

# *OpenMP*

Arquitetura de Computadores  
Mestrado Integrado em  
Engenharia Informática

# Material de Apoio

- Tutoriais, exemplos e material diverso no *site* oficial:  
<http://www.openmp.org/>
- Secções “1.3 – Execution Model” e “1.4 – Memory Model”, do “Open MP – Application Programming Interface”, v. 4.0  
(<http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>)

# O que é o OpenMP

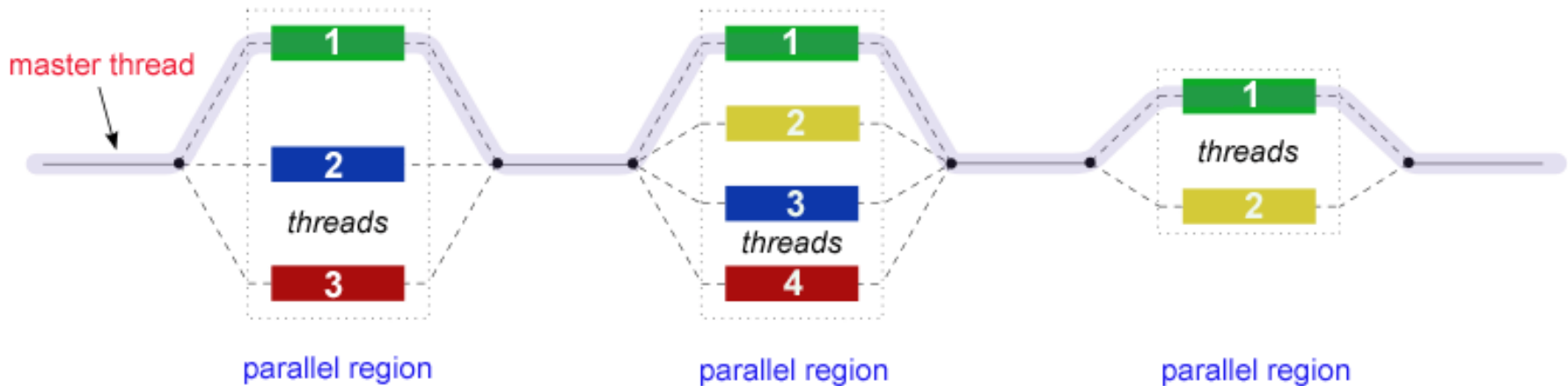
## *Open Multi Processing*

- **API** para expressar **paralelismo *multi-threaded*** e de **memória partilhada**
- standard mantido pelo *OpenMP Architecture Review Board*
- Nov.2015 : versão 4.5 (maior parte dos compiladores suporta 4.0)
- 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]

# 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>
```

# 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

```
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);`
  3. variável de ambiente `OMP_NUM_THREADS`  
`> export OMP_NUM_THREADS=8`
  4. Por omissão: dependente da implementação  
normalmente igual ao número de processadores disponível  
para o programa

# 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` – ordem de execução

```
#include <omp.h>

#pragma omp parallel num_threads(2)
{
    printf("Há %d threads\n",omp_get_num_threads ());
    printf("Esta é a thread %d\n",omp_get_thread_num());
}

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

```
>./prog
Há 2 threads!
Esta é a thread 0!
program done
Há 2 threads!
Esta é a thread 1!
program done
>_
```

```
>./prog
Há 2 threads!
Há 2 threads!
Esta é a thread 1!
Esta é a thread 0!
program done
>_
```

```
>./prog
Há 2 threads!
Esta é a thread 0!
program done
Há 2 threads!
Esta é a thread 1!
>_
```

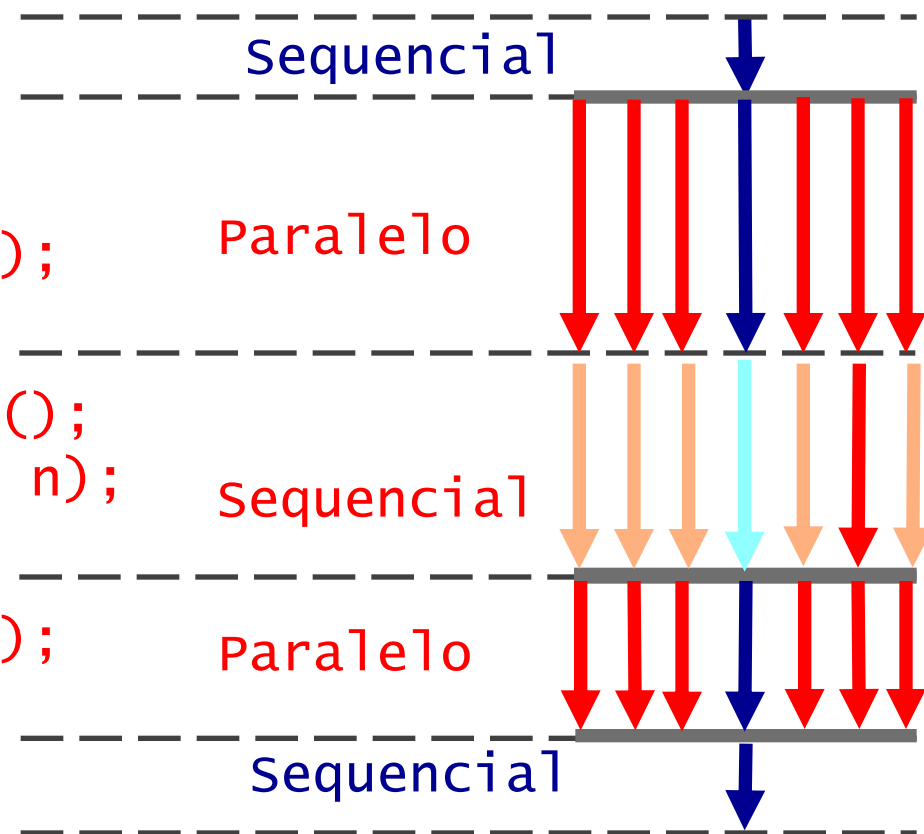
Selecione o *output* possível!

# 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");
```



# parallel – ordem de execução

```
#include <omp.h>
#pragma omp parallel num_threads(2) {
    printf("Esta é a thread %d\n",omp_get_thread_num());
    #pragma omp single
        printf("Há %d threads"\n",omp_get_num_threads ());
}
printf("program done\n");
```

```
> ./prog
Há 2 threads!
Esta é a thread 1!
Esta é a thread 0!
program done
> _
```

```
> ./prog
Esta é a thread 1!
Há 2 threads!
program done
Esta é a thread 0!
> _
```

```
> ./prog
Esta é a thread 0!
Há 2 threads!
Esta é a thread 1!
program done
> _
```

Selecione o *output* possível!

## Loop construct : directiva `for`

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

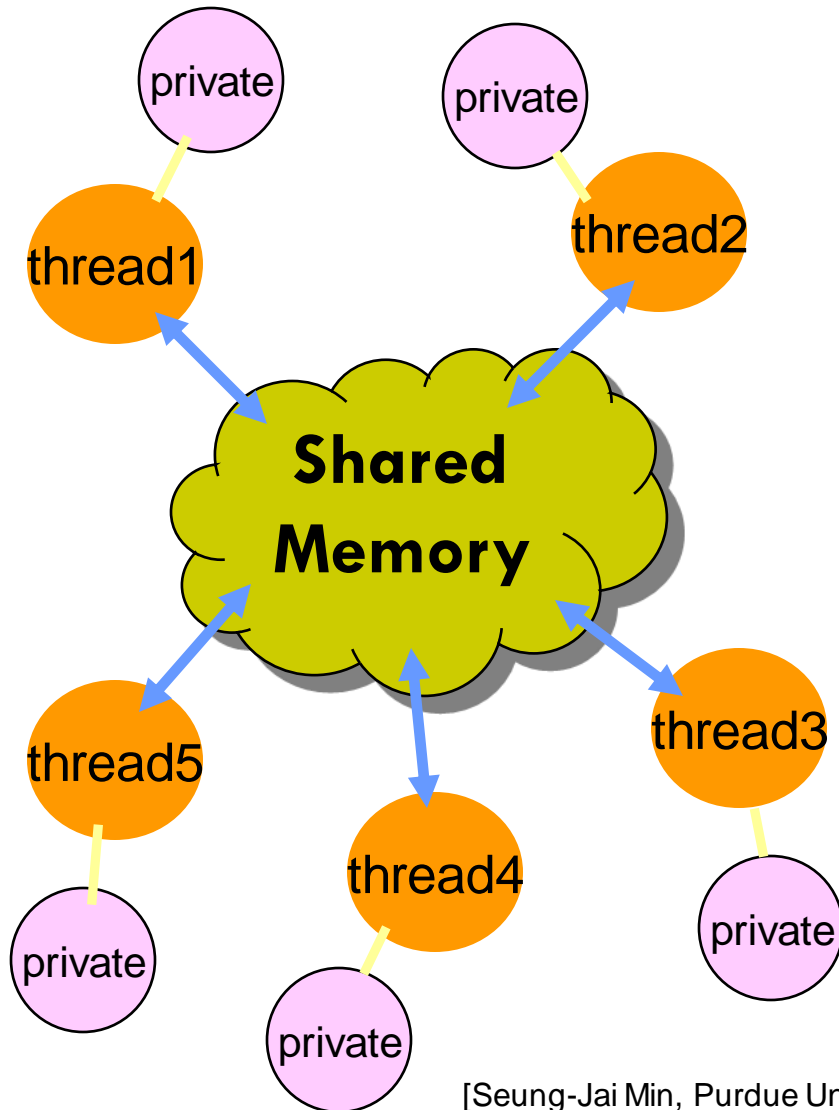
```
#pragma omp for
```

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

- deve estar dentro de um bloco `parallel`
- distribui as iterações do ciclo pelas *threads* activas no grupo (*team*) actual:
  - o espaço de iterações (`i=0 .. N-1`, neste exemplo) é decomposto em sub-intervalos (*chunks*) consecutivos;
  - os *chunks* são distribuídos pelas *threads*
  - sem informação adicional nada se pode assumir sobre o número ou tamanho dos *chunks*, nem sobre a sua distribuição pelas *threads*, excepto de que cada *thread* será responsável pela execução de pelo menos 1 *chunk*
- `for` e `parallel` podem ser combinadas

```
#pragma omp parallel for num_threads(2)
for (i=0; i<N; i++) A[i] = B[i] + C[i];
```

# Modelo de dados



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

[Seung-Jai Min, Purdue University]

# Modelo de dados

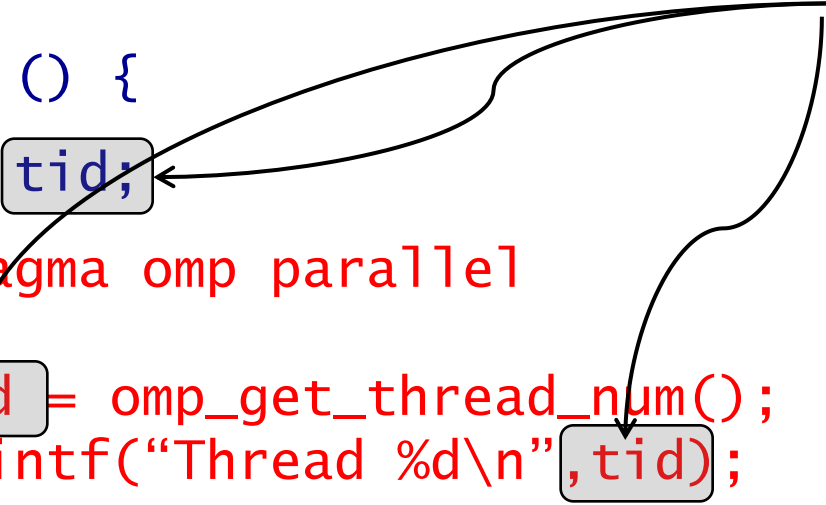
- Por omissão os dados são partilhados, i.e.,  
as variáveis globais a um bloco paralelo são partilhadas
- Variáveis privadas:
  - declaradas dentro de um bloco paralelo
  - explicitamente marcadas como privadas
  - índices dos ciclos associados a uma directiva `for`



# directiva `parallel` – *data scope*

Variáveis globais a um bloco paralelo são partilhadas por omissão

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



**variável partilhada:**  
as várias *threads*  
escrevem e lêem em  
qualquer ordem, sem  
controlo de acesso

**Resultado indeterminado**

# directiva `parallel` – *data scope*

Variáveis globais a um bloco paralelo são partilhadas por omissão

```
main () {  
    int tid;  
    #pragma omp parallel num_threads(2)  
    {  
        tid = omp_get_thread_num();  
        printf("T %d\n", tid);  
    }  
    printf("program done\n");  
}
```

## Exemplo:

- . *T0*: tid = 0
- . *T1*: tid = 1
- . *T0*: escreve "T1"
- . *T1*: escreve "T1"
- . *T0*: "program done"

**NOTA:** outras ordens de execução são possíveis

## directiva `parallel` – *data scope*

- 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 dos ciclos abrangidos pela directiva `for`

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

# directiva `parallel` – *data scope*

Cláusula **private**: 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*

**Declaração dentro do bloco: 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 `for` – *data scope*

- Apenas são privados os índices dos ciclos associados a uma directiva `for`

```
int r, c, k;  
#pragma omp parallel for private(c,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!!

# directiva for - collapse

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

```
int i,c;  
#pragma omp parallel for collapse(2)  
for (i=0; i<N; i++) {  
    for (c=0 ; c<M ; c++) {  
        M[i,c] = sqrt (M[i,c]); } }  
}
```

i e c são privadas

i	0	0	0	0	1	1	1	1	...	N-1	N-1	N-1	N-1
c	0	1	...	M-1	0	1	...	M-1	...	0	1	...	M-1

*chunks* são intervalos contíguos neste espaço de  $N \cdot M$  iterações, seguindo a ordem sequencial conforme tabelado neste exemplo.

# controle 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.

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

Se uma *thread* está dentro de uma região crítica,  
então nenhuma outra *thread* entra nessa região até a *thread* anterior sair:  
-> a execução das regiões críticas não acontece em paralelo, é **sequencial**

# controlo de acessos a dados partilhados

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

As regiões críticas sem nome são consideradas a mesma região!  
Se uma *thread* está em  $x = x+1$ , então nenhuma outra *thread* entra em  $y = y+1$  e vice-versa

Duas regiões críticas distintas.  
 $x = x+1$  e  $y = y+1$   
podem acontecer em paralelo

```
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

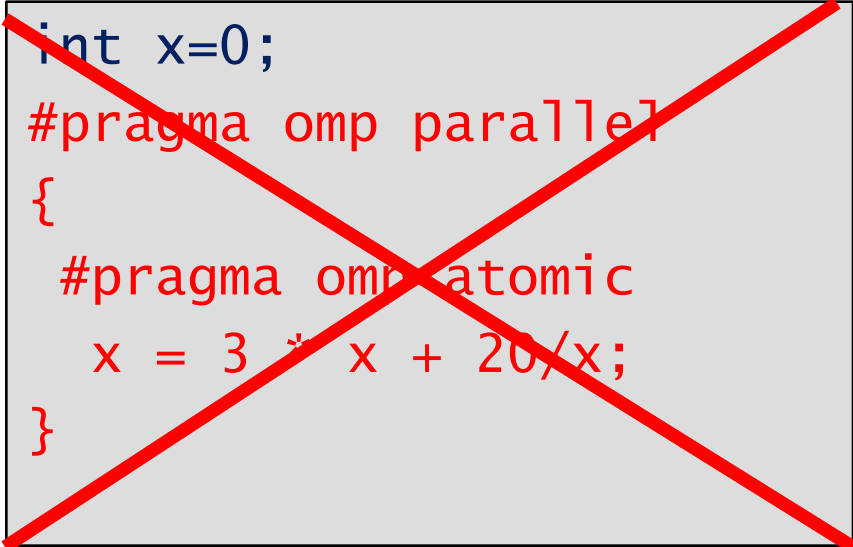
directiva `atomic`: garante que um endereço de memória é acedido de forma atómica. Pode ser vista como uma versão leve de `critical`.

Só se aplica a operações simples (atómicas).

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 maxl = a[0];  
    #pragma omp for  
    for (i=0; i< SIZE ; i++)  
        if (a[i]>maxl) maxl=a[i];  
    #pragma omp critical  
    if (maxl > max) max = maxl;  
}
```

# 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 em vírgula flutuante (float, 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

- Caso 1 - A quantidade de trabalho a realizar é igual para todas as iterações. Exemplo:

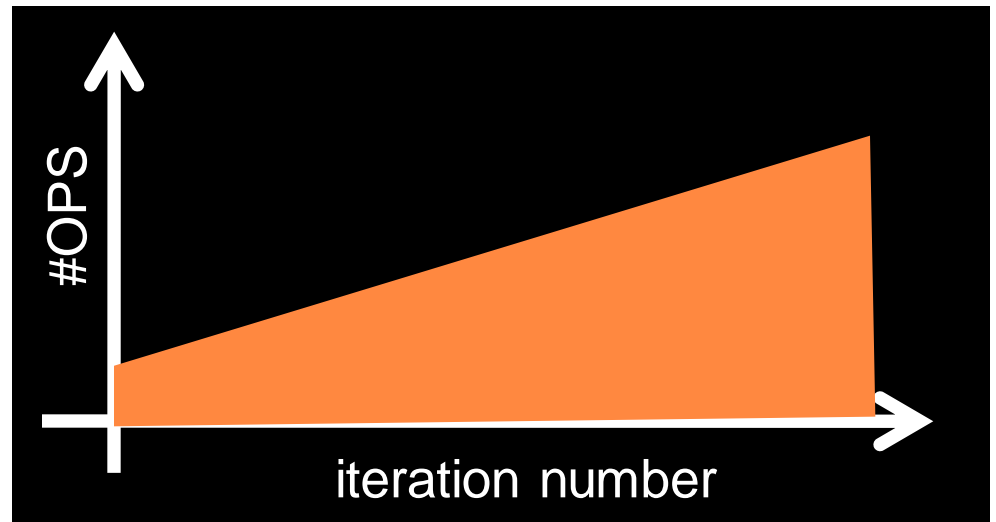
```
#pragma omp parallel for  
{  
  for (i=0; i< SIZE ; i++)  
    b[i] = a[i]*a[i] + 10. / a[i]; }
```



# Escalonamento

- Caso 2 - A quantidade de trabalho a realizar varia entre iterações. Exemplo:

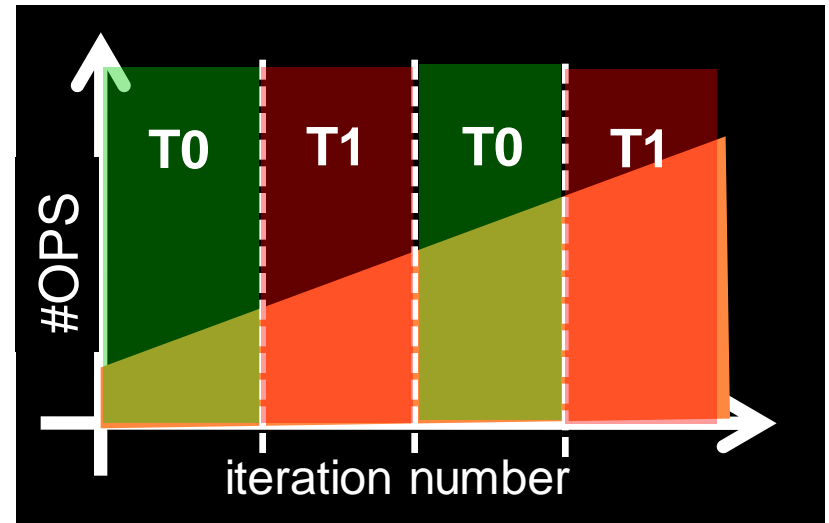
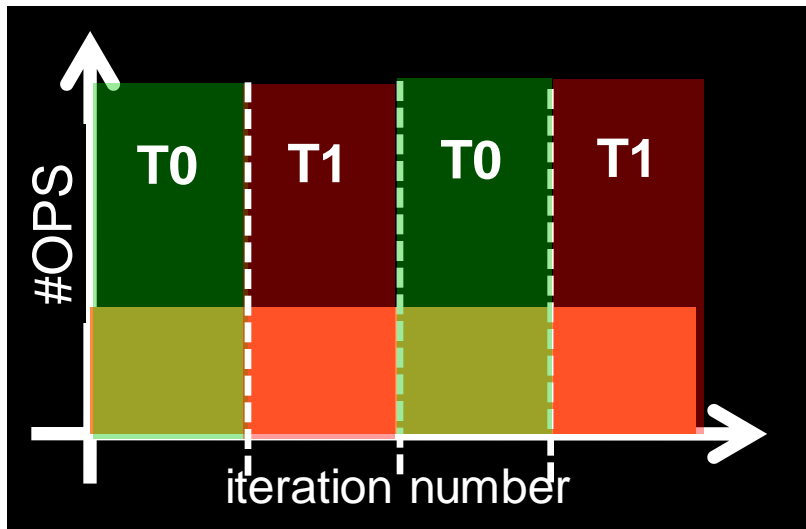
```
#pragma omp parallel for private (j)
{
  for (i=0; i< SIZE ; i++) {
    b[i] = 1.F;
    for (j=2 ; j<= i ; j++)
      b[i] *= j; } }
```



# Escalonamento Estático

**#pragma omp parallel for schedule(static, <chunksize>)**

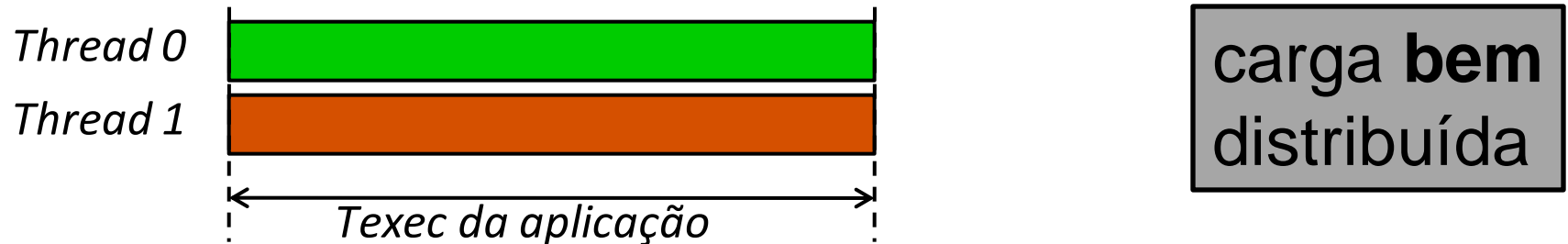
- O espaço de iterações é dividido em *chunks*, com *chunksize* iterações cada
- Os *chunks* são atribuídos às *threads* de forma **estática** usando **round robin**, antes da execução do ciclo se iniciar
- Pode resultar em desbalanceamento de carga se a quantidade de trabalho variar entre *chunks*



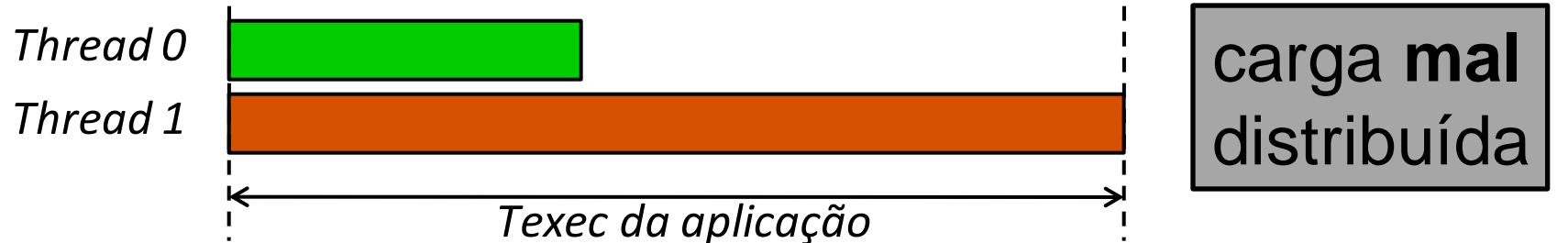
# Escalonamento estático

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

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



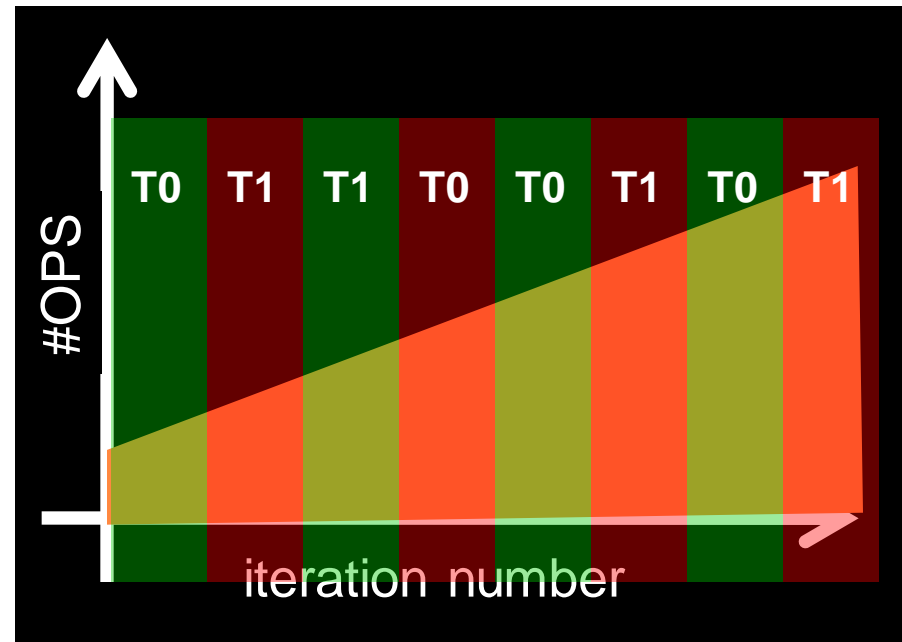
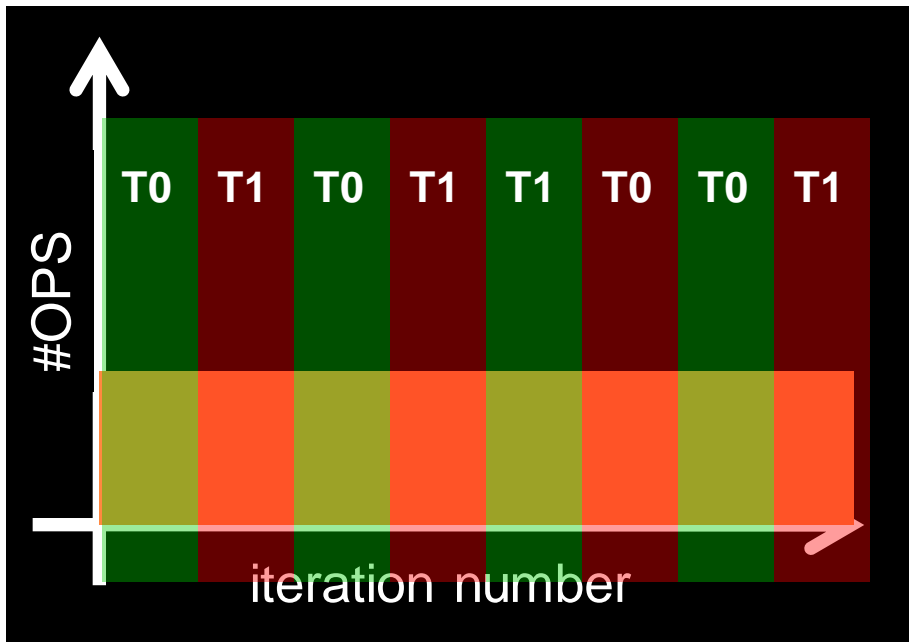
**2 – carga variável para diferentes iterações**



# Escalonamento dinâmico

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

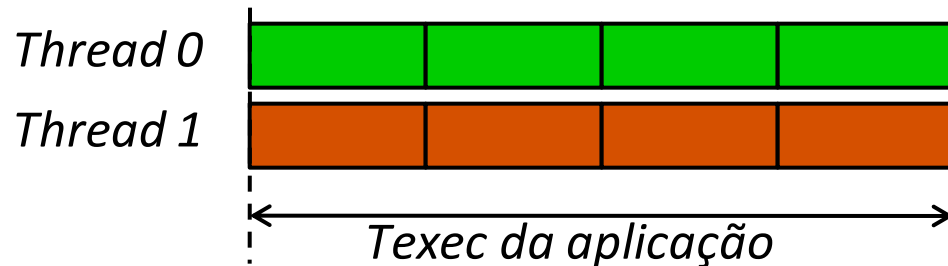
- O ciclo é dividido em muitos segmentos (*chunks*), todos com o mesmo número de iterações, e distribuídos pelas *threads* a pedido



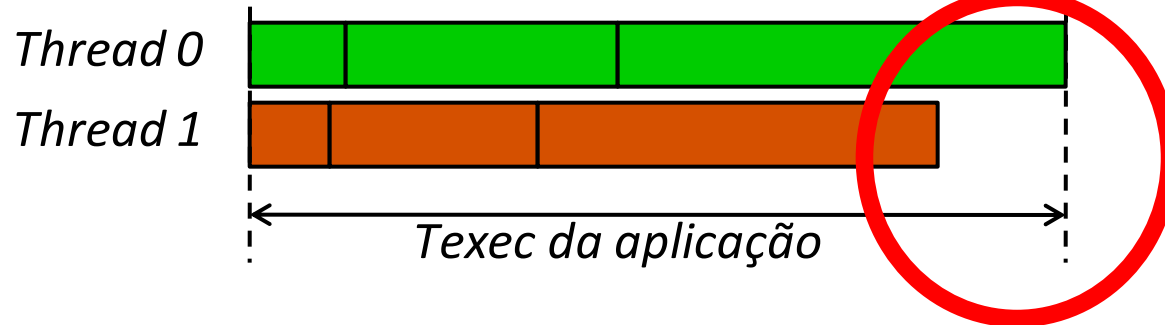
# Escalonamento dinâmico

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

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



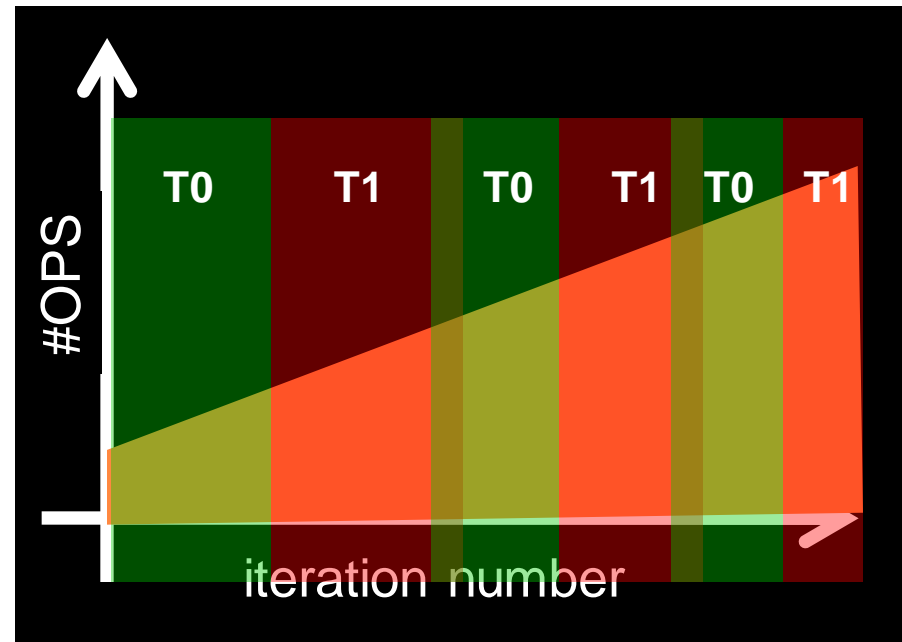
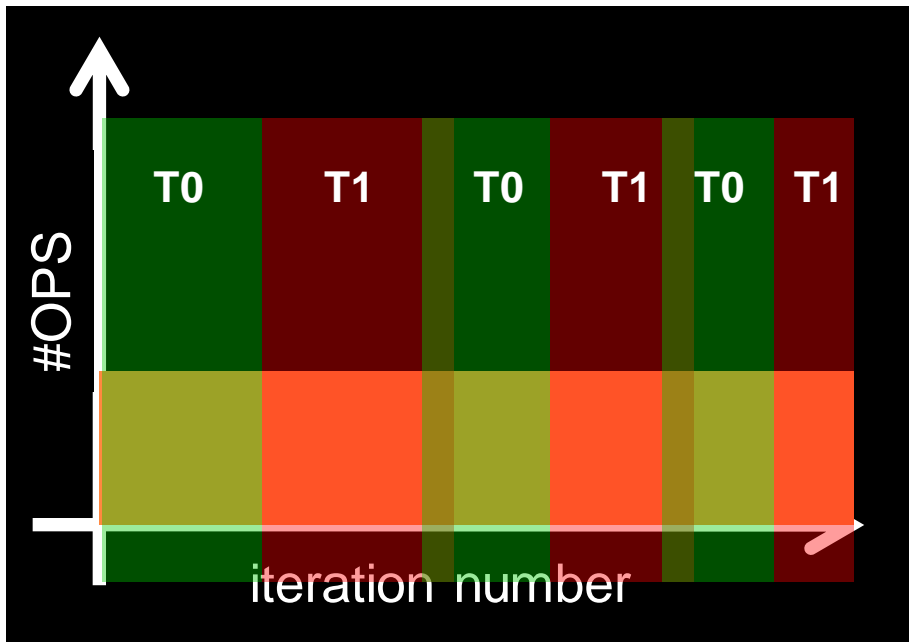
**2 – carga variável para diferentes iterações**



# Escalonamento guiado

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

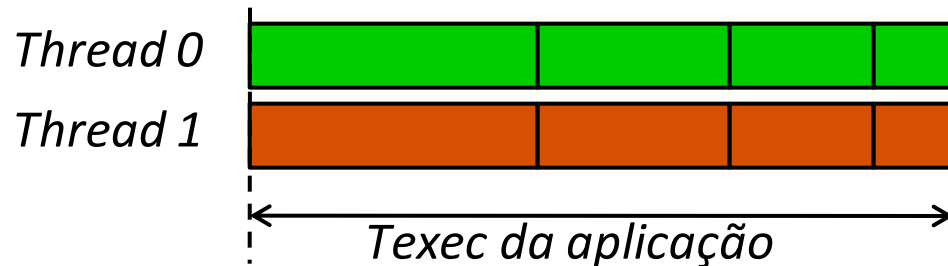
- O ciclo é dividido em muitos segmentos (*chunks*), cada vez com menor número de iterações, e distribuídos pelas *threads* a pedido



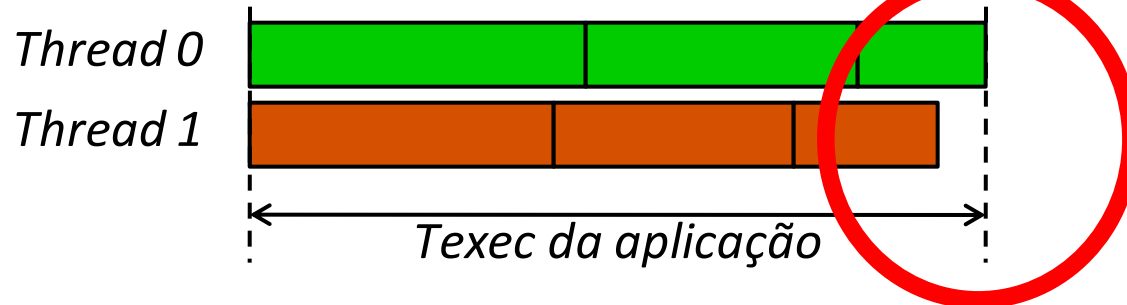
# Escalonamento guiado

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

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



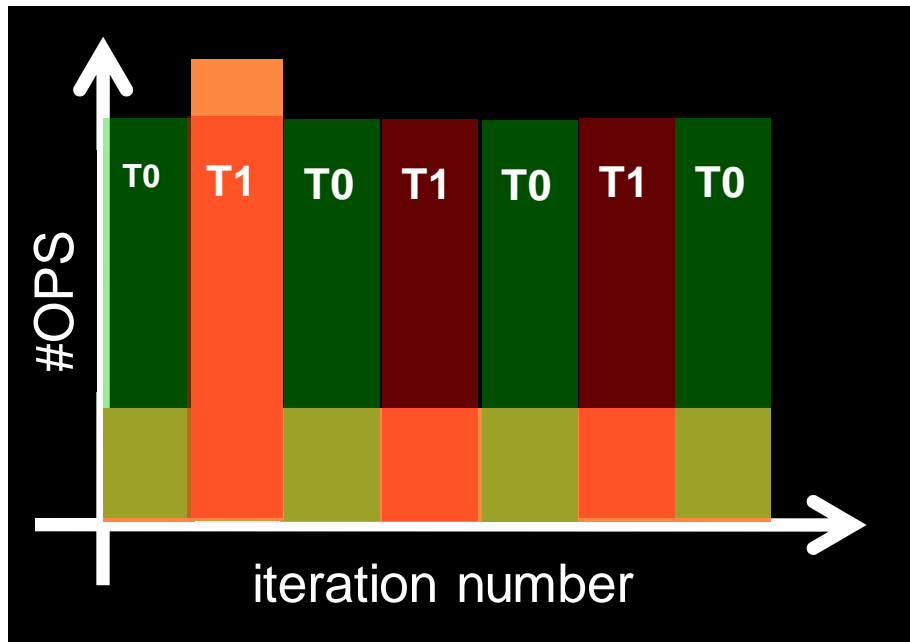
**2 – carga variável para diferentes iterações**



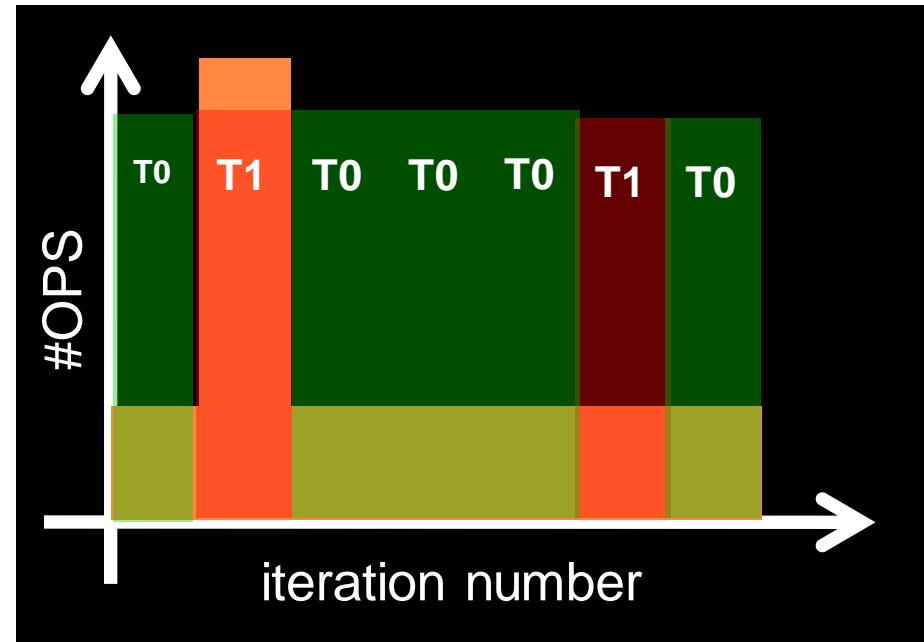


# Escalonamento estático *versus* dinâmico

Exemplo para carga muito irregular



static



dynamic

# desempenho

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

- Com a programação *multithreaded* o número de instruções executadas aumenta, devido à gestão do paralelismo
- Como medir o CPI?

# Desempenho: como medir o CPI

- CPI por processador  $p$

Tende a manter-se constante com a introdução de múltiplos cores

$$CPI_p = \frac{\#cc_p}{\#I_p}$$

- CPI global

Tende a manter-se constante com a introdução de múltiplos cores

$$CPI = \frac{\sum_{p=0}^{P-1} \#cc_p}{\sum_{p=0}^{P-1} \#I_p}$$

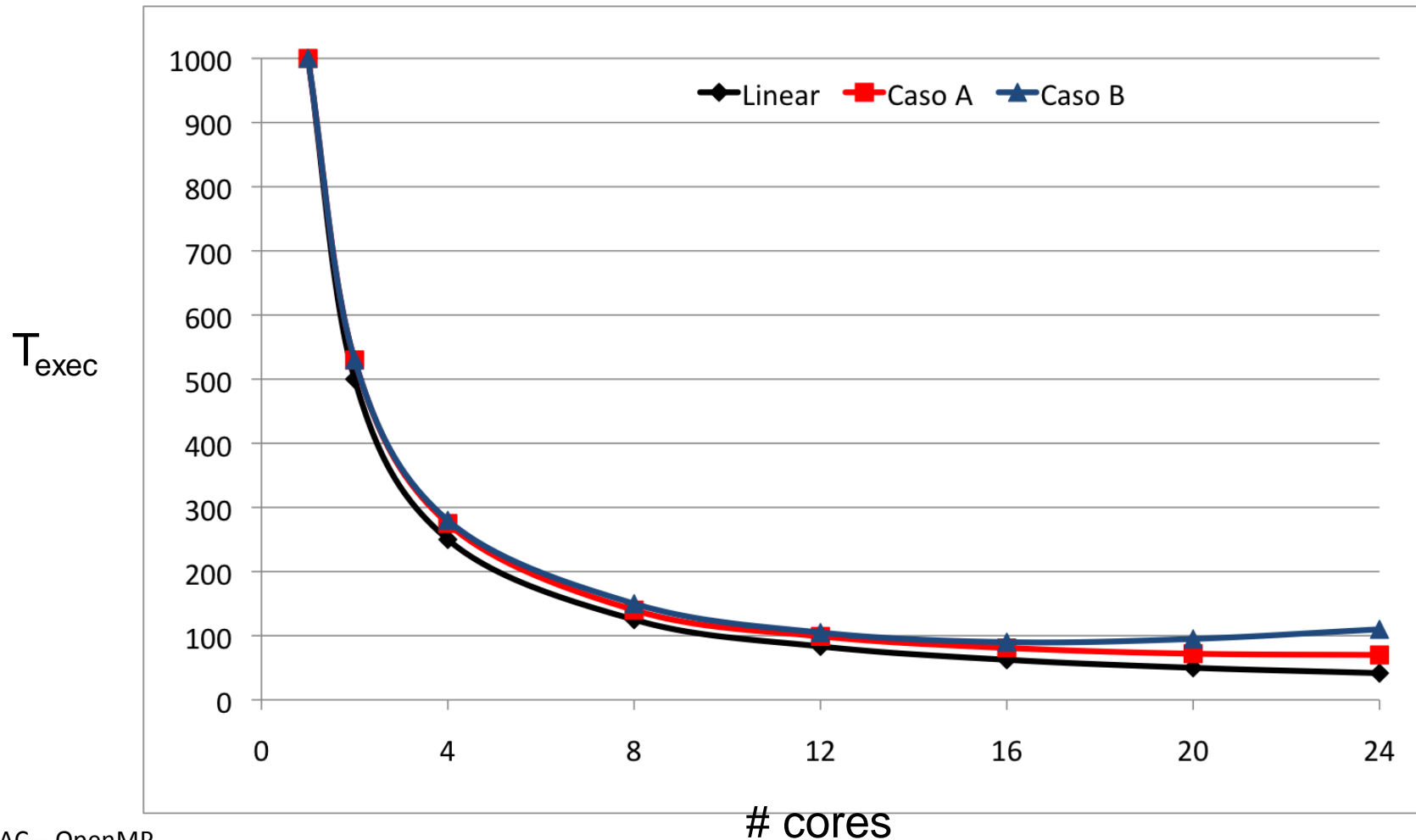
- CPI percebido  
(pelo utilizador)

Tende a diminuir com a adição de múltiplos cores, se o tempo de execução diminuir

$$CPI_{\text{perceived}} = \frac{p \cdot \max(\#cc_p)}{\sum_{p=0}^{P-1} \#I_p}$$

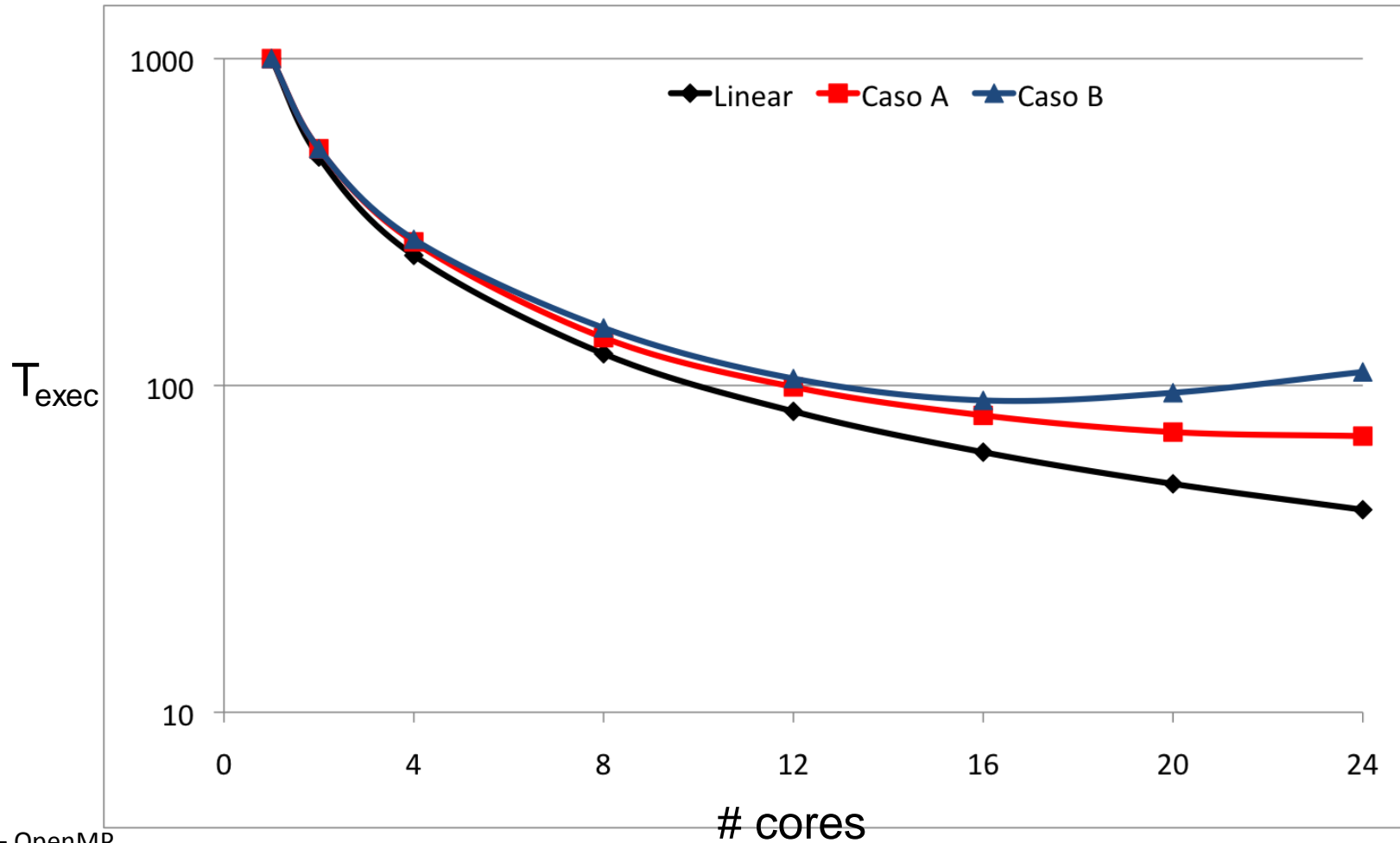
# desempenho – tempo de execução

- 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  $p=1$

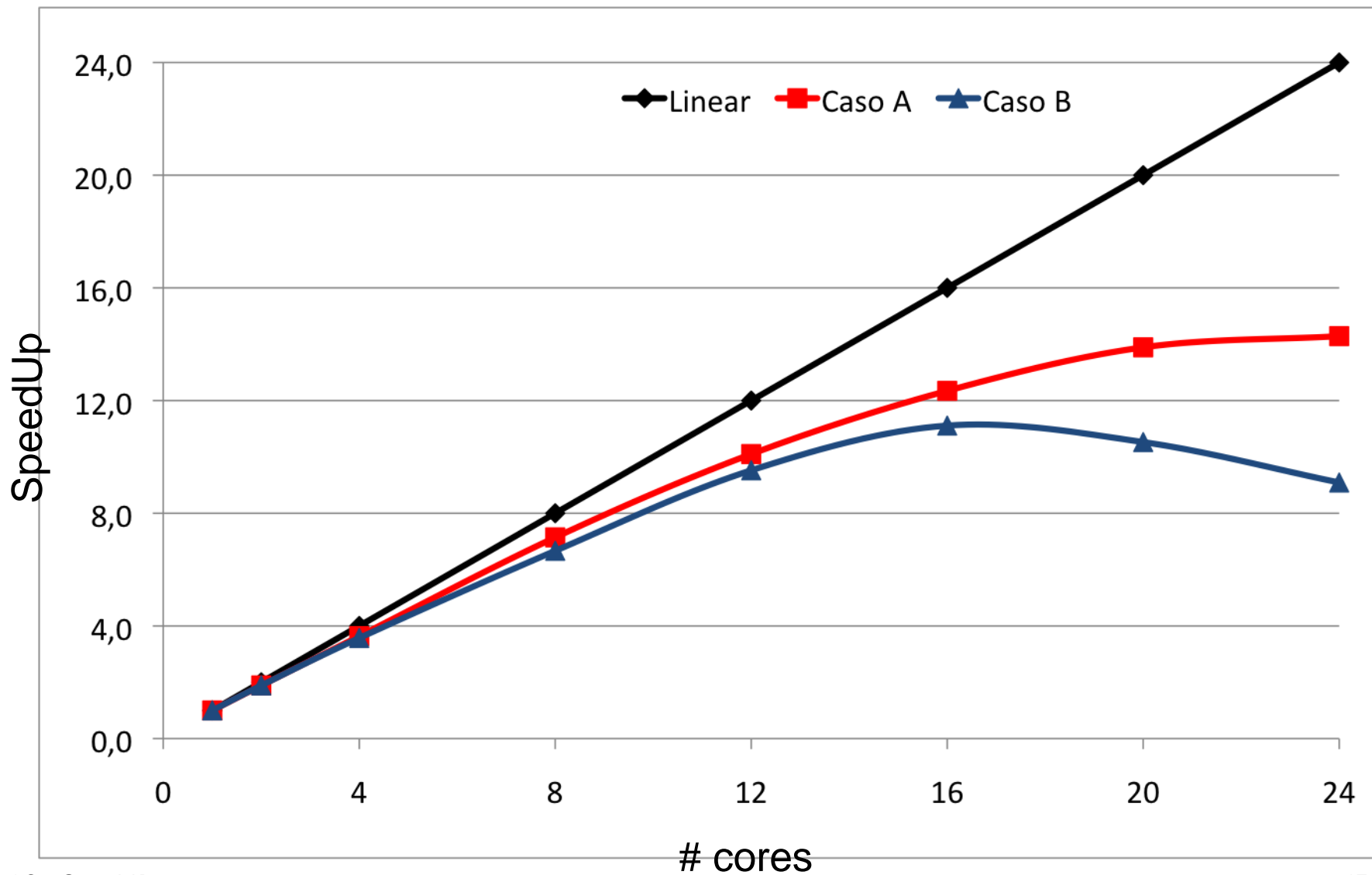
$T_p$  – tempo de execução

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

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

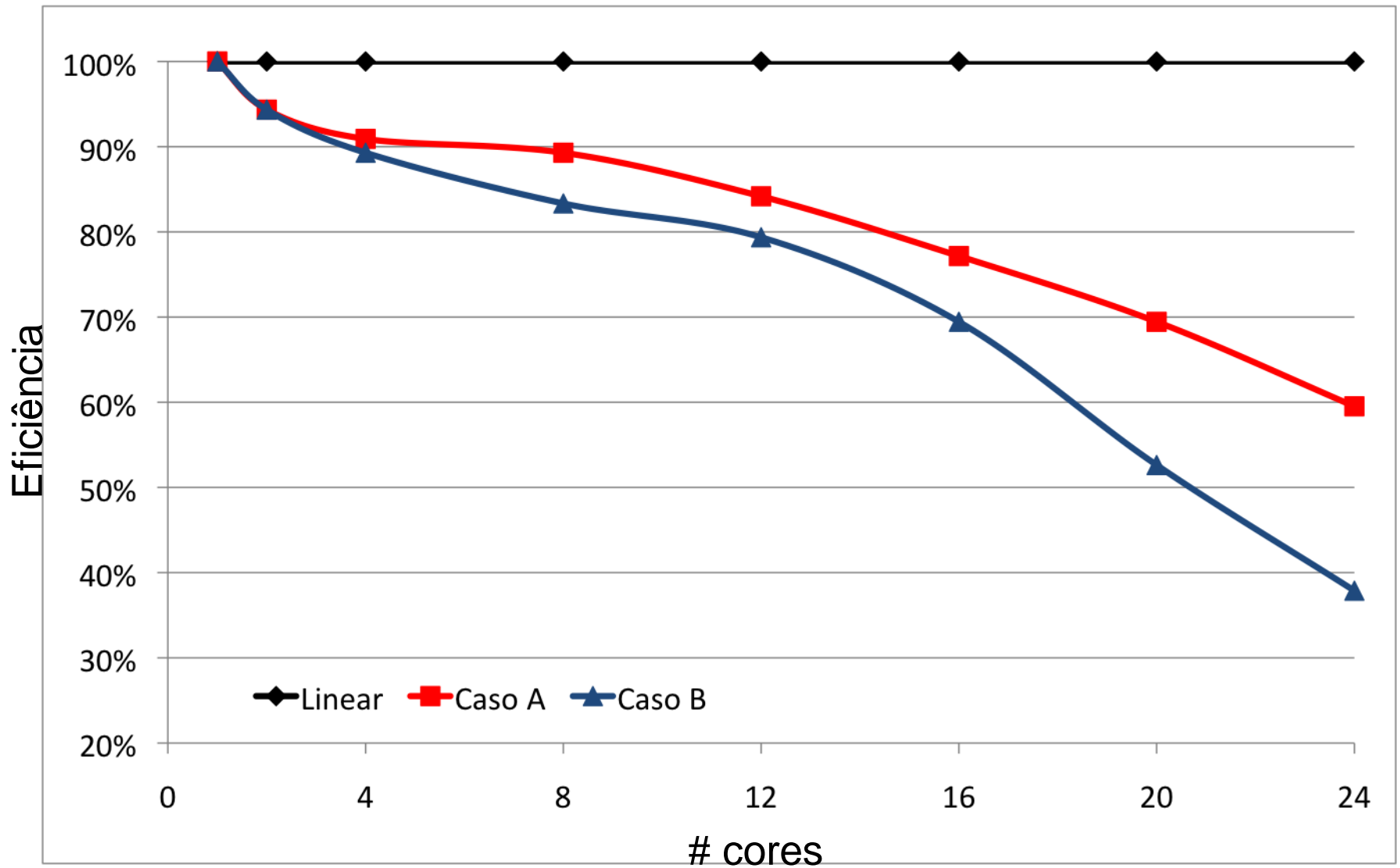
$p$  – número de processadores

$S_p$  – *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