

OpenMP

Arquitetura de Computadores
Licenciatura em Engenharia Informática
Luís Paulo Santos

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

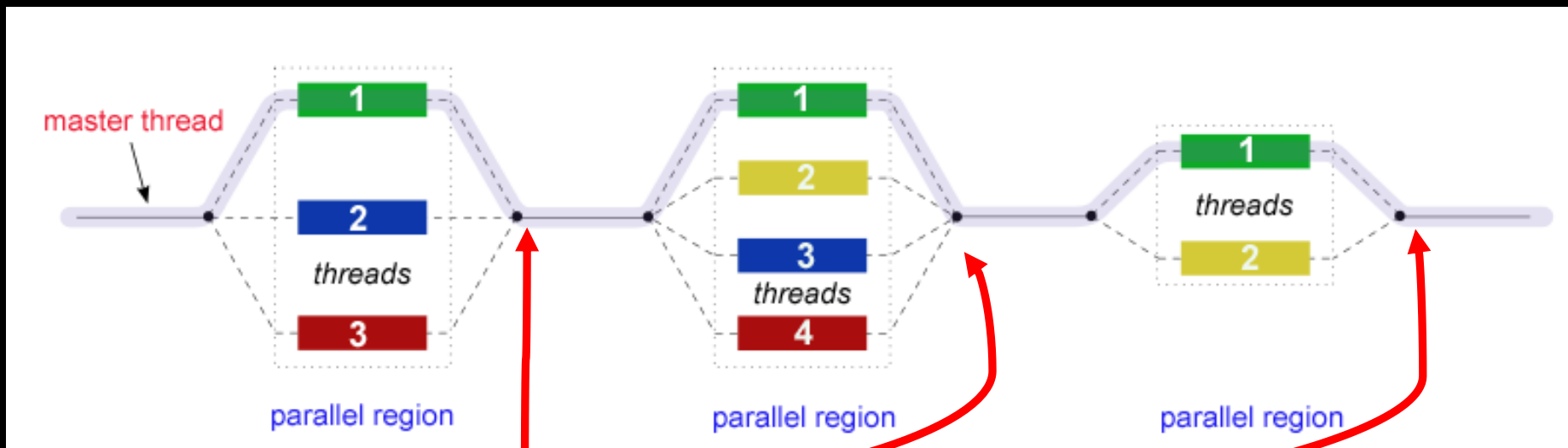
Open Multi Processing

- **API** para expressar **paralelismo *multi-threaded*** e de **memória partilhada**
- standard mantido pelo *OpenMP Architecture Review Board*
- 14.Nov.2024 : versão 6.0
- 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 estão inactivas

Modelo de execução

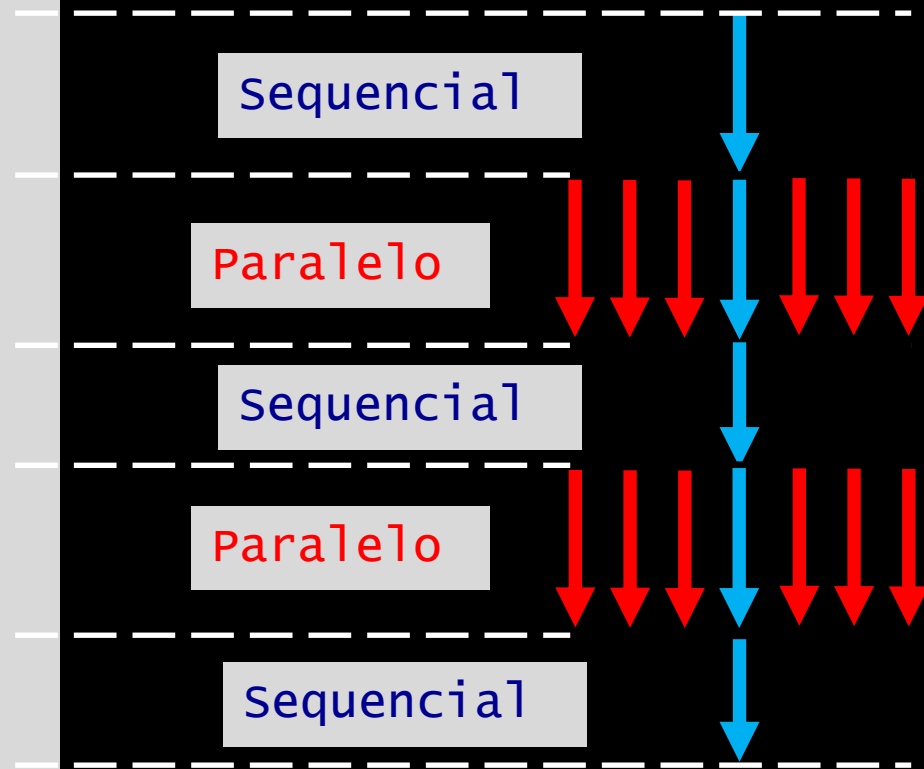
```
printf("program begin\n");  
N = 1000;
```

```
#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");
```



[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!
Há 2 threads!
Esta é a thread 1!
program done
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!
> _
```

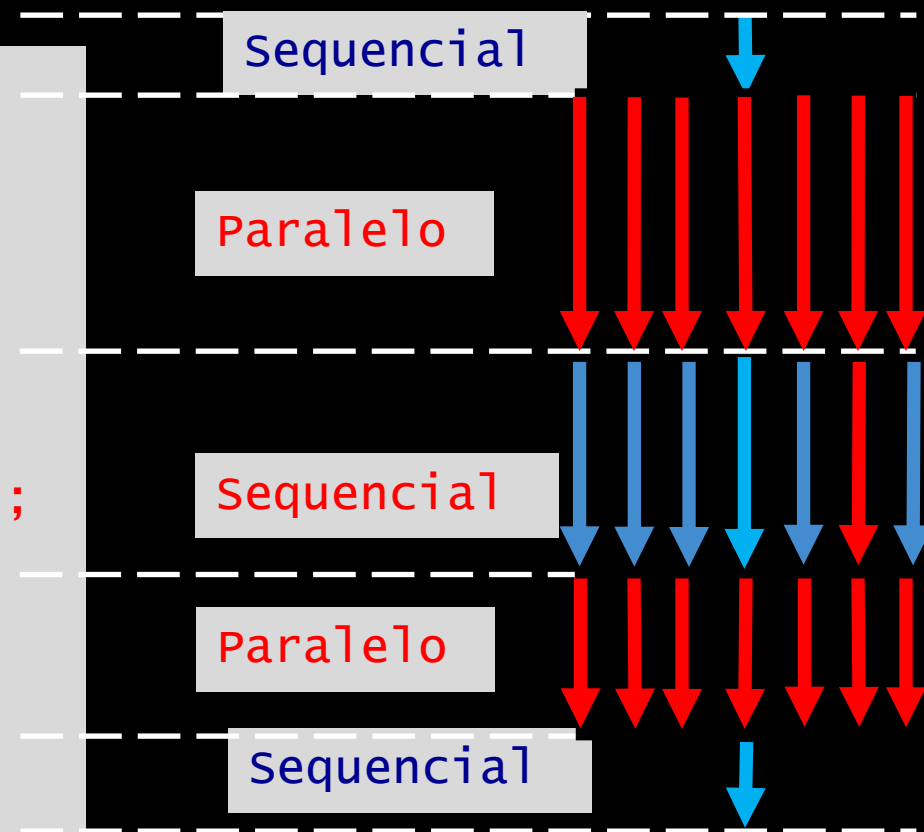
Selecione o *output* possível!

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("Thread %d...\n",omp_get_thread_num());
    #pragma omp single
        printf("Há %d threads!"\n",omp_get_num_threads ());
    printf("...thread %d\n",omp_get_thread_num());
}
printf("program done\n");
```

```
>./prog
Há 2 threads!
Thread 1...
Thread 0...
...thread 0
...thread 1
program done
>_
```

```
>./prog
Thread 1...
Há 2 threads!
...thread 1
Thread 0...
...thread 0
program done
>_
```

```
>./prog
Thread 0...
Há 2 threads!
Thread 1...
...thread 0
...thread 1
program done
>_
```

Selecione o *output* possível!

parallel – ordem de execução

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

nowait elimina a
barreira no fim do
bloco **single**

```
>./prog
Há 2 threads!
Thread 1...
Thread 0...
...thread 0
...thread 1
program done
>_
```

```
>./prog
Thread 1...
Há 2 threads!
...thread 1
Thread 0...
...thread 0
program done
>_
```

```
>./prog
Thread 0...
Há 2 threads!
Thread 1...
...thread 0
...thread 1
program done
>_
```

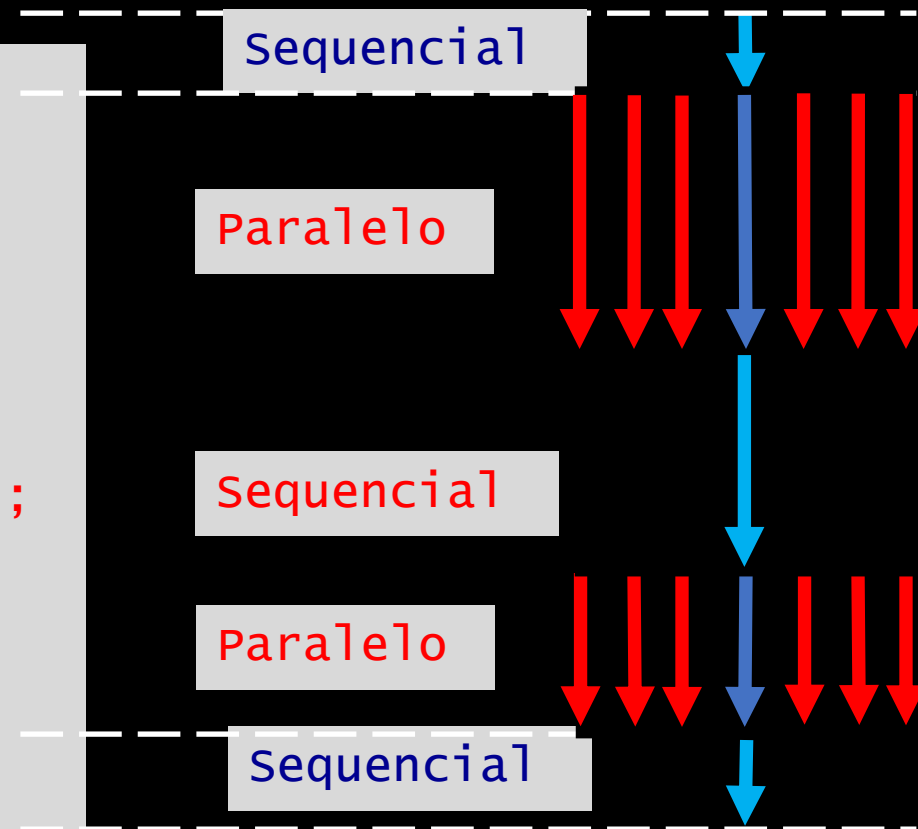
Selecione o *output* possível!

directiva master

Apenas a **master thread** executa o bloco

Não há sincronização implícita no fim do bloco

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

```
>./prog
Há 2 threads!
Thread 1...
Thread 0...
...thread 0
...thread 1
program done
>_
```

```
>./prog
Thread 1...
Há 2 threads!
...thread 1
Thread 0...
...thread 0
program done
>_
```

```
>./prog
Thread 0...
Thread 1...
...thread 1
Há 2 threads!
...thread 0
program done
>_
```

Seleccione 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];
```

Loop construct : directiva `for`

```
#pragma omp parallel for num_threads(4)
  for (i=0; i<100000; i++)
    A[i] = B[i] + C[i];
```

Exemplo

Thread 0	Thread 1	Thread 2	Thread 3
<pre>for (i=0; i<25000; i++) A[i] = B[i] + C[i];</pre>	<pre>for (i=25000; i<50000; i++) A[i] = B[i] + C[i];</pre>	<pre>for (i=50000; i<75000; i++) A[i] = B[i] + C[i];</pre>	<pre>for (i=75000; i<100000; i++) A[i] = B[i] + C[i];</pre>

Desempenho:

se as diferentes *threads* executam em *cores* diferentes **em paralelo** , então o tempo de execução diminui!

desempenho

- 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

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

desempenho

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

- Com a programação *multithreaded* o número de instruções executadas aumenta, devido à gestão do paralelismo
- O $CPI_{perceived}$ diminui com o número de *cores*
 - Esta diminuição compensa largamente o aumento de $\#I$ levando a diminuições muito significativas de T_{exec}
 - A taxa de diminuição do $CPI_{perceived}$ reduz à medida que o número de cores aumenta

desempenho

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

#I= 4e6 #cc=2e6 f=2e9 Hz

CPI= 0.5 $T = 4e6 * 0.5 / 2e9 = 1e-3 \text{ s} = 1 \text{ ms}$

#I₁= 1.9e6 #cc₁=0.9e6 f=2e9 Hz

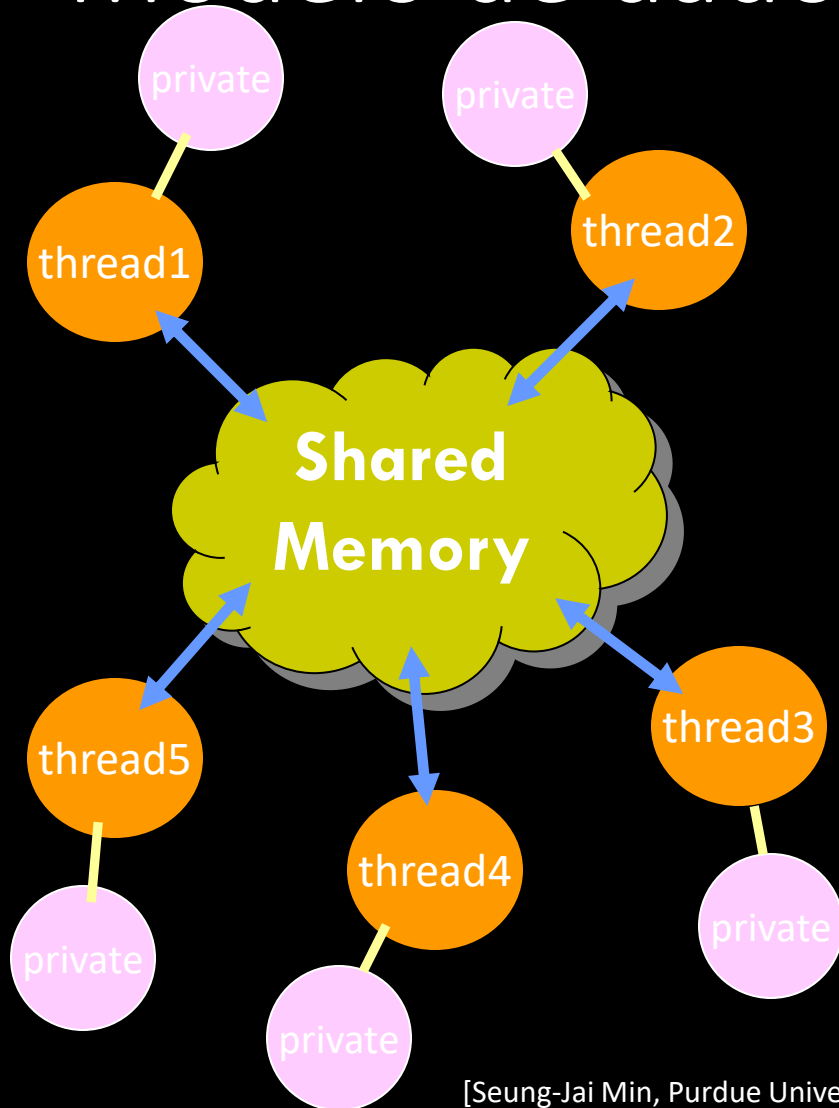
#I₂= 2.1e6 #cc₂=1.2e6

2 cores

$CPI_{perceived} = 1.2e6 / 4e6 = 0.3$

$T = 4e6 * 0.3 / 2e9 = 0.6e-3 \text{ s} = 0.6 \text{ ms}$

Modelo de dados



[Seung-Jai Min, Purdue University]

- 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

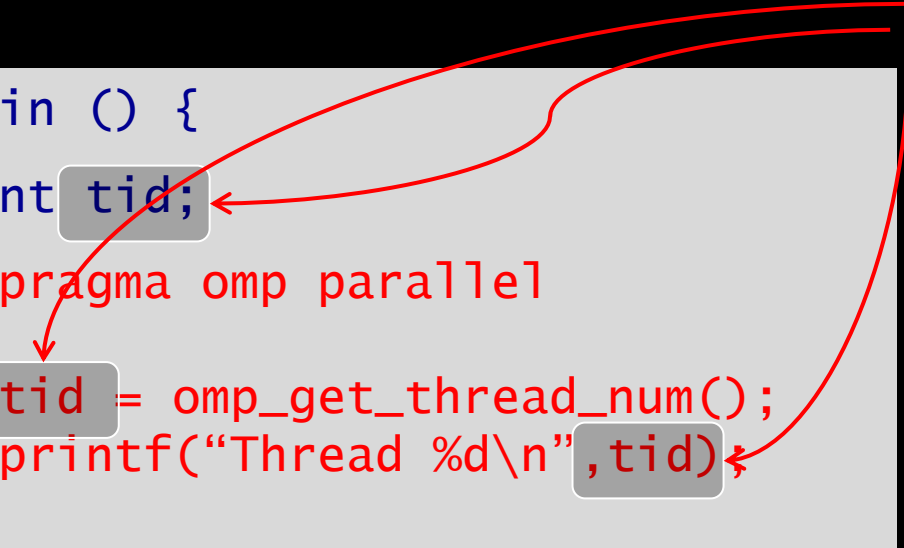
Modelo de dados

- 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!!

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**

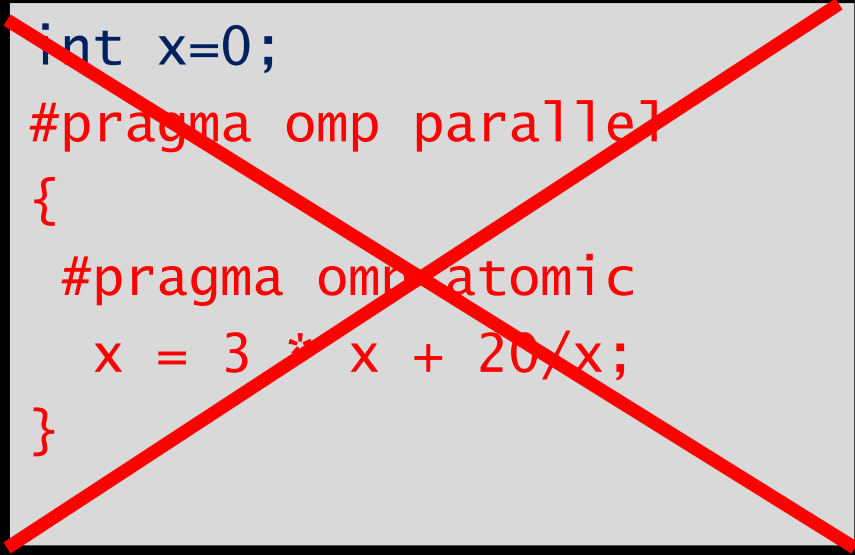
controlo de acessos a dados partilhados

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 atómicas: *update* de uma posição de memória.

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

sum é uma
variável partilhada

redução

muito ineficiente
NA verdade a execução é
sequencial

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

redução

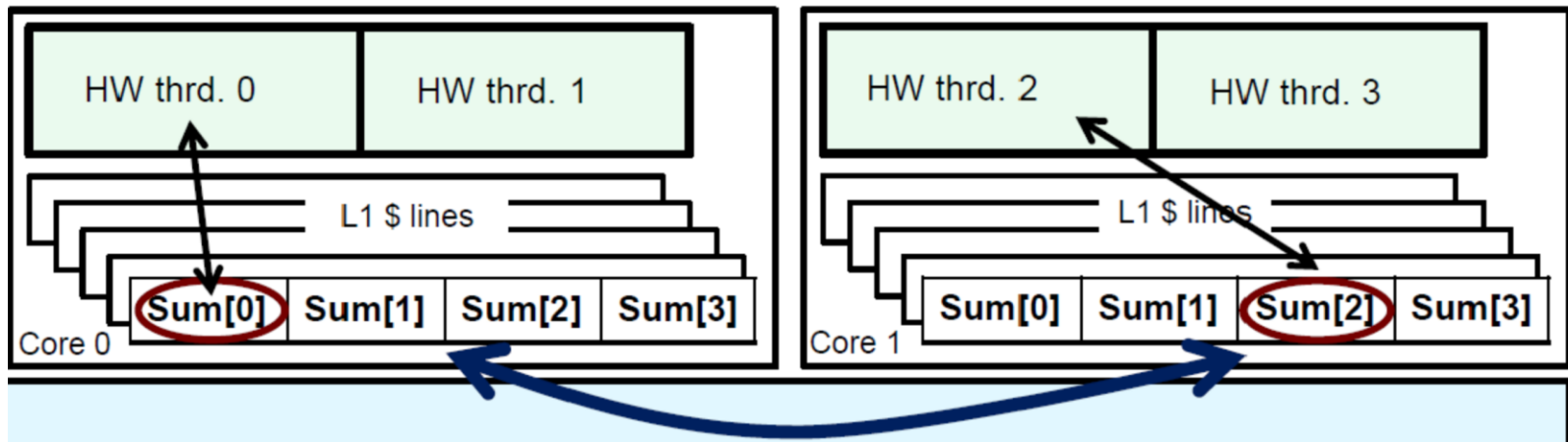
potencial para *trashing*
(*false sharing*)

```
int a[SIZE];
... inicializar a[]
int sum=0, suml[MAXTHREADS];
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    suml[tid] = 0;
    #pragma omp for
        for (i=0; i< SIZE ; i++)
            suml[tid] += a[i];
    #pragma omp atomic
        sum += suml[tid];
}
```

False sharing

False sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads ... This is called “**false sharing**”.



[https://wiki.cdote.senecacollege.ca/wiki/DPS921/Group_8#False_Sharing_in_Parallel_Programming]

redução

variável privada



```
int a[SIZE];  
... inicializar a[]  
int sum=0;  
#pragma omp parallel  
{  
    int sum1 = 0;  
    #pragma omp for  
        for (i=0; i< SIZE ; i++)  
            sum1 += a[i];  
    #pragma omp atomic  
        sum += sum1;  
}
```

redução

- A redução é tão comum que o OpenMP inclui uma cláusula específica

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

- Apenas para operações associativas!

redução

- As operações sobre operandos em vírgula flutuante (float, double) não são associativas:

```
float a[SIZE], sum=0.;  
... inicializar a[]  
#pragma omp parallel for reducti  
{  
    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 Estático

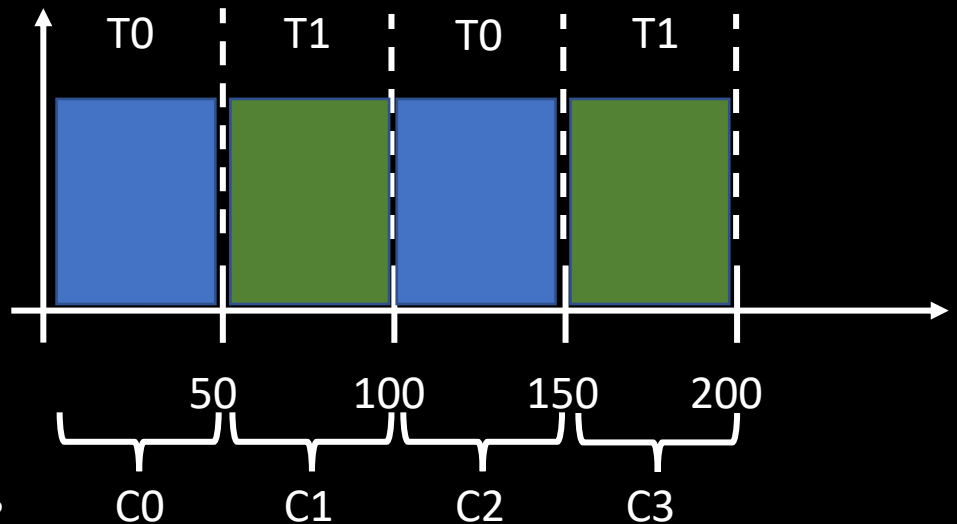
```
#pragma for schedule(static, <chunksize>)
```

- 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 for schedule(static, <chunksize>)
```

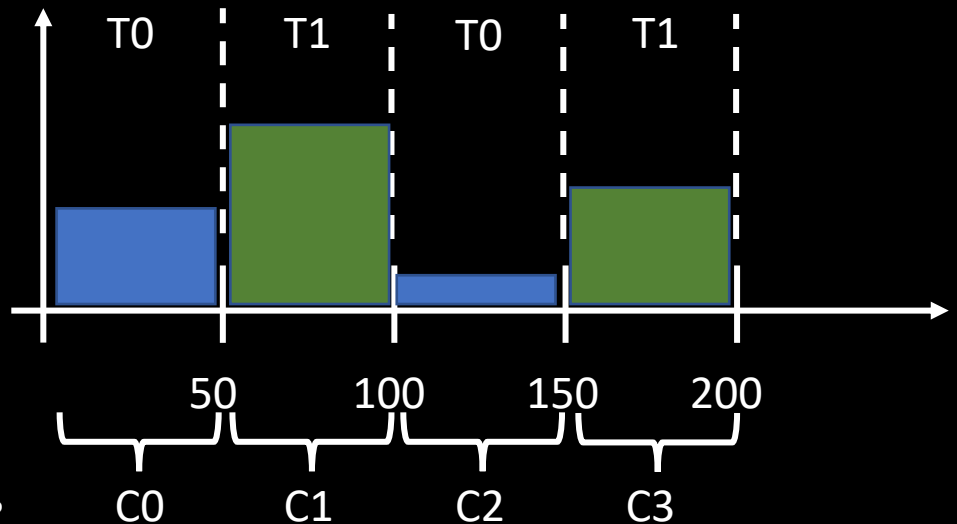
```
#pragma omp parallel for schedule(static, 50) num_threads(2)
{
    for (i=0; i< 200 ; i++)
        a[i] *= a[i];
}
```



Escalonamento Estático

```
#pragma for schedule(static, <chunksize>)
```

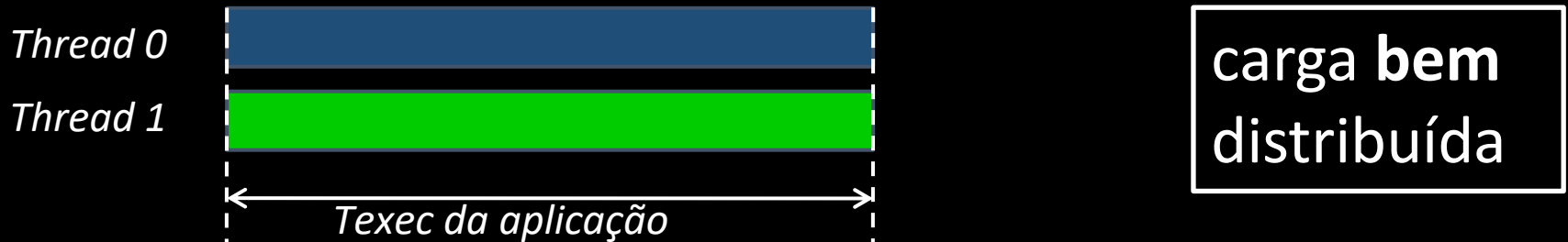
```
#pragma omp parallel for schedule(static, 50) num_threads(2)
{
    for (i=0; i< 200 ; i++)
        a[i] *= func (a[i]);
}
```



Escalonamento estático

#pragma omp parallel for schedule(static)

1 – carga uniforme para todas as iterações



2 – carga variável para diferentes iterações



Escalonamento Dinâmico

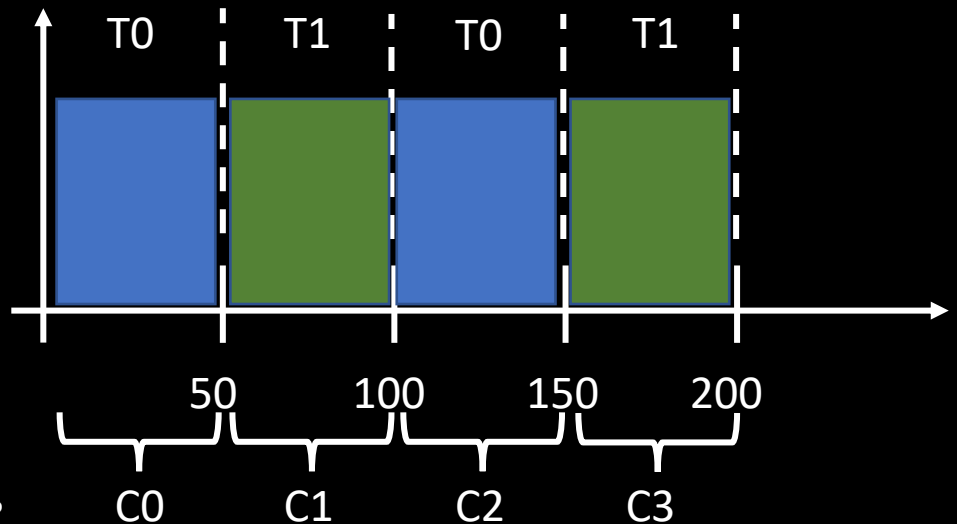
```
#pragma for schedule(dynamic, <chunksize>)
```

- O espaço de iterações é dividido em *chunks*, com *chunksize* iterações cada
- Os *chunks* são atribuídos às *threads* de forma **dinâmica**, a pedido, **durante a execução do ciclo**

Escalonamento Dinâmico

```
#pragma for schedule(dynamic, <chunksize>)
```

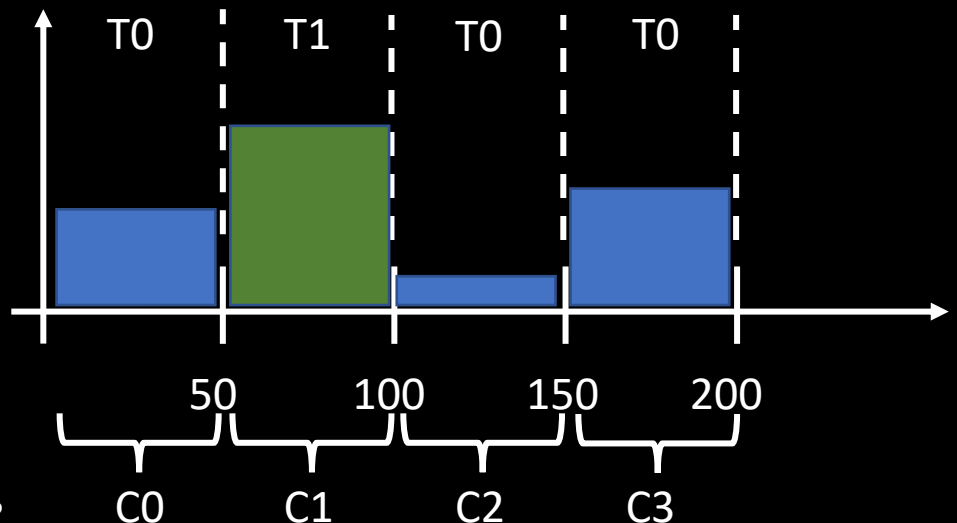
```
#pragma omp parallel for schedule(dynamic, 50)  
num_threads(2)  
{  
  for (i=0; i< 200 ; i++)  
    a[i] *= a[i];}
```



Escalonamento Estático

```
#pragma for schedule(static, <chunksize>)
```

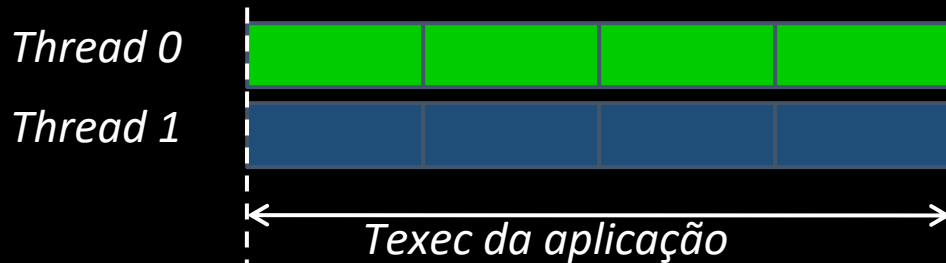
```
#pragma omp parallel for schedule(dynamic, 50)  
num_threads(2)  
{  
  for (i=0; i< 200 ; i++)  
    a[i] *= func (a[i]); }  
}
```



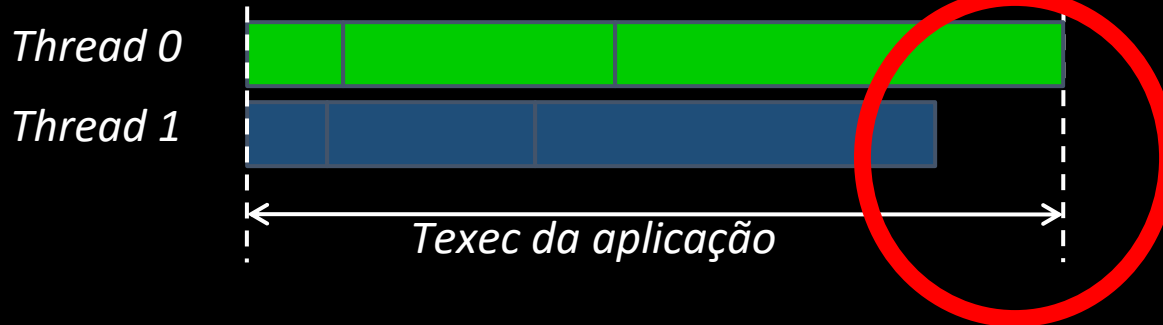
Escalonamento dinâmico

#pragma omp parallel for schedule(dynamic)

1 – carga uniforme para todas as iterações

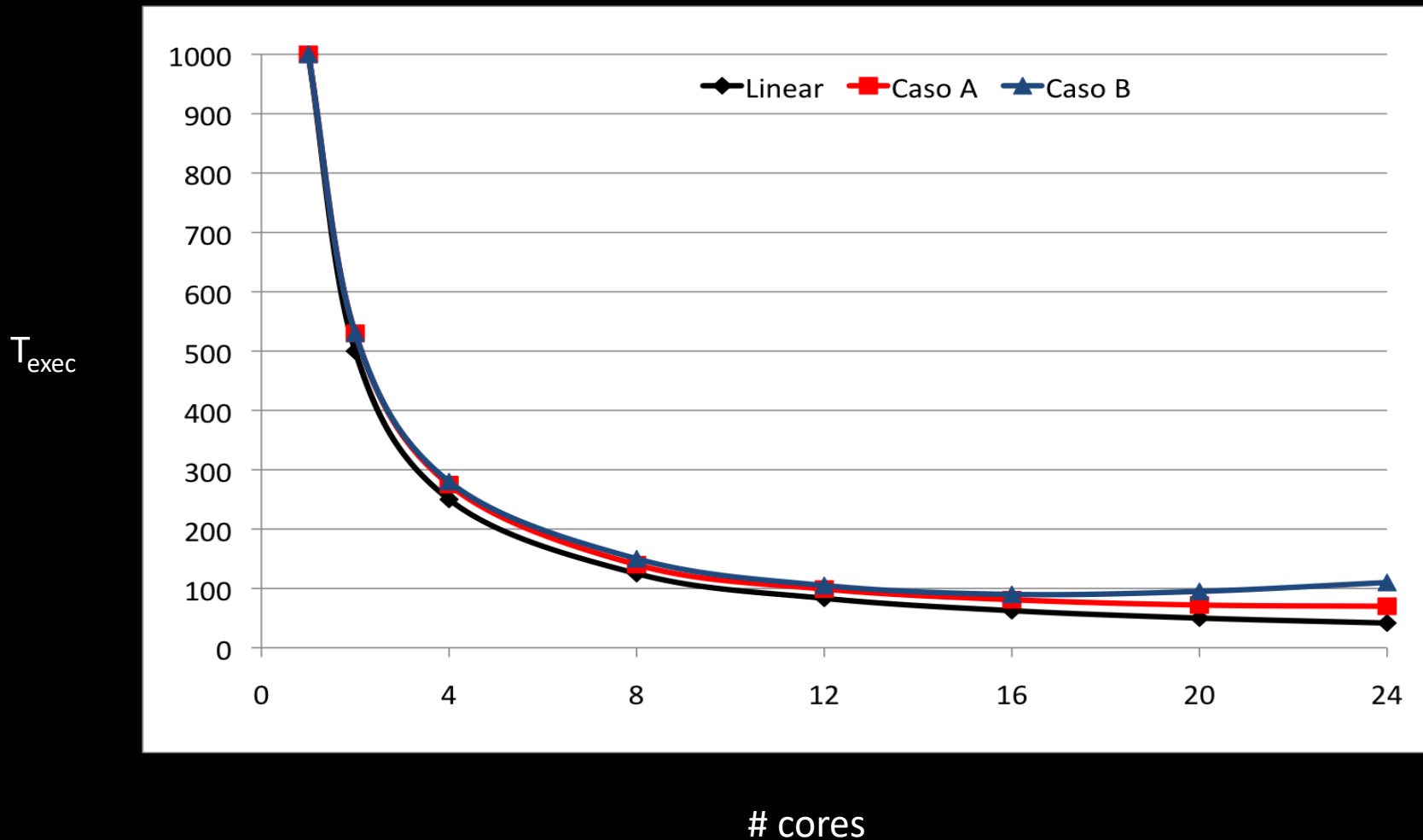


2 – carga variável para diferentes iterações



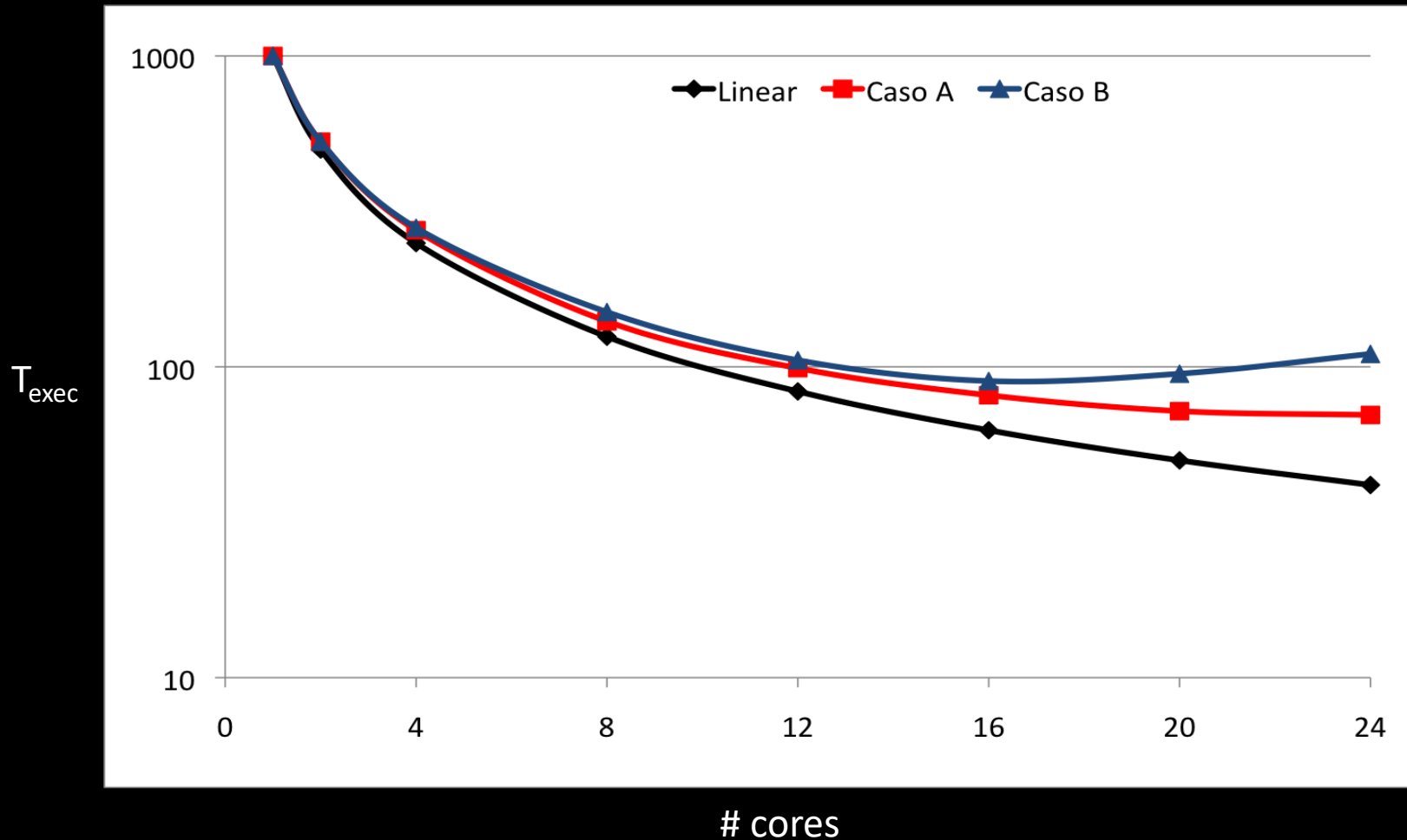
desempenho – tempo de execução

- OBJECTIVO: diminuir o tempo de execução



desempenho – tempo de execução

- Escala Logarítmica



desempenho – *speed up*

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

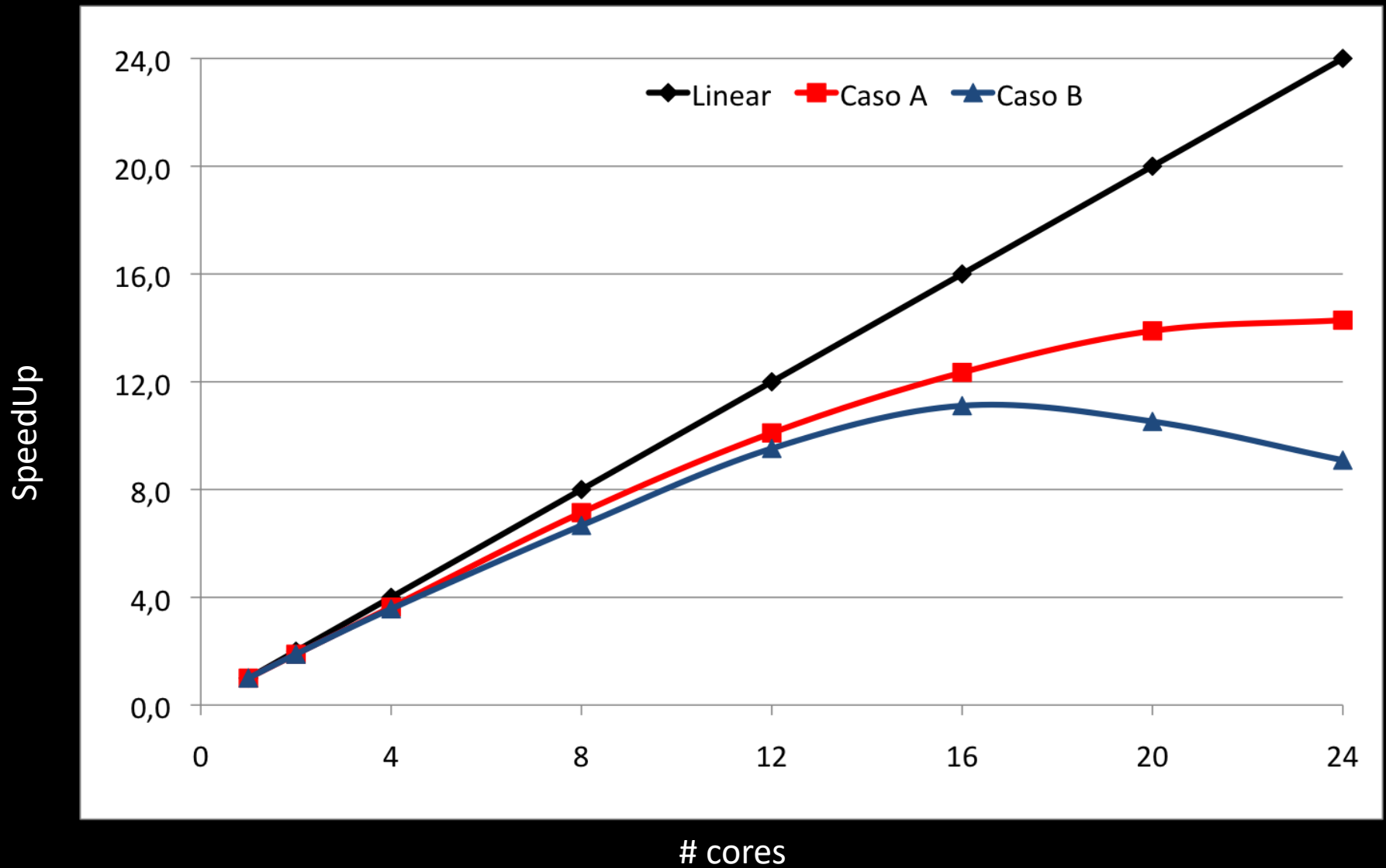
p – número de processadores

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

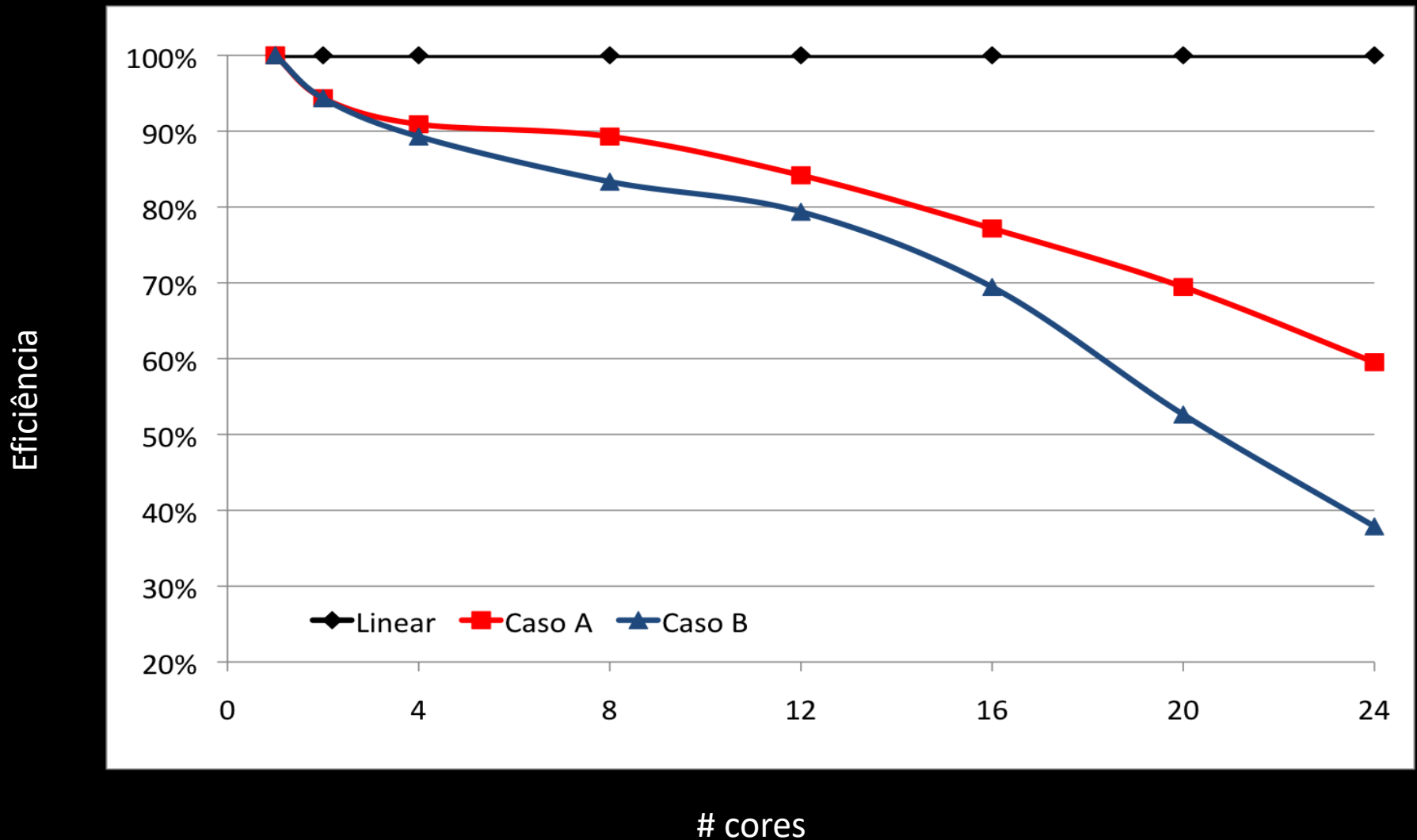
$$E_p = \frac{S_p}{p}$$

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