

OpenMP

Data: [19-12-2022](#)

PDFs: [09-OpenMP.pdf](#); [link](#)

Tags: [#ARQC](#) [#SoftwareEngineering](#) [#C](#)

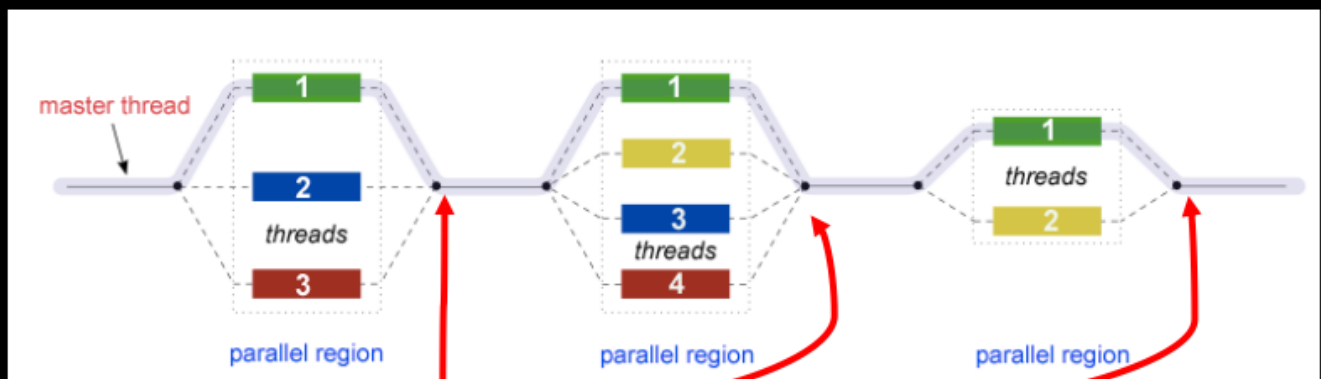
Open Multi Processing

- API para expressar paralelismo multi-threaded e de memória partilhada
- Objetivos: normalização; portabilidade; fácil utilização.

Modelo de execução

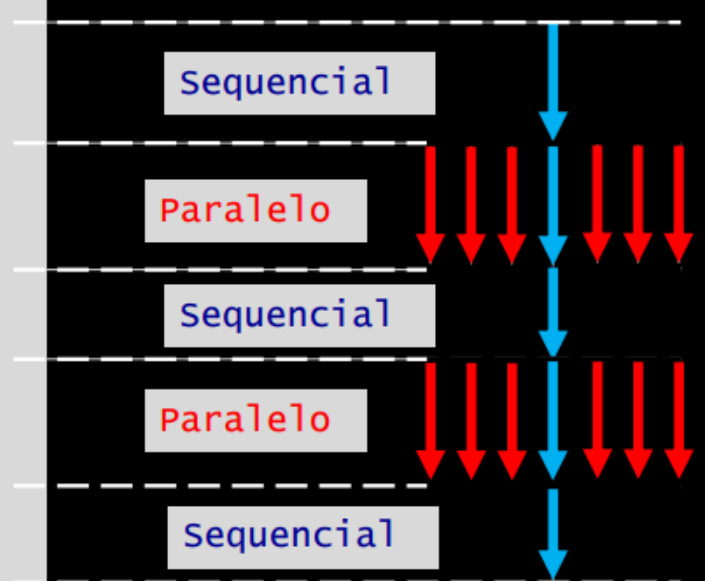
Criação explícita de blocos de código executados por um grupo de threads.

Modelo Fork & Join



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

```
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");
```



Diretivas

Paralelismo especificado usando diretivas *embeded* no código:

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

Cada diretiva aplica-se ao bloco de instruções que se lhe segue:

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

O compilador ignora as diretivas se não for usada a opção que ativa o OpenMP. Exemplo:

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

Diretiva 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

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

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

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 ...	

```
#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!
>_
```

Diretiva single

```

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");

```

Sequencial

Paralelo

Sequencial

Paralelo

Sequencial

```

#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
>_

```

```
#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
>_
```

Quote ▾

A worksharing region has no barrier on entry; however, an implied barrier exists at the end of the worksharing region, unless a `nowait` clause is specified. If a `nowait` clause is present, an implementation may omit the barrier at the end of the worksharing region. In this case, threads that finish early may proceed straight to the instructions following the worksharing region without waiting for the other members of the team to finish the worksharing region, and without performing a flush operation.

Portanto, no segundo caso, o código que imprime `... thread 1` pode ser executado antes do código que imprime `Thread 0...`, visto que não há uma barreira implícita de sincronização.

Loop construct: diretiva `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 ativas no grupo atual:
 - 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*, exceto 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];
```

```
#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
for (i=0; i<25000; i++) A[i] = B[i] + C[i];	for (i=25000; i<50000; i++) A[i] = B[i] + C[i];	for (i=50000; i<75000; i++) A[i] = B[i] + C[i];	for (i=75000; i<100000; i++) A[i] = B[i] + C[i];

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

- 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 percecionado (pelo utilizador) - tende a diminuir com a adição de múltiplos cores, se o tempo de execução diminuir

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

$$T_{exec} = \#I \cdot CPI_{percieved} / f$$

Com a programação *multithreaded* o número de instruções executadas aumenta, devido à gestão do paralelismo. O $CPI_{percieved}$ 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 $CPI_{percieved}$ reduz à medida que o número de *cores* aumenta

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.

Variáveis privadas:

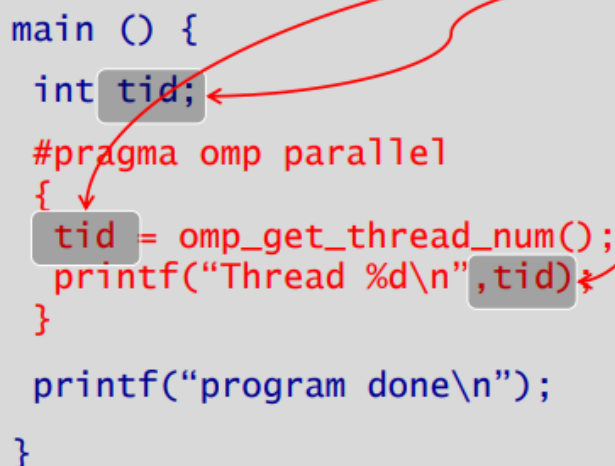
- declaradas dentro de um bloco paralelo

- explicitamente marcadas como privadas
- índices dos ciclos associados a uma diretiva

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

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

- 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];
```

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");
}
```

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");  
}
```

- 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!!

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

Caso 2: Read after write.

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**

Execução de regiões críticas \implies ordem sequencial.

controle 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; }
```

No primeiro caso, $x = x+1$ e $y = y+1$ não podem acontecer em paralelo.

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

Só se aplica a operações simples (atômicas).

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

```
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;
}
```

Quote

The atomic construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values.

Syntax

C/C++

The syntax of the **atomic** construct takes either of the following forms:

```
#pragma omp atomic [read | write | update |  
capture] [seq_cst] new-line  
expression-stmt
```

or:

```
#pragma omp atomic capture [seq_cst] new-line  
structured-block
```

where *expression-stmt* is an expression statement with one of the following forms:

- If clause is **read**:
`v = x;`
- If clause is **write**:
`x = expr;`
- If clause is **update** or not present:
`x++;`
`x--;`
`++x;`
`--x;`
`x binop= expr;`
`x = x binop expr;`
`x = expr binop x;`

Redução

Quote

The reduction clause specifies a reduction-identifier and one or more list items. For each list item, a private copy is created in each implicit task or SIMD lane, and is initialized with the initializer value of the reduction-identifier. After the end of the region, the original list item is updated with the values of the private copies using the combiner associated with the reduction-identifier.

The reduction clause can be used to perform some forms of recurrence calculations (involving mathematically associative and commutative operators) in parallel. For parallel and worksharing constructs, a private copy of each list item is created, one for each implicit task, as if the private clause had been used. For the simd construct, a private copy of each list item is created, one for each SIMD lane as if the private clause had been used. For the teams construct, a private copy of each list item is created, one for each team in the league as if the private clause had been used. The private copy is then initialized as specified above. At the end of the region for which the reduction clause was specified, the original list item is updated by combining its original value with the final value of each of the private copies, using the combiner of the specified reduction-identifier.

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

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;  
}
```

- 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!

- 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
```

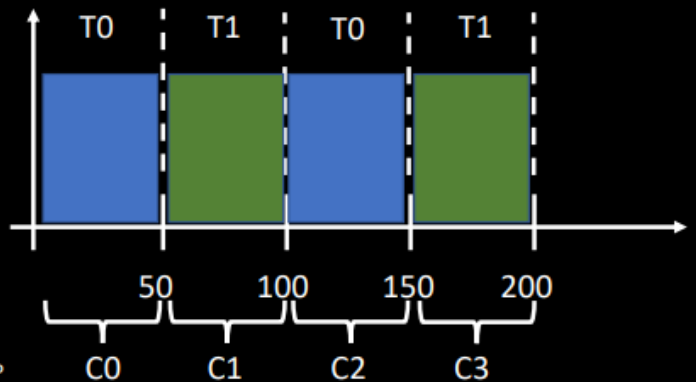
Escalonamento Estático

```
#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*

#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];
}
```

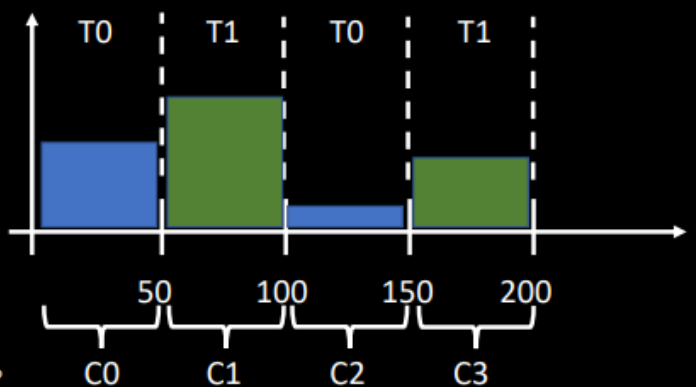


AC – OpenMP

38

#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]);
}
```

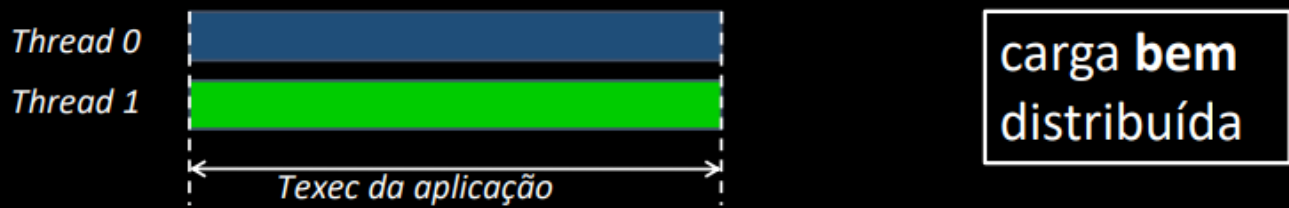


AC – OpenMP

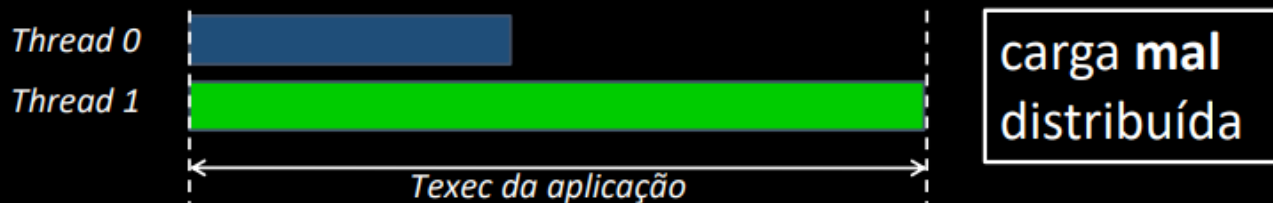
39

#pragma omp parallel for schedule(static)

1 – carga uniforme para todas as iterações



2 – carga variável para diferentes iterações



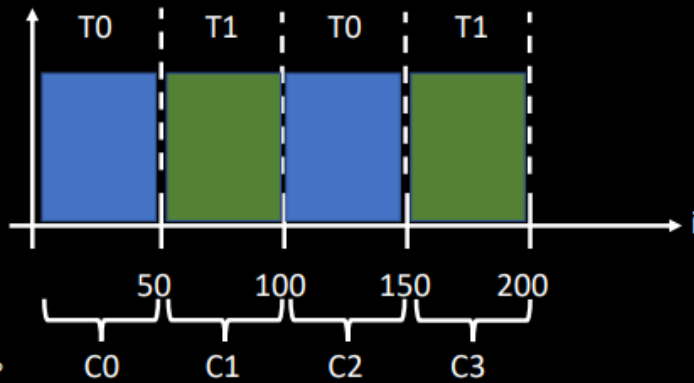
Escalonamento Dinâmico

```
#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.

#pragma for schedule(dynamic, <chunksize>)

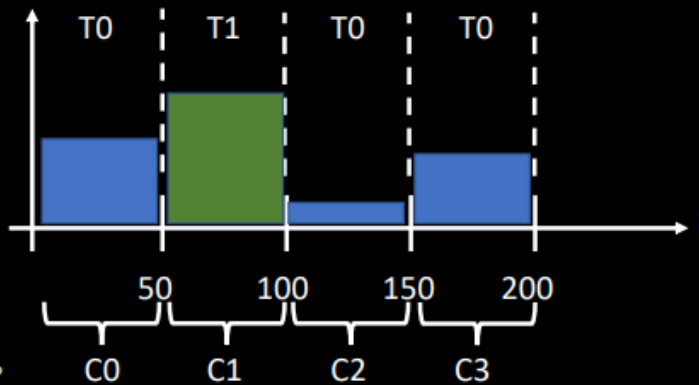
```
#pragma omp parallel for schedule(dynamic, 25)  
num_threads(2)  
{  
  for (i=0; i< 200 ; i++)  
    a[i] *= a[i];}
```



AC – OpenMP

42

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



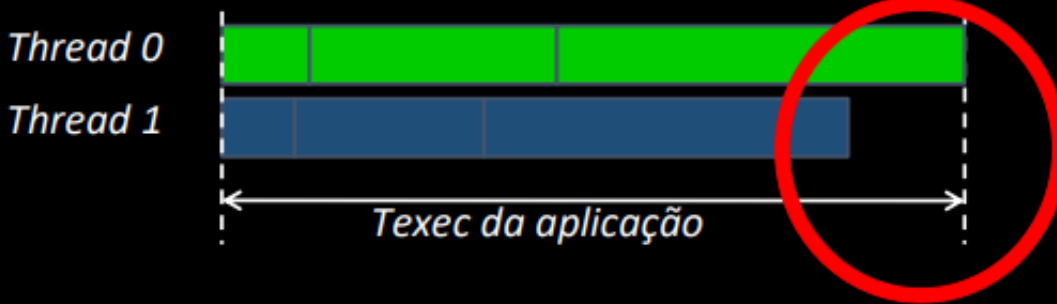
AC – OpenMP

43

1 – carga uniforme para todas as iterações



2 – carga variável para diferentes iterações



Quote

If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space that is then divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

The iteration count for each associated loop is computed before entry to the outermost loop. If execution of any associated loop changes any of the values used to compute any of the iteration counts, then the behavior is unspecified. The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined. A worksharing loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if the associated loop(s) were executed by a single thread. The schedule clause specifies how iterations of the associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team. Each thread executes its assigned chunk(s) in the context of its implicit task. The `chunk_size` expression is evaluated using the original list items of any variables that are made private in the loop construct. It is unspecified whether, in what order, or how many times, any side effects of the evaluation of this expression occur. The use of a variable in a schedule clause expression of a loop construct causes an implicit reference to the variable in all enclosing constructs. Different loop regions with the same schedule and iteration count, even if they occur in the same parallel region, can distribute iterations among threads differently. The only exception is for the static schedule as specified in Table 2-1. Programs that depend on which thread executes a particular iteration under any other circumstances are non-conforming.

static	<p>When <code>schedule(static, chunk_size)</code> is specified, iterations are divided into chunks of size <code>chunk_size</code>, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.</p> <p>When no <code>chunk_size</code> is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. Note that the size of the chunks is unspecified in this case.</p> <p>A compliant implementation of the static schedule must ensure that the same assignment of logical iteration numbers to threads will be used in two loop regions if the following conditions are satisfied: 1) both loop regions have the same number of loop iterations, 2) both loop regions have the same value of <code>chunk_size</code> specified, or both loop regions have no <code>chunk_size</code> specified, 3) both loop regions bind to the same parallel region, and 4) neither loop is associated with a SIMD construct. A data dependence between the same logical iterations in two such loops is guaranteed to be satisfied allowing safe use of the nowait clause.</p>
dynamic	<p>When <code>schedule(dynamic, chunk_size)</code> is specified, the iterations are distributed to threads in the team in chunks as the threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.</p> <p>Each chunk contains <code>chunk_size</code> iterations, except for the last chunk to be distributed, which may have fewer iterations.</p> <p>When no <code>chunk_size</code> is specified, it defaults to 1.</p>

Speed Up

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!

Eficiência

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%

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