

# *OpenMP*

Arquitetura de Computadores  
Lic. em Engenharia Informática

# O que é o OpenMP

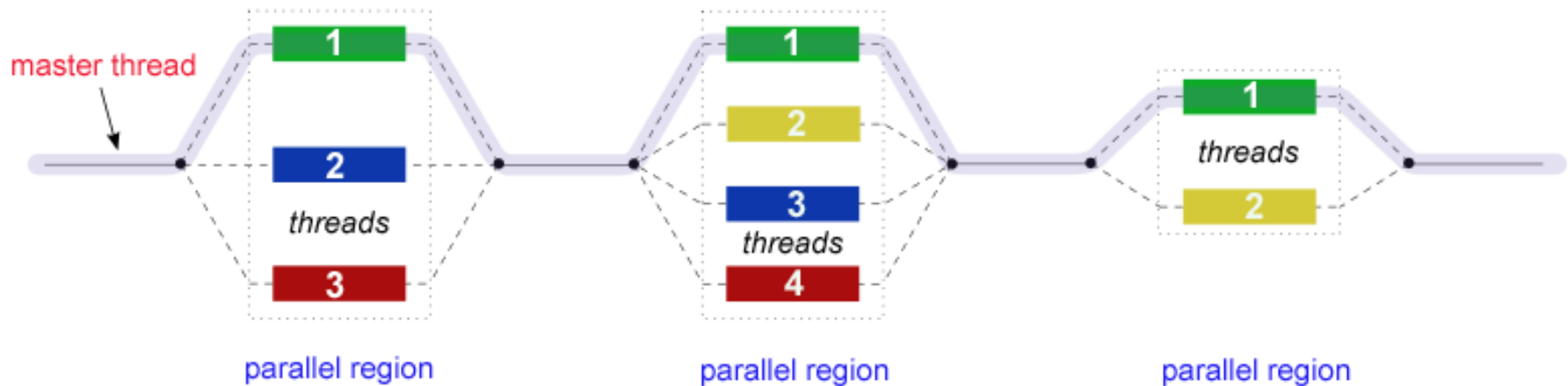
## *Open Multi Processing*

- **API** para expressar **paralelismo *multi-threaded*** e de **memória partilhada**
- standard mantido pelo *OpenMP Architecture Review Board*
- actualmente (2014) na versão 4.0.1
- Objectivos:
  - normalização (*standard*)
  - portabilidade
  - fácil utilização

# Modelo de execução

- Criação **explícita** de blocos paralelos de código executados por um grupo (*team*) de *threads*

## Modelo Fork & Join



- No final de cada bloco:
  - todas as *threads* sincronizam (barreira implícita)
  - todas as *threads* excepto a principal deixam de existir

# Modelo de execução

```
printf("program begin\n");
```

```
N = 1000;
```

```
#pragma omp parallel for  
for (i=0; i<N; i++)  
    A[i] = B[i] + C[i];
```

```
M = 500;
```

```
#pragma omp parallel for  
for (j=0; j<M; j++)  
    p[j] = q[j] - r[j];
```

```
printf("program done\n");
```

Sequencial

Paralelo

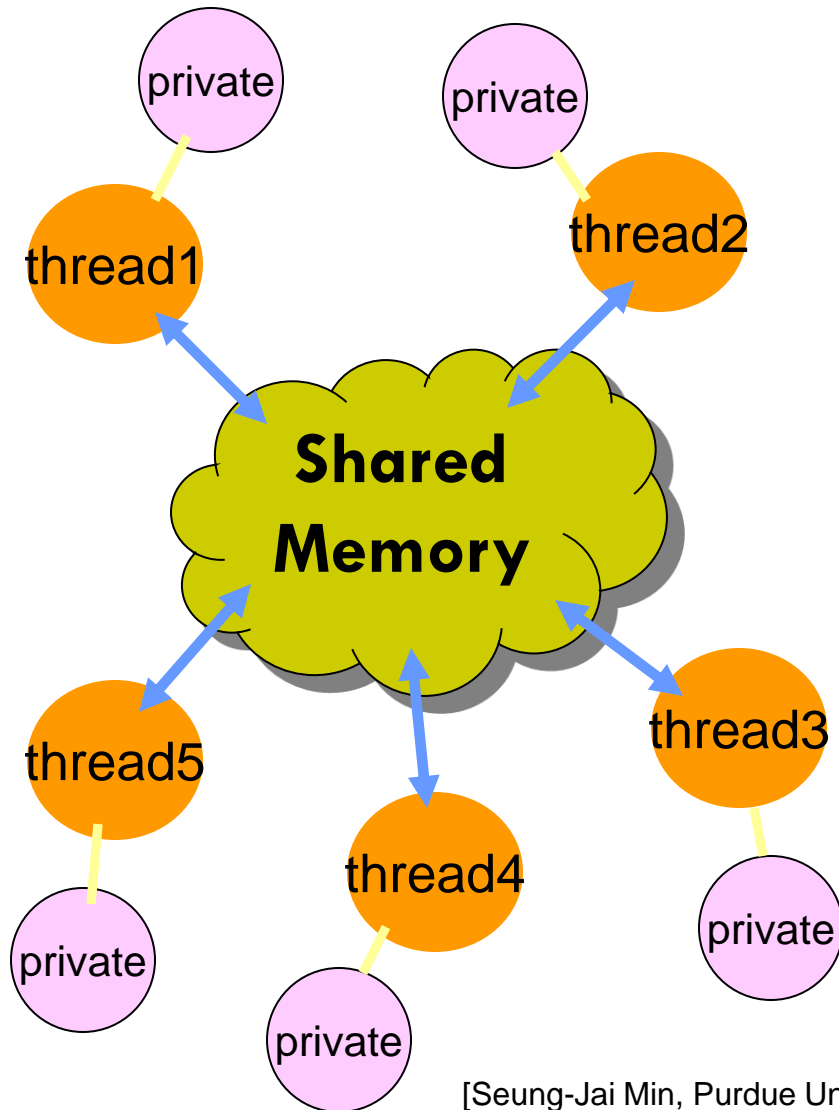
Sequencial

Paralelo

Sequencial

[Seung-Jai Min, Purdue University]

# Modelo de execução



- Os dados podem ser **partilhados** ou **privados**
- Dados partilhados são acessíveis a todas as *threads*
- Dados privados são acessíveis apenas à *thread* que os possui

[Seung-Jai Min, Purdue University]

# Modelo de execução

- Por defeito os dados são partilhados
- As variáveis globais a um bloco paralelo são partilhadas
- Variáveis privadas:
  - declaradas dentro de um bloco paralelo
  - explicitamente marcadas como privadas
  - índices de alguns ciclos for

# Directivas

- Paralelismo especificado usando directivas embebidas no código

```
#pragma omp <nome directiva> [cláusula, ...]
```

- Cada directiva aplica-se ao bloco de instruções que se lhe segue

```
#pragma omp parallel
{
    ... // bloco paralelo
}
... // bloco sequencial
```

- O compilador ignora as directivas se não for usada a opção que activa o OpenMP. Exemplo:

```
gcc -fopenmp <filename>
icc -openmp <filename>
```

# Funções

#include <omp.h>	
Função	Descrição
<code>int omp_get_thread_num (void)</code>	Devolve ID da thread
<code>int omp_get_num_threads (void)</code>	Devolve número de threads actualmente existentes num bloco paralelo
<code>void omp_set_num_threads (int)</code>	Estabelece número de threads a ser criadas no próximo bloco paralelo
<code>int omp_get_num_procs (void)</code>	Devolve número de processadores disponíveis para o programa
<code>double omp_get_wtime (void)</code>	Devolve um <i>time stamp</i> em segundos
... e muitas mais ...	



# directiva `parallel`

```
#pragma omp parallel
{
    ... // bloco paralelo
}
```

- cria um grupo (*team*) de N threads
- cada uma destas threads executa independentemente o bloco paralelo
- no fim do bloco existe uma barreira (sincronização) implícita: a *thread* principal só continua depois de todas as outras também terem chegado ao fim do bloco

# directiva parallel

shared by default!

```
char *s = "Hello, world!";  
#pragma omp parallel  
{  
    printf("%s\n",s);  
}  
printf("program done\n");
```

```
> ./prog  
Hello, world!  
Hello, world!  
program done  
> _
```

# directiva `parallel`

- Quantas *threads* há num grupo?

1. cláusula `num_threads(int)`

```
#pragma omp parallel num_threads(64)
```

2. função `omp_set_num_threads(int)`

```
#omp_set_num_threads(12);  
#pragma omp parallel
```

3. variável de ambiente `OMP_NUM_THREADS`

```
> export OMP_NUM_THREADS=8
```

4. dependente da implementação

normalmente igual ao número de processadores disponível para o programa

## directiva `parallel` - `if`

- cláusula `if` (condição)
  - se condição  $\neq 0$  então o grupo de threads é criado
  - se condição  $= 0$  então execução sequencial

```
int i;  
scanf ("%d", &i);  
#pragma omp parallel if(i)  
{  
    printf("Hello!\n");  
}  
printf("program done\n");
```

## directiva `parallel` – *data scope*

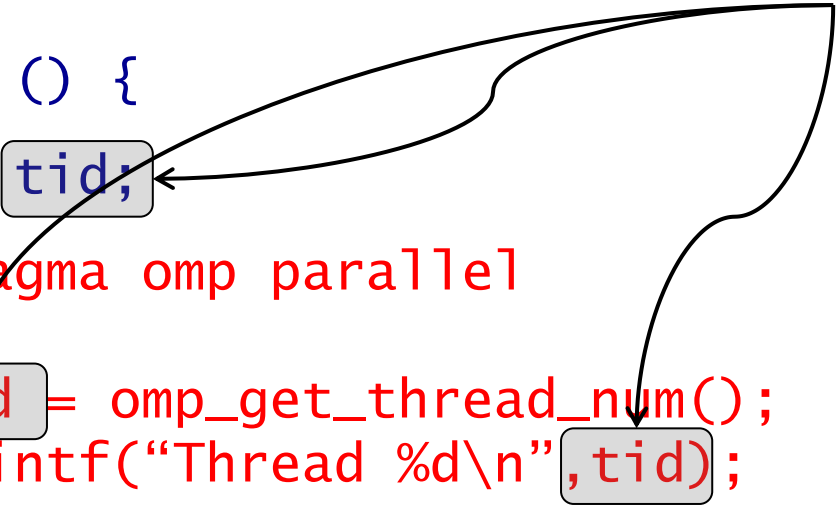
- Variáveis globais a um bloco paralelo são partilhadas por defeito
- São variáveis privadas:
  - locais ao bloco

```
#pragma omp parallel
{  int i;
  ... }
```
  - explicitamente declaradas com a cláusula `private(...)`

```
int x;
#pragma omp parallel private (x)
```
  - os índices de alguns ciclos (ver directiva `for`)
- A cláusula `default(shared | private)` permite alterar a situação por defeito

## directiva `parallel` – *data scope*

```
main () {  
    int tid;  
    #pragma omp parallel  
    {  
        tid = omp_get_thread_num();  
        printf("Thread %d\n", tid);  
    }  
    printf("program done\n");  
}
```



**variável partilhada onde várias *threads* escrevem sem qualquer controlo de acesso**

**Resultado indeterminado**

# directiva `parallel` – *data scope*

## variável privada

cada *thread* tem a sua própria instância local de **`tid`**

```
main () {  
    int tid;  
  
    #pragma omp parallel private (tid)  
    {  
        tid = omp_get_thread_num();  
        printf("Thread %d\n", tid);  
    }  
  
    printf("program done\n");  
}
```

# directiva `parallel` – *data scope*

## variável privada

cada *thread* tem a sua própria instância local de **tid**

```
main () {  
  
    #pragma omp parallel  
    {  
        int tid;  
  
        tid = omp_get_thread_num();  
        printf("Thread %d\n", tid);  
    }  
  
    printf("program done\n");  
}
```

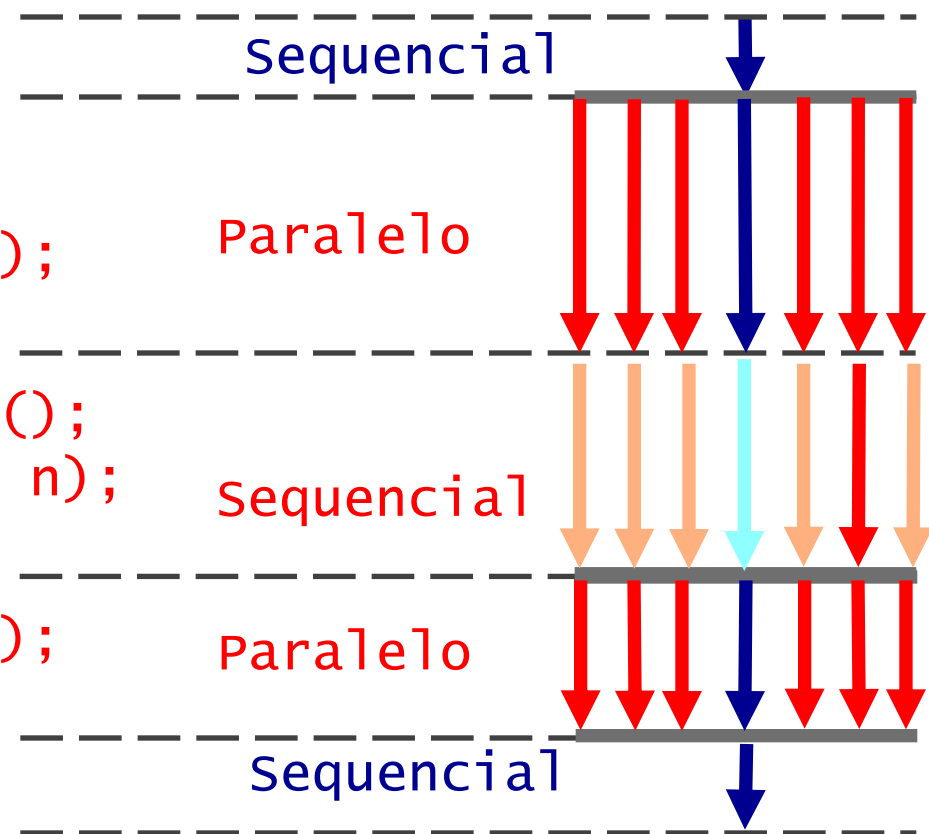


# directiva single

Apenas a primeira *thread* a atingir o bloco single o executa

Todas as *threads* sincronizam no fim do bloco (barreira implícita)

```
int n;  
#pragma omp parallel  
{ int tid;  
  tid = omp_get_thread_num();  
  
#pragma omp single  
  { n = omp_get_num_threads();  
    printf ("%d threads\n", n);  
  }  
  
  printf (thread %d\n", tid);  
}  
printf("program done\n");
```



# directiva `for`

- distribui as iterações do ciclo `for` pelas *threads* do grupo
- o índice do ciclo é privado a cada thread

```
#pragma omp parallel
{
    printf ("Hello\n");
    #pragma omp for
    for (i=0; i<N; i++) A[i] = B[i] + C[i];
}
```

- deve estar dentro de um bloco `parallel`
- por defeito a distribuição é estática:
  - as iterações são distribuídas pelas *threads* em *chunks* do mesmo tamanho e consecutivos

# directiva `for`

- `for` e `parallel` podem ser combinadas

```
#pragma omp parallel for
```

```
for (i=0; i<N; i++) A[i] = B[i] + C[i];
```

```
#pragma omp parallel for private(j)
```

```
for (i=0; i<N; i++) {  
    for (j=0 ; j<M ; j++)  
        M[i,j] = B[i] + C[j];  
}
```

Só é privado o índice do ciclo imediatamente a seguir à directiva!!

# directiva for - collapse

- `collapse(n)` aplica-se a ciclos `for` aninhados, aumentando o número de iterações disponíveis para execução paralela

```
#pragma omp parallel for collapse (2) private(k)
for (r=0; r<N; r++) {
    for (c=0 ; c<N ; c++) {
        for (k=0 ; k<N ; k++) {
            M[r,c] = A[r,k] * B[k,c];
        }
    }
}
```

Só são privados os índices dos ciclos abrangidos pela directiva!!

# controlo de acessos a dados partilhados

*race conditions*: o resultado depende da ordem de acesso a dados partilhados

```
int x=0;
```

```
#pragma omp parallel num_threads(2)
```

```
  x = x+1;
```

## Caso 1

- . T0: lê x (valor 0)
- . T0: calcula  $0+1 = 1$
- . T1: lê x (valor 0)
- . T0: escreve  $x=1$
- . T1: calcula  $0+1 = 1$
- . T1: escreve  $x=1$

## Caso 2

- . T0: lê x (valor 0)
- . T0: calcula  $0+1 = 1$
- . T0: escreve  $x=1$
- . T1: lê x (valor 1)
- . T1: calcula  $1+1 = 2$
- . T1: escreve  $x=2$

# controlo de acessos a dados partilhados

directiva `critical`: apenas uma *thread* executa esse bloco em cada instante.

Execução **sequencial** desse bloco

```
int x=0;
#pragma omp parallel
  #pragma omp critical
    x = x+1;
```

As regiões críticas sem nome são consideradas a mesma região!

```
int x=0, y=0;
#pragma omp parallel
{
  #pragma omp critical
    x = x+1;
  #pragma omp critical
    y = y+1; }
```

```
int x=0, y=0;
#pragma omp parallel
{
  #pragma omp critical C1
    x = x+1;
  #pragma omp critical C2
    y = y+1; }
```

# controlo de acessos a dados partilhados

directiva `atomic`: garante que um endereço de memória é acedido de forma atómica. Pode ser vista como uma versão leve de `critical`.

Só se aplica a operações simples e ao lado esquerdo da atribuição.

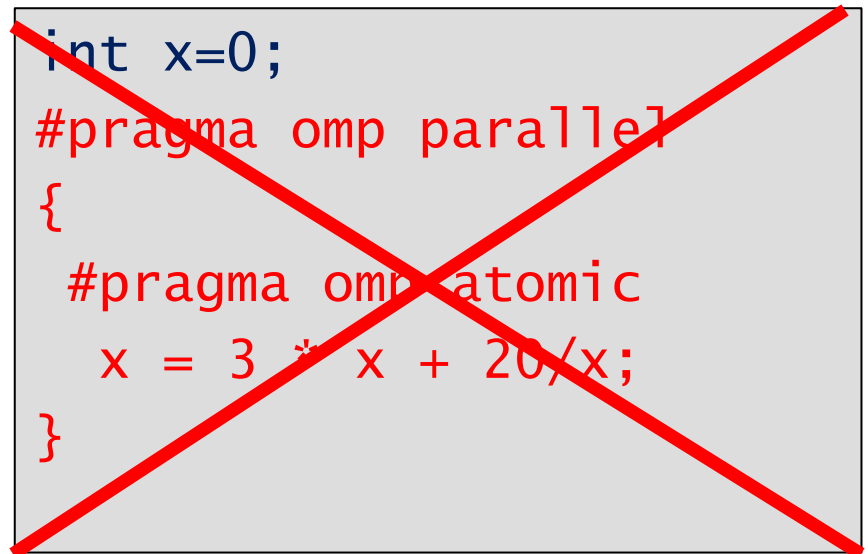
Não garante que o lado direito da atribuição é avaliado de forma atómica.

```
int x=0;
```

```
#pragma omp parallel
```

```
    #pragma omp atomic
```

```
        x += 10;
```



```
int x=0;  
#pragma omp parallel  
{  
    #pragma omp atomic  
    x = 3 * x + 20/x;  
}
```

# redução

- designa-se por **redução** uma operação que processa um conjunto de dados para a partir dele gerar um único valor, exemplo, a soma/máximo/produto de todos os elementos de um vector

```
int a[SIZE];  
... inicializar a[]  
int max=a[0];  
#pragma omp parallel for  
{  
    for (i=0; i< SIZE ; i++)  
        if (a[i]>max) max=a[i]; }
```

max é uma  
variável partilhada



# redução

muito ineficiente  
NA verdade a execução  
é sequencial

```
int a[SIZE];  
... inicializar a[]  
int max=a[0];  
#pragma omp parallel for  
{  
    for (i=0; i< SIZE ; i++)  
        #pragma omp critical  
        if (a[i]>max) max=a[i];  
}
```

```
int a[SIZE];  
... inicializar a[]  
int max=a[0];  
#pragma omp parallel  
{int max1 = a[0];  
    #pragma omp for  
    for (i=0; i< SIZE ; i++)  
        if (a[i]>max1) max1=a[i];  
    #pragma omp critical  
    if (max1 > max) max = max1;  
}
```

# redução

- A redução é tão comum que o OpenMP inclui uma cláusula específica (mas não para o máximo)

```
int a[SIZE];  
... inicializar a[]  
int sum=0;  
#pragma omp parallel for reduction (+:sum)  
{  
    for (i=0; i< SIZE ; i++)  
        sum += a[i]; }  
}
```

# redução

- A cláusula `reduction` aplica-se apenas a operações associativas.

## C / C++

**x = x op expr**

**x = expr op x (excepto subtracção)**

**x binop= expr**

**x++; ++x**

**x--; --x**

**x** – variável escalar

**expr** – expressão escalar que não refere x

**op** – +, \*, -, /, &, ^, |, &&, ||

**binop** - +, \*, -, /, &, ^, |

# redução

- As operações sobre operandos (double) não são associativas:

```
float a[SIZE], sum=0.;  
... inicializar a[]  
#pragma omp parallel for re  
{  
    for (i=0; i< SIZE ; i++)  
        sum += a[i]; }  
printf ("sum= %.1f\n", sum)
```

```
> ./prog  
sum= 1233458.0  
  
> ./prog  
sum= 1233463.0  
  
> ./prog  
sum= 1233457.0
```

# escalonamento

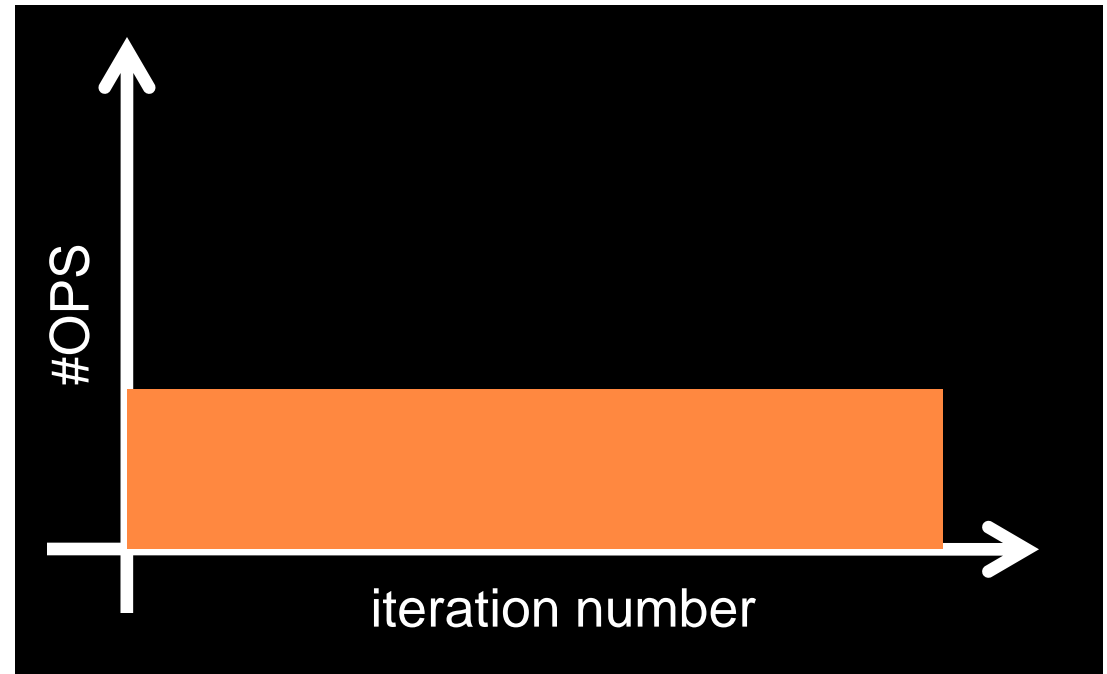
- Problema A

```
#pragma omp parallel for  
{  
    for (i=0; i< SIZE ; i++)  
        b[i] = a[i]*a[i] + 10. / a[i]; }
```

- Problema B

```
#pragma omp parallel for private (j)  
{  
    for (i=0; i< SIZE ; i++) {  
        b[i] = 1.F;  
        for (j=2 ; j<= i ; j++)  
            b[i] *= j; } }
```

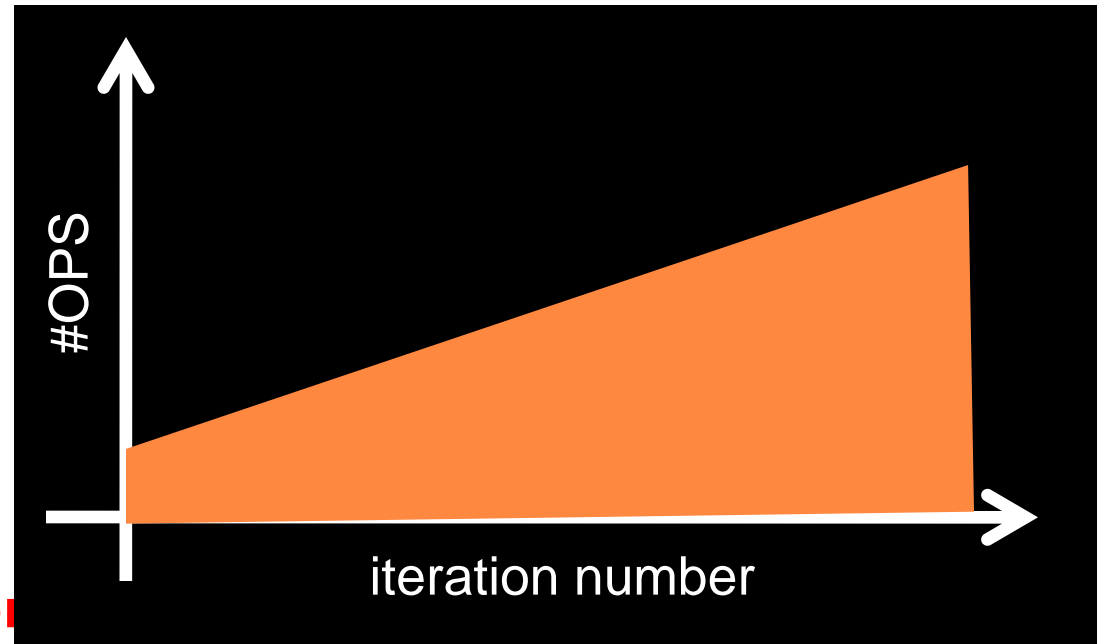
# escalonamento



- Problema A

```
#pragma omp parallel for  
{  
  for (i=0; i< SIZE ; i++)  
    b[i] = a[i]*a[i] + 10. / a[i]; }
```

# escalonamento



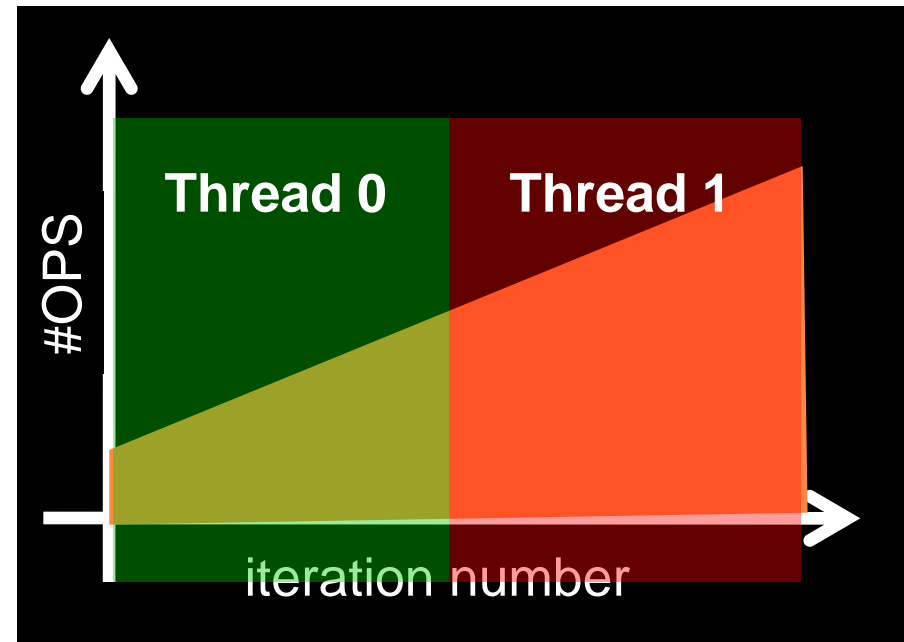
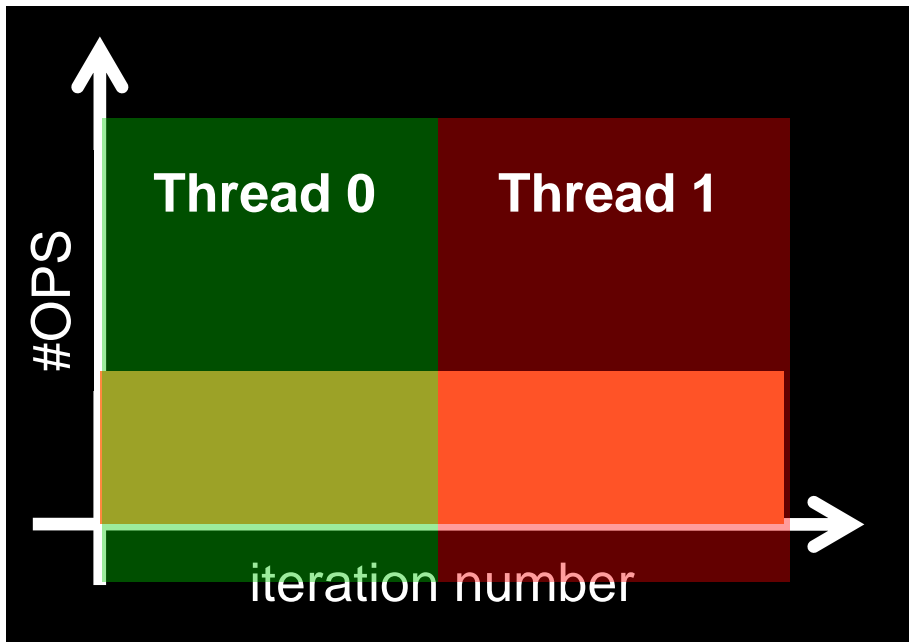
- Problema B

```
#pragma omp parallel for  
{  
  for (i=0; i< SIZE ; i++) {  
    b[i] = 1.F;  
    for (j=2 ; j<= i ; j++)  
      b[i] *= j; } }
```

# escalonamento

## #pragma omp parallel for schedule(static)

- O ciclo é dividido em *#threads* segmentos (*chunks*), todos com o mesmo número de iterações

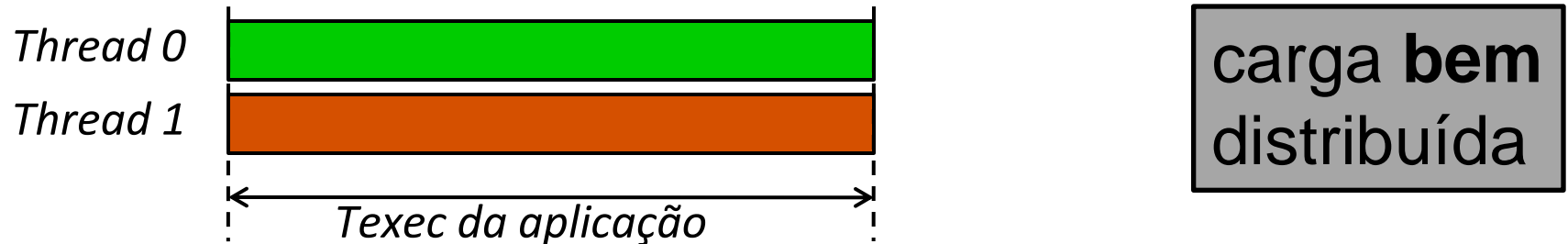




# escalonamento

**#pragma omp parallel for schedule(static)**

**1 – carga uniforme para todas as iterações**



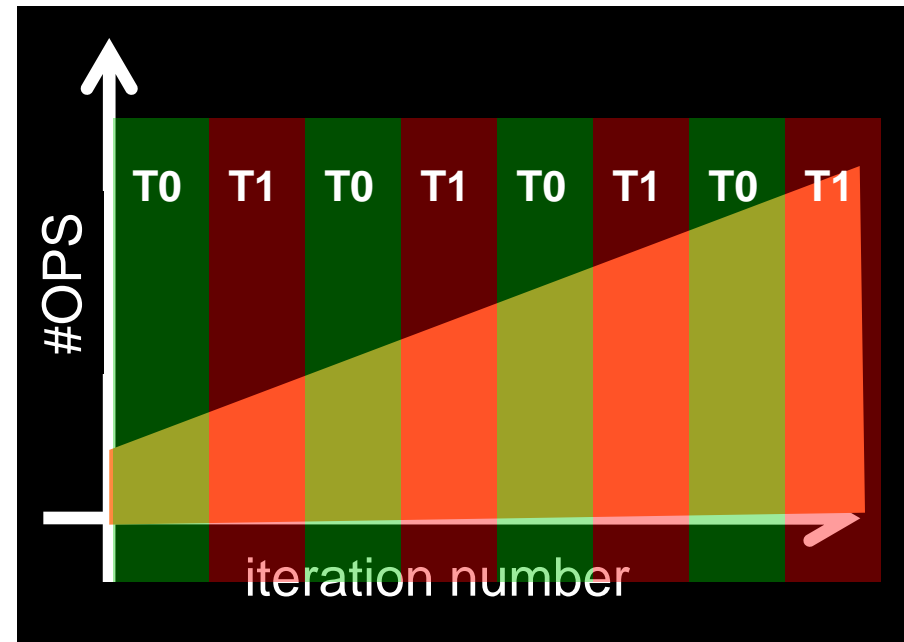
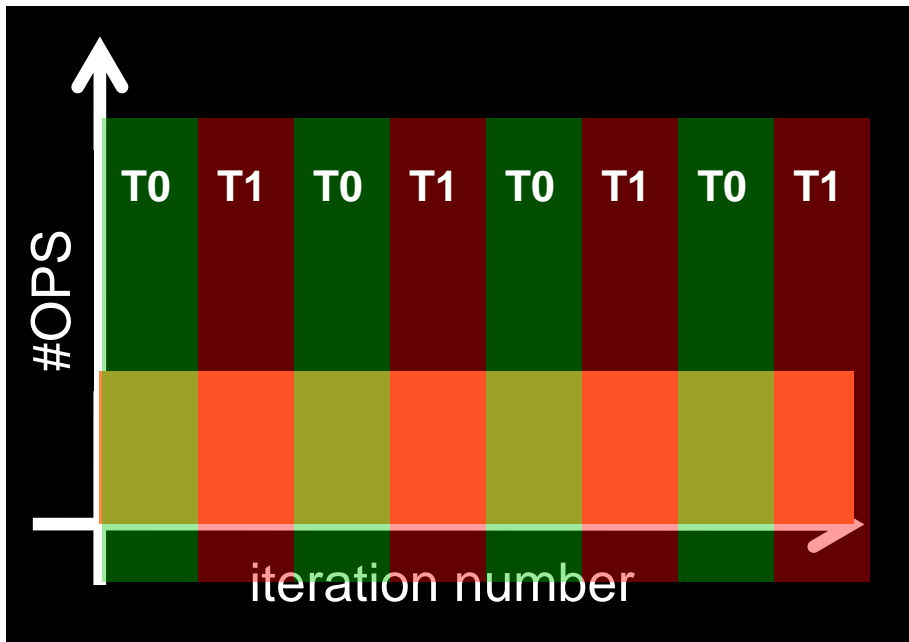
**2 – carga variável para diferentes iterações**



# escalonamento

## #pragma omp parallel for schedule(dynamic)

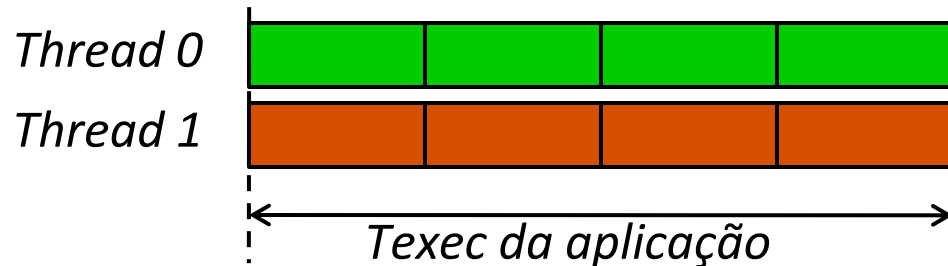
- O ciclo é dividido em muitos segmentos (*chunks*), todos com o mesmo número de iterações, e distribuídos pelas *threads* a pedido



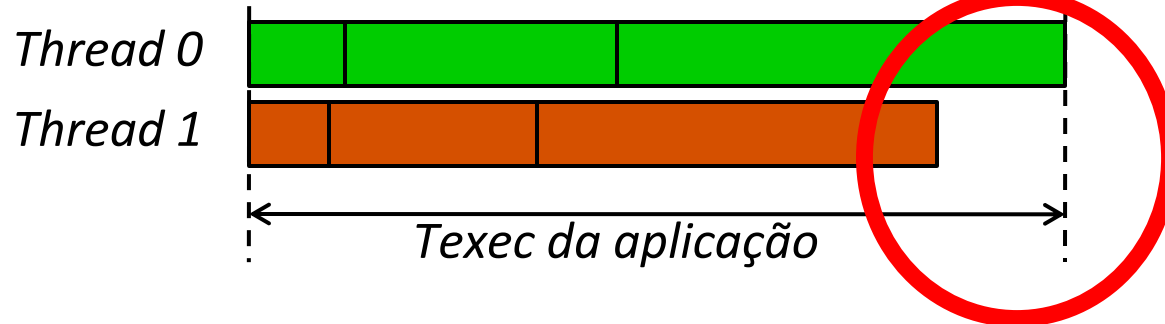
# escalonamento

**#pragma omp parallel for schedule(dynamic)**

**1 – carga uniforme para todas as iterações**



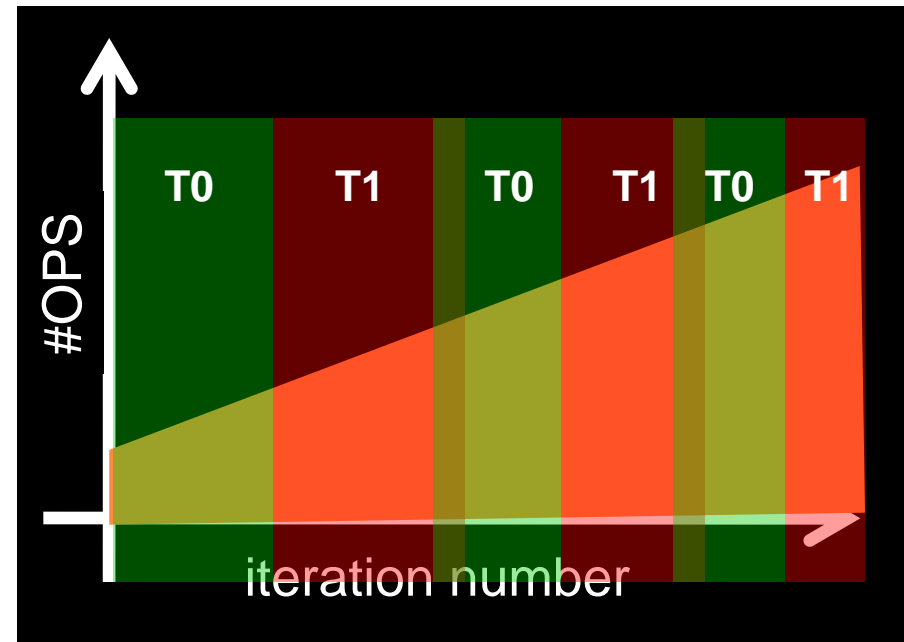
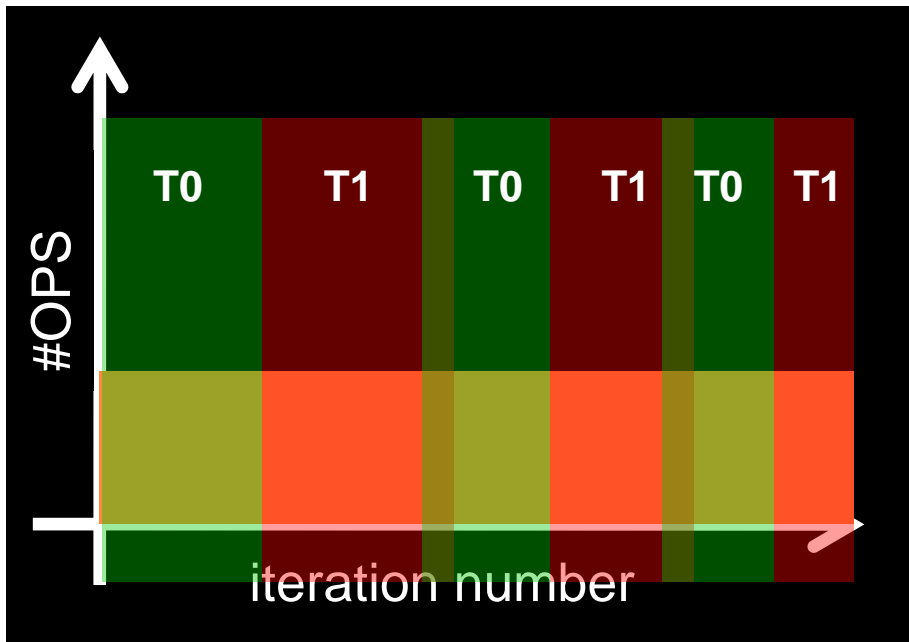
**2 – carga variável para diferentes iterações**



# escalonamento

## #pragma omp parallel for schedule(guided)

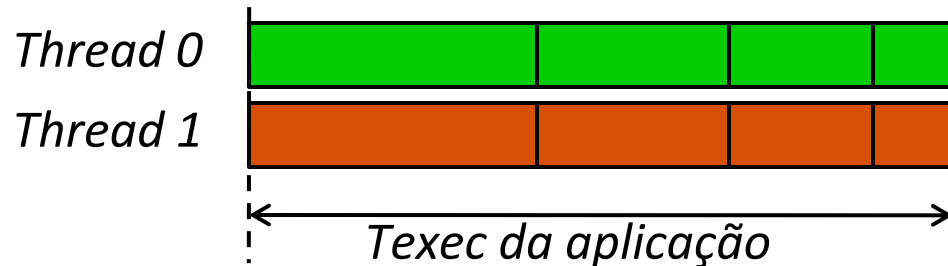
- O ciclo é dividido em muitos segmentos (*chunks*), cada vez com menor número de iterações, e distribuídos pelas *threads* a pedido



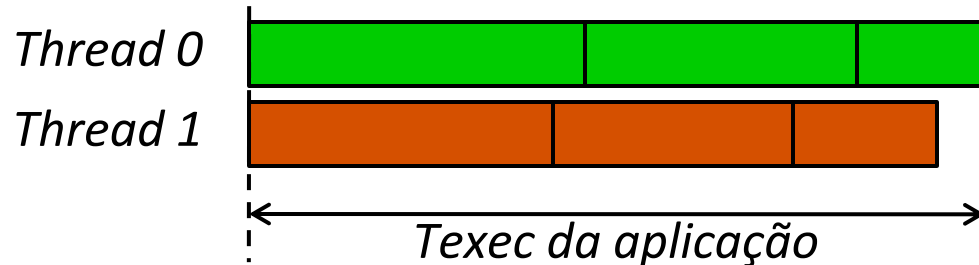
# escalonamento

**#pragma omp parallel for schedule(guided)**

**1 – carga uniforme para todas as iterações**



**2 – carga variável para diferentes iterações**



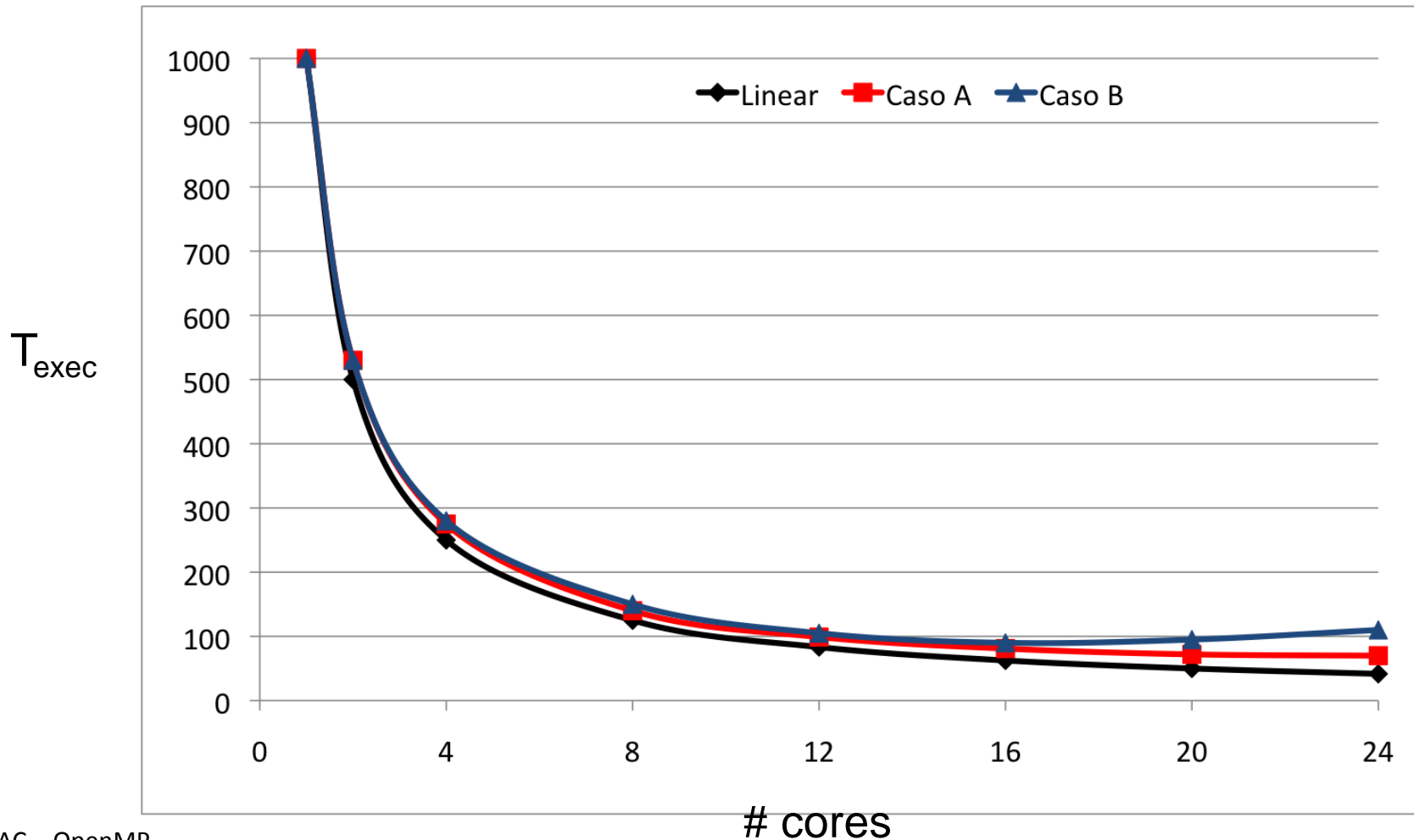
# desempenho

$$T_{exec} = \#I * CPI * f$$

- Com a programação *multithreaded* o número de instruções executadas aumenta, devido à gestão do paralelismo
- CPI diminui, pois várias instruções são executadas simultaneamente

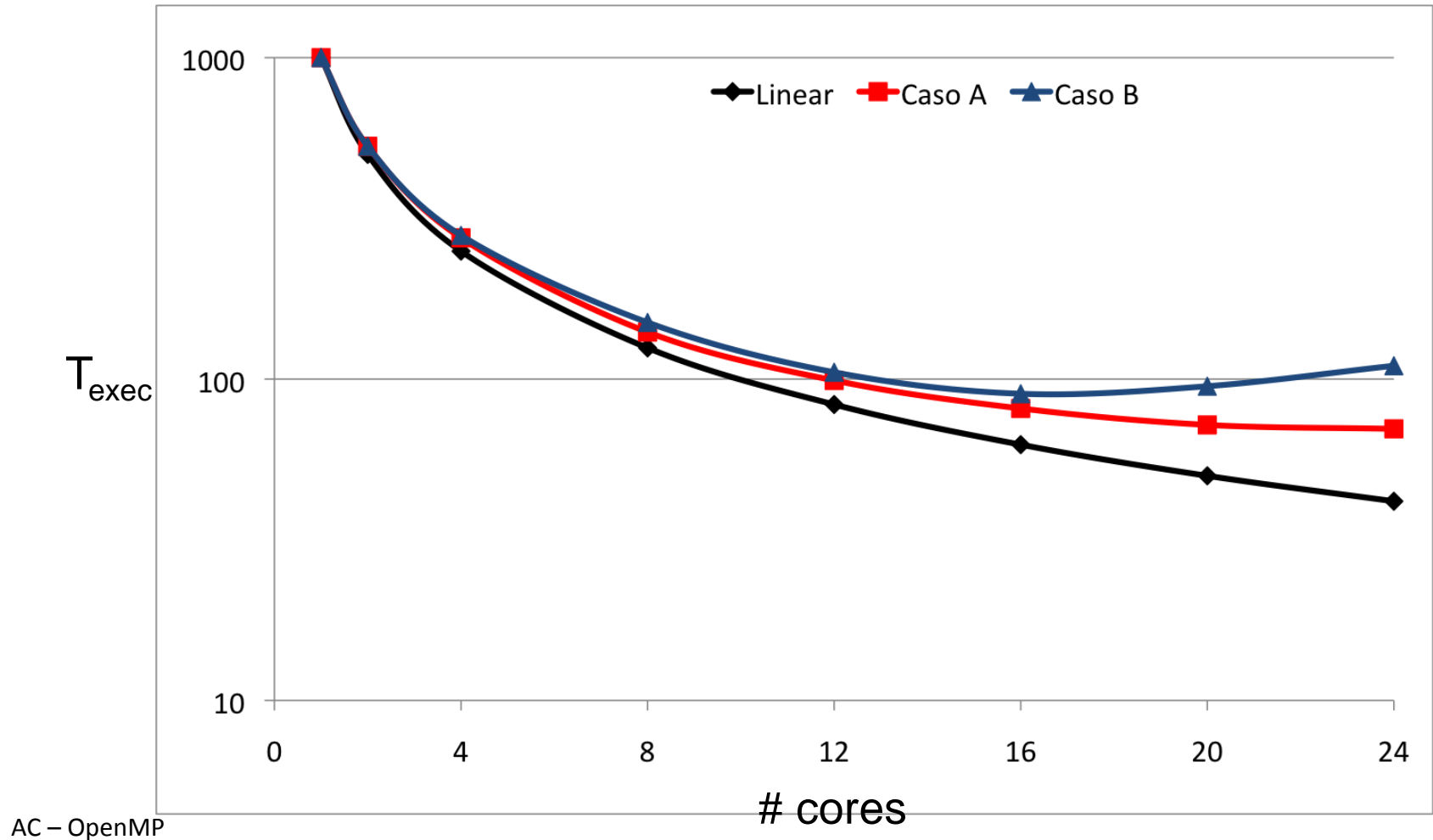
# desempenho – tempo de execução

- OBJECTIVO: diminuir o tempo de execução



# desempenho – tempo de execução

- Escala Logarítmica





## desempenho – *speed up*

$$S_p = \frac{T_1}{T_p}$$

$p$  – número de processadores

$T_1$  – tempo de execução  $p=1$

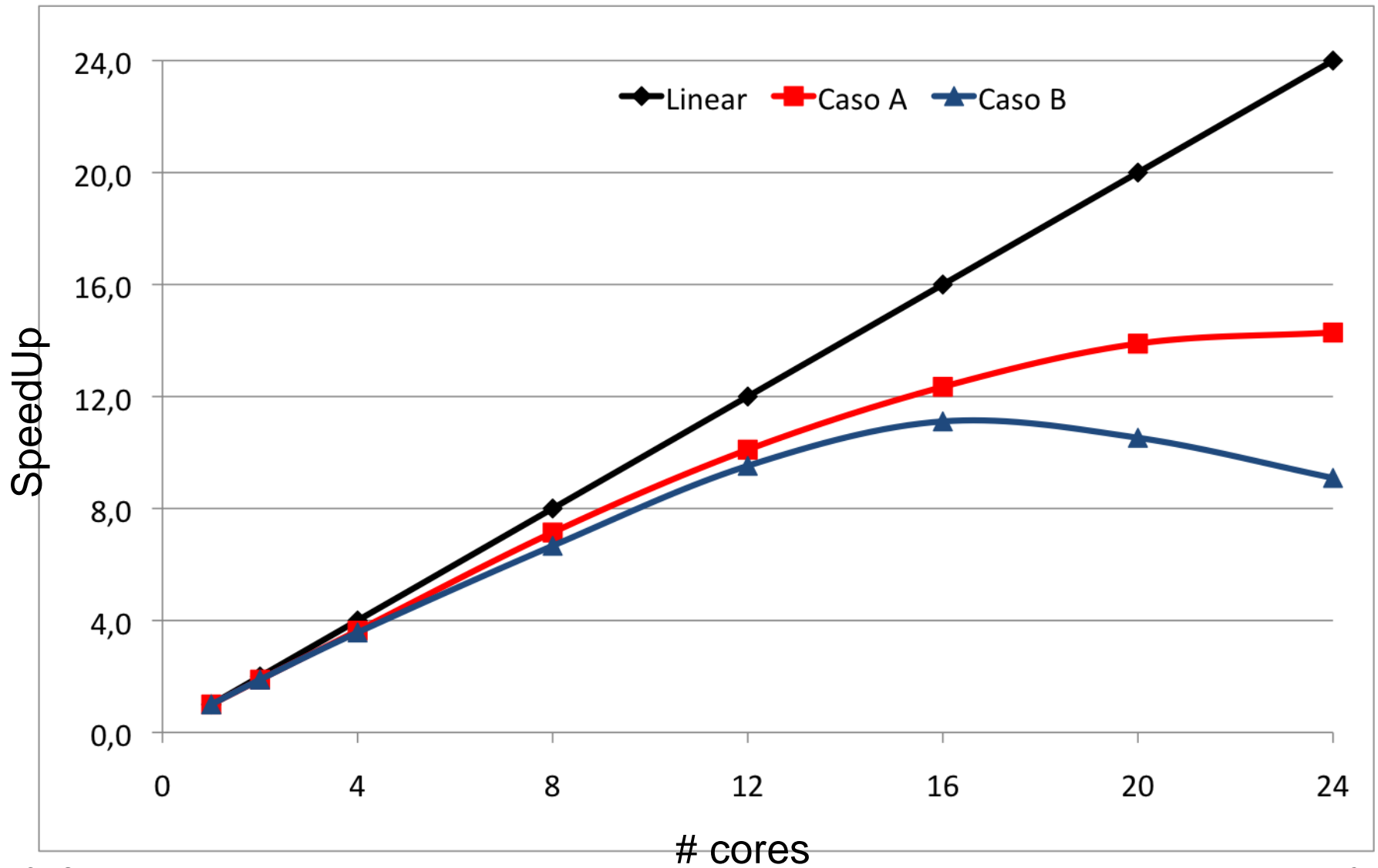
$T_p$  – tempo de execução

com  $p$  processadores

- indica quantas vezes mais rápida é a versão paralela com  $p$  processadores relativamente à versão sequencial
- O desafio está na escolha de  $T_1$ :
  - deve-se usar o mesmo algoritmo mas apenas 1 processador?
  - deve-se usar o melhor algoritmo sequencial conhecido para aquele problema?

A resposta depende claramente do que se pretende avaliar com este ganho!

# desempenho – *speed up*



## desempenho – eficiência

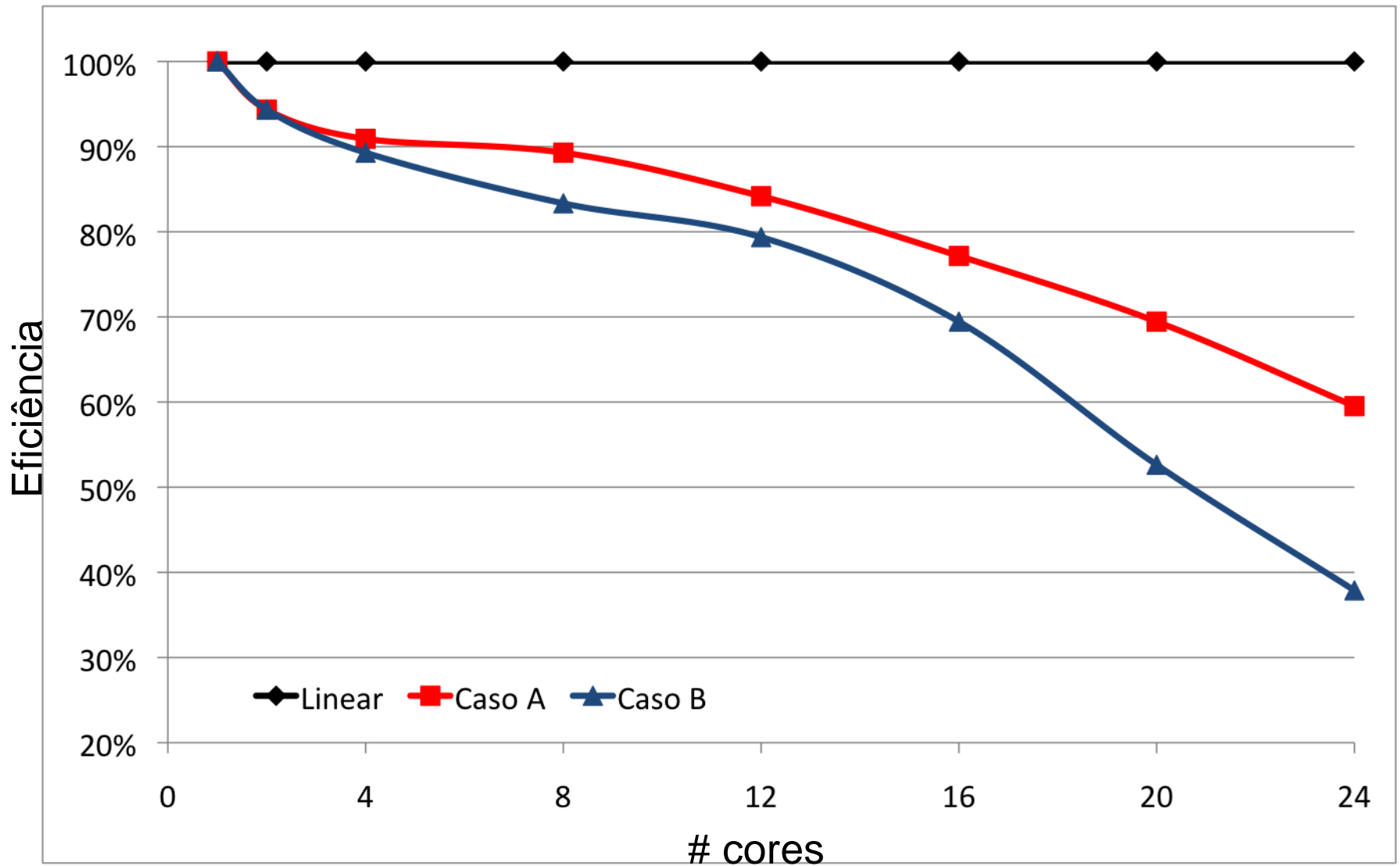
$$E_p = \frac{S_p}{p}$$

$p$  – número de processadores

$S_p$  – *speed up* com  $p$  processadores

- indica em que medida estão os  $p$  processadores a ser bem utilizados
- Razão entre o *speed up* observado e o ideal (=p)
- A utilização total efectiva dos processadores resultaria numa eficiência de 100%

# desempenho – eficiência



# desempenho

- O *speed up* observado é inferior ao linear (ou a eficiência é inferior a 100%) devido a vários custos (*overheads*) associados ao paralelismo:
  - gestão do paralelismo
  - replicação de trabalho
  - distribuição da carga
  - comunicação / sincronização