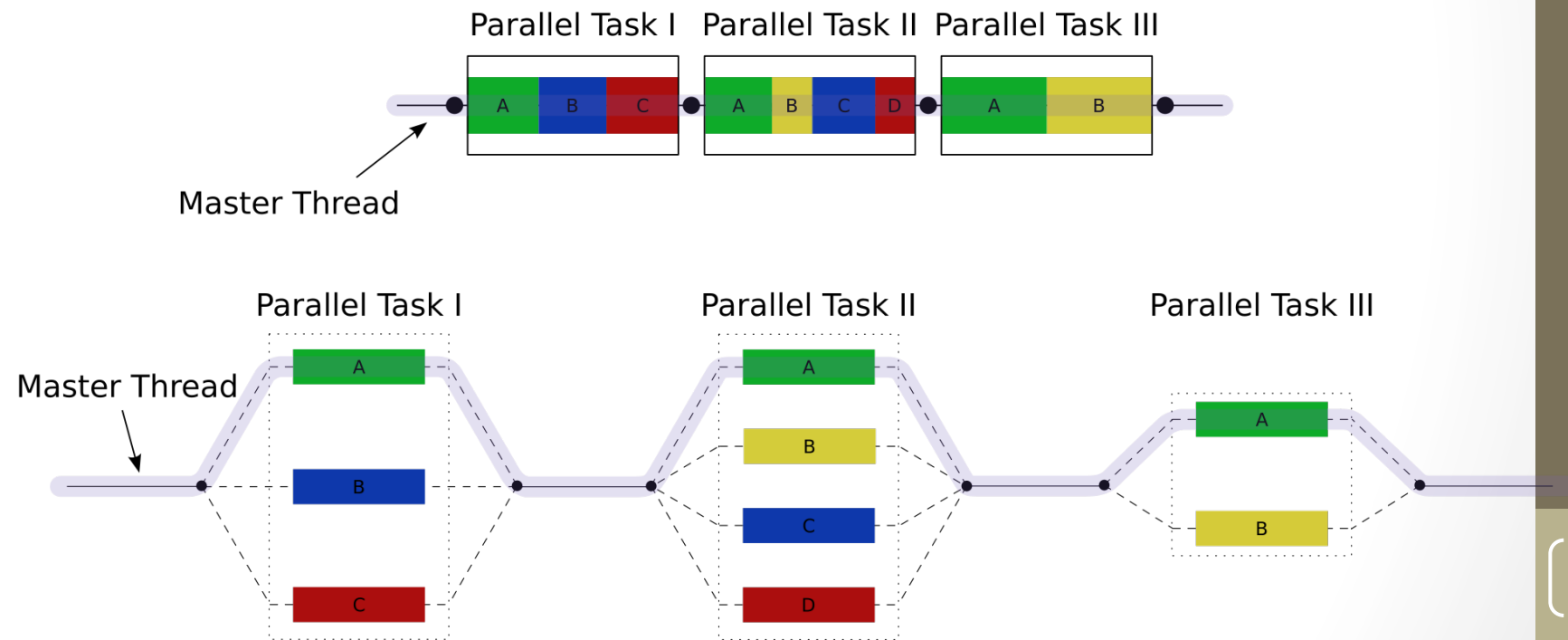
# Por que são úteis?

Têm potencial para paralelizar padrões irregulares e chamadas recursivas

```
#pragma omp parallel
{
    #pragma omp single
    { // block 1
        node *p = head_of_list;
        while (p) { //block 2
            #pragma omp task private
            (p)
            process(p);
            p = p->next; //block 3
        }
    }
}
```



# Tasks: Perspectiva



# Barreiras implícitas

- Barreiras implícitas
  - **parallel** – barreira necessária, não pode ser removida
  - **for**
  - **single**
- Barreiras desnecessárias, podem ser removidas com a cláusula **nowait**
  - Aplicável a **nowait** :
    - **for**
    - **single**

# Nowait

```
#pragma omp for nowait  
for(...)  
{...};
```

```
#pragma single nowait  
{ [...] }
```

- Utilize quando não há necessidade de espera entre os loops

```
#pragma omp for schedule(dynamic,1) nowait  
for(int i=0; i<n; i++)  
    a[i] = bigFunc1(i);  
  
#pragma omp for schedule(dynamic,1)  
for(int j=0; j<m; j++)  
    b[j] = bigFunc2(j);
```

# Barreiras

- Sincronização explícita
- Cada thread irá aguardar na barreira até que todas as threads cheguem nele.

```
#pragma omp parallel shared(A, B, C)
{
    DoSomeWork(A,B); // Processed A into B
#pragma omp barrier
    DoSomeWork(B,C); // Processed B into C
}
```



# Operações atômicas

- Atualizações únicas em variáveis

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

# Exemplo: produto escalar

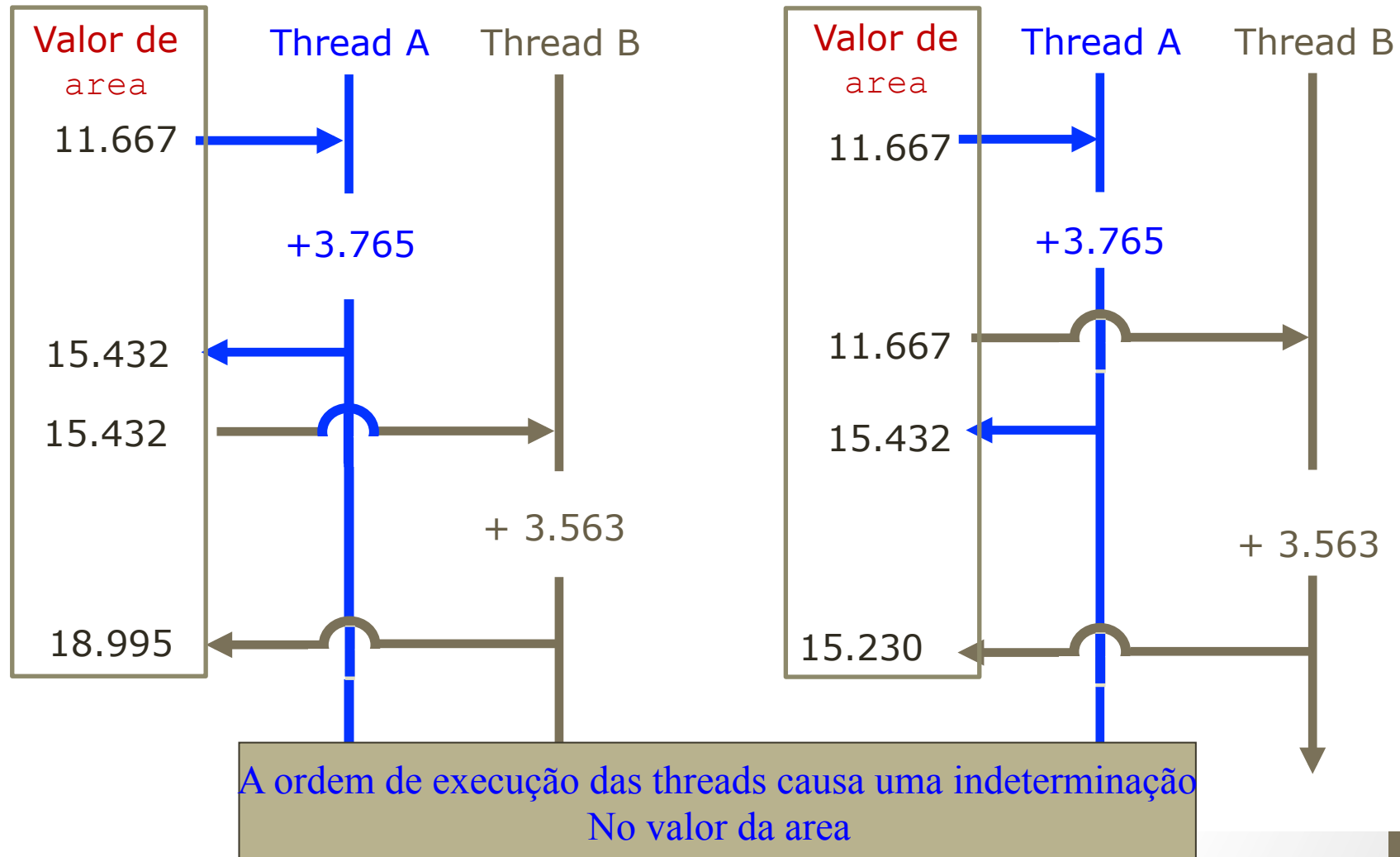
```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

O Que está errado?

# Condição de corrida

- Condição de corrida é um comportamento não determinado que ocorre quando duas ou mais threads alteram/acessam uma variável compartilhada.
- Por exemplo, suponha que duas threads estejam executando a mesma instrução `area += 4.0 / (1.0 + x*x);`

# Dois possíveis cenários



# Proteção dos dados compartilhados

- Devemos proteger o acesso ao dados compartilhados que são modificados.

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

# Cláusula Reduction

`reduction (op : list)`

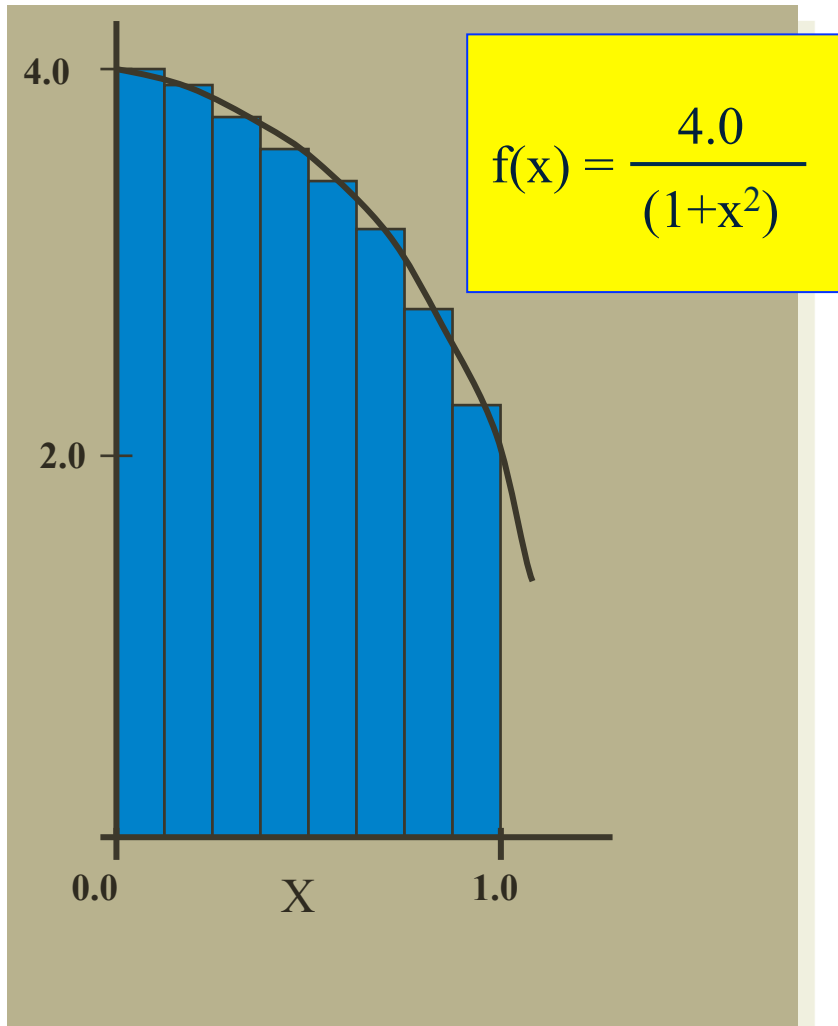
- A lista de variáveis deve ser declarada como “shared”
- Dentro da região paralela:
  - Uma cópia provada de cada variável da lista é criada e inicializada, dependendo da operação.
  - As cópias são atualizadas localmente pelas threads
  - No final, o contrutor combina as variáveis privadas

# Example de redução

```
#pragma omp parallel for reduction(+:sum)
for(i=0; i<N; i++) {
    sum += a[i] * b[i];
}
```

- Cada thread tem sua própria variável sum
- Todas as cópias locais de sum são somadas no final e armazenadas numa variável compartilhada.

# Integração numérica



$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \text{pi}$$

```
static long num_steps=100000;
double step, pi;

void main() {
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```



# Integração numérica

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

int main(int argc, char* argv[])
{
    int num_steps = atoi(argv[1]);
    double step = 1./((double)(num_steps));
    double sum;

    #pragma omp parallel for reduction(+:sum)
    {
        for(int i=0; i<num_steps; i++) {
            double x = (i + .5)*step;
            sum += 4.0/(1. + x*x);
        }
    }

    double my_pi = sum*step;
    printf("Pi = %f\n", my_pi);

    return 0;
}
```

# C/C++ Operações de Redução

Operador	Valor inicial
+	0
*	1
-	0
^	0

Operador	Valor Inicial
&	~0
	0
&&	1
	0