



OpenMP

Enabling HPC since 1997

The OpenMP API specification for parallel programming

[Home](#)[Specifications](#)[Blog](#)[Community](#)[Resources](#)[News & Events](#)[About](#)

OpenMP ARB Members

The OpenMP API is jointly defined by a group of major computer hardware and software vendors and major parallel computing user facilities.

[READ MORE](#)

Latest News



An Interview with InsideHPC

Dec 14, 2017

View the insideHPC video from SC17 where Michael Klemm discuss the OpenMP ARB, the latest Technical Report 6 (TR6) and asks for feedback via the OpenMP Forum. [more](#)

**OpenMPCon 2017 Presentations
Now Available for Download**

SC17 In-Booth Talks Video and Slides Available

Nov 27, 2017

Twelve in-booth talks from SC17 - Denver are now viewable from our [SC'17 Presentations Page](#).

Using OpenMP - The Next Step

Oct 01, 2017

Release of OpenMP Technical Report 6 (TR6) Addresses Top User Requests

Nov 13, 2017

OpenMP ARB Technical Report 6 (TR6) extends TR4 adding a number of key features and is a preview of OpenMP 5.0, expected in November 2018. [more](#)

**OpenMP Accelerator Support for
GPUs**

@OpenMP_ARB

OpenMP ARB
@OpenMP_ARB
Great work!
13 Feb 2018

OpenMP ARB
@OpenMP_ARB
"As we prepare for the next gen of supercomputers and #GPUs, #OpenMP is growing to meet challenges of programming scientific apps in a world of accelerators, unified memory, and explicitly

Specifications

[Home](#) > [Specifications](#)



OpenMP 4.5 Specifications

- [OpenMP 4.5 Complete Specifications \(Nov 2015\) pdf](#)
 - [OpenMP 4.5 Discussion Forum](#)
- [OpenMP 4.5 Summary Card – C/C++ \(Nov 2015\) pdf](#)
- [OpenMP 4.5 Summary Card – Fortran \(Nov 2015\) pdf](#)
- [OpenMP 4.5 Examples \(Nov 2016\) pdf](#)
 - [OpenMP 4.5 Examples Discussion Forum](#)



OpenMP 4.0 Specifications

- [OpenMP 4.0 Complete Specifications \(July 2013\) \(PDF\)](#)
- [OpenMP 4.0 Discussion Forum](#)
- [OpenMP 4.0 Summary Card – C/C++ \(October 2013 PDF\)](#)
- [OpenMP 4.0 Summary Card – Fortran \(October 2013 PDF\)](#)
- [OpenMP Examples 4.0.2 \(March 2015 PDF\)](#)
- [OpenMP 4.0.1 Examples \(February 2014 PDF\)](#)

Active Technical Report Drafts and Proposals

- **TR6: OpenMP Version 5.0 Preview 2**

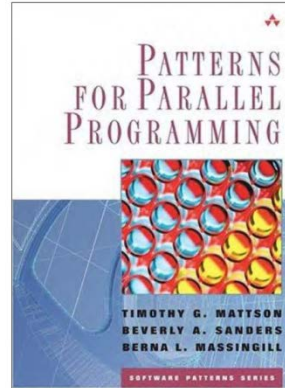
This Technical Report is the latest draft of the OpenMP Version 5.0 specifications. (Nov 2017 PDF)

[TR6 Discussion Forum](#)

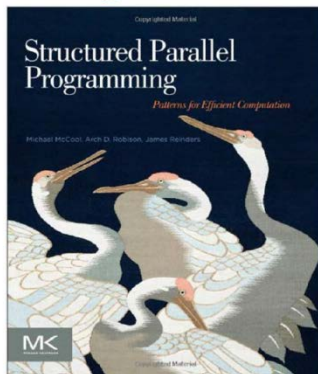
Tutorial:<https://computing.llnl.gov/tutorials/openMP/>



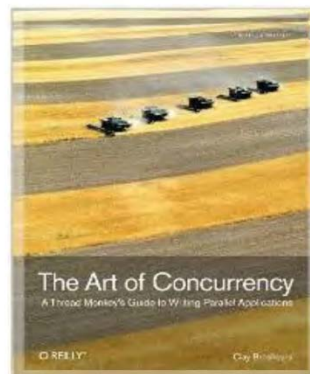
- A book about OpenMP by a team of authors at the forefront of OpenMP's evolution.



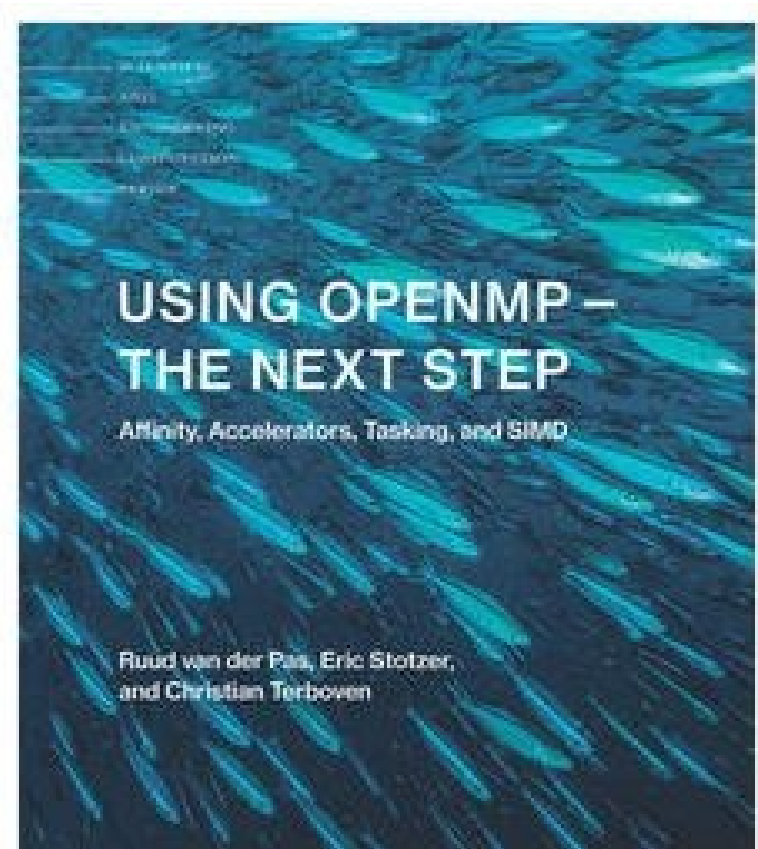
- A book about how to “think parallel” with examples in OpenMP, MPI and java



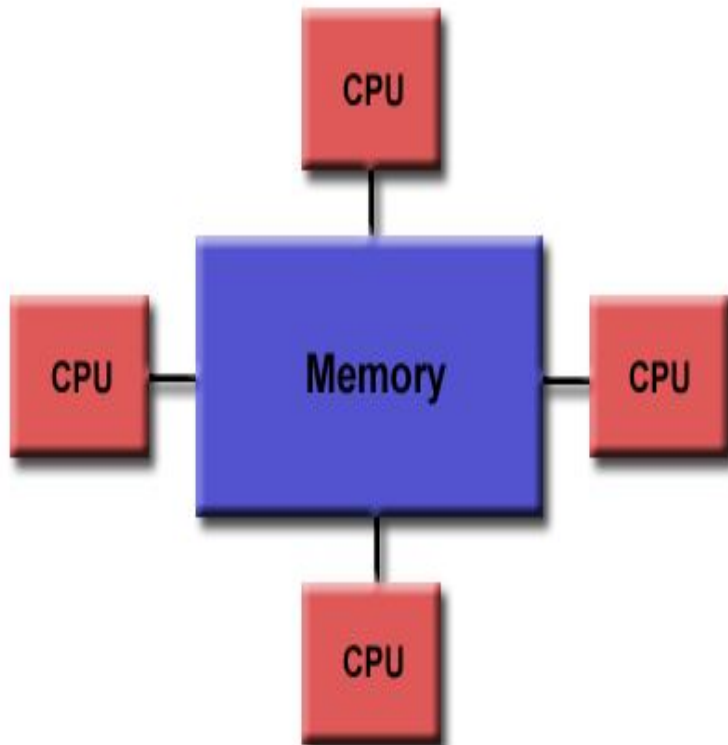
This book explores key patterns with Cilk, TBB, OpenCL, and OpenMP (by McCool, Robison, and Reinders)



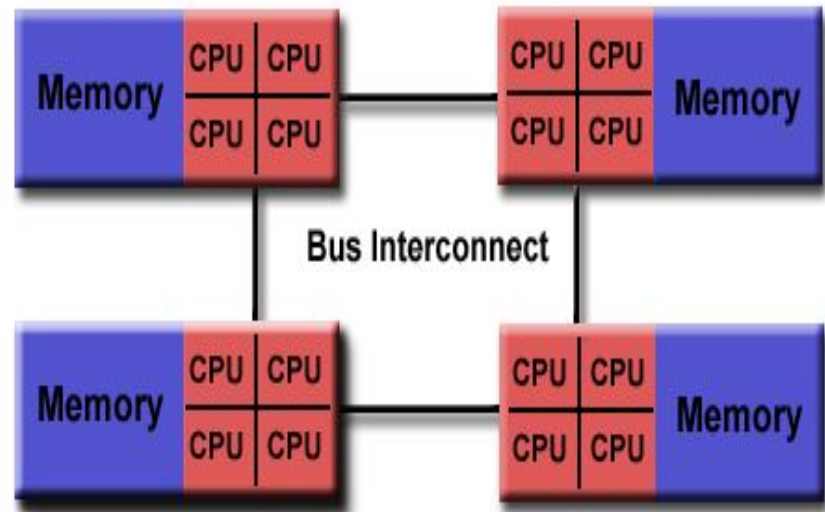
An introduction to and overview of multithreaded programming in general (by Clay Breshears)



Modelo de Programação do OpenMP

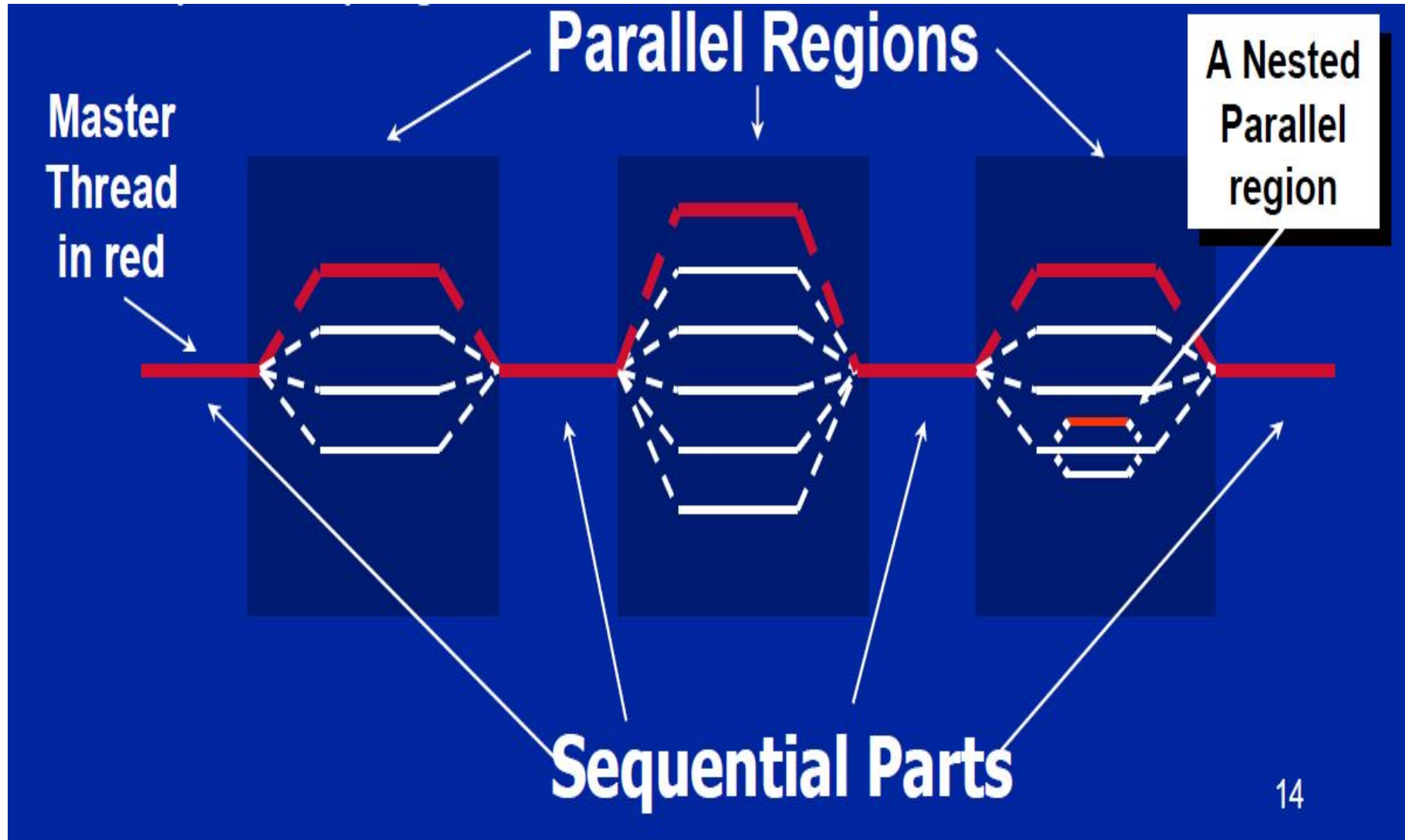


Uniform Memory Access

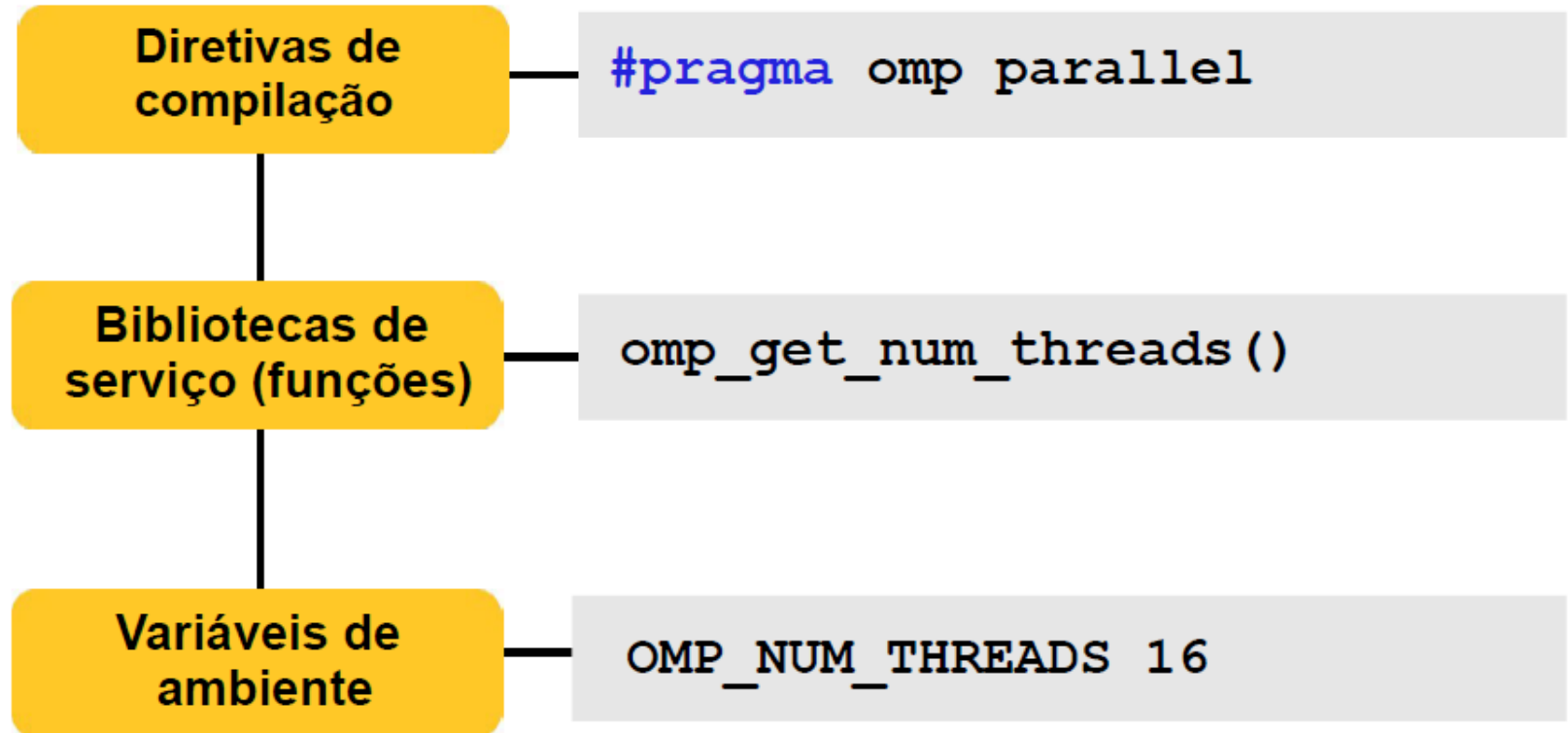


Non-Uniform Memory Access

Modelo de programação



Estrutura do OpenMP API



Diretivas de Compilação

- **Diretivas** - Consiste em uma linha de código com significado “especial” para o compilador.

C/C++

```
#pragma omp parallel
```

FORTRAN

```
!$OMP OMP PARALLEL
```

Compilação

nome do arquivo: teste.c

- **icpc** **-openmp** teste.c | -o exec
- **gcc** **-fopenmp** teste.c -o exec
- **cc** **-xopenmp=parallel** teste.c -o exec

Compiler / Platform	Compiler Commands	OpenMP Flag
Intel Linux	icc icpc ifort	-qopenmp -openmp
GNU Linux IBM Blue Gene CORAL EA	gcc g++ g77 gfortran	-fopenmp
PGI Linux CORAL EA	pgcc pgCC pgf77 pgf90	-mp
Clang Linux CORAL EA xlflang CORAL EA	clang clang++ xlflang	-fopenmp
IBM XL Blue Gene *	bgxlc_r, bgcc_r bgxlc_r, bgxlc++_r bgxlc89_r bgxlc99_r bgxlf_r bgxlf90_r bgxlf95_r bgxlf2003_r	-qsmp=omp
IBM XL CORAL EA *	xlC_r xlC_r, xlc++_r xlf_r xlf90_r xlf95_r xlf2003_r xlf2008_r	-qsmp=omp

Platform	Compiler	Version Flag	Default Version	Supports
Linux	Intel C/C++, Fortran	--version	16.0.3	OpenMP 4.0
	GNU C/C++, Fortran	--version	4.4.7 (TOSS 2) 4.9.3 (TOSS 3)	OpenMP 3.0 OpenMP 4.0
	PGI C/C++, Fortran	-v --version	8.0.1 (TOSS 2) 16.9-0 (TOSS 3)	OpenMP 3.0 OpenMP 3.1
	Clang C/C++	--version	3.7.0 (TOSS 2) 4.0.0 (TOSS 3)	OpenMP 3.1 Some OpenMP 4.0 and 4.5
BG/Q	IBM XL C/C++	-qversion	12.1	OpenMP 3.1
	IBM XL Fortran	-qversion	14.1	OpenMP 3.1
	GNU C/C++, Fortran	--version	4.4.7	OpenMP 3.0
CORAL EA	IBM XL C/C++	-qversion	14.01 beta	OpenMP 4.5
	IBM XL Fortran	-qversion	16.01 beta	OpenMP 4.5
	GNU C/C++	--version	4.9.3	OpenMP 4.0
	GNU Fortran	--version	4.8.5	OpenMP 3.1
	PGI C/C++, Fortran	-v --version	17.4-0	OpenMP 3.1
	Clang C/C++	--version	4.0 beta	OpenMP 4.5
	xlflang Fortran	--version	4.0 beta	OpenMP 4.5

Construtor Paralelo

```
#pragma omp parallel
```

- Informa ao compilador a existência de uma região que deve ser executada em paralelo.

```
#pragma omp parallel
{
    for (i = 0; i < n; i++)
        c[i] = a[i]+b[i];
}
```

Constructor “ PARALLEL”

```
[professor@sdumont12 exemplo_1]$ cat exemplo_01.c
```

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

```
int main()
```

```
{
```

```
    #pragma omp parallel
```

```
    {
```

```
        printf("Hello World\n");
```

```
    }
```

```
}
```

Primeiro Exemplo no SDUmont

- Para compilar executar o batch:

```
$/compilar.sh
```

- OU executar os comandos abaixo para cada exemplo:

```
$ load gcc/8.3
```

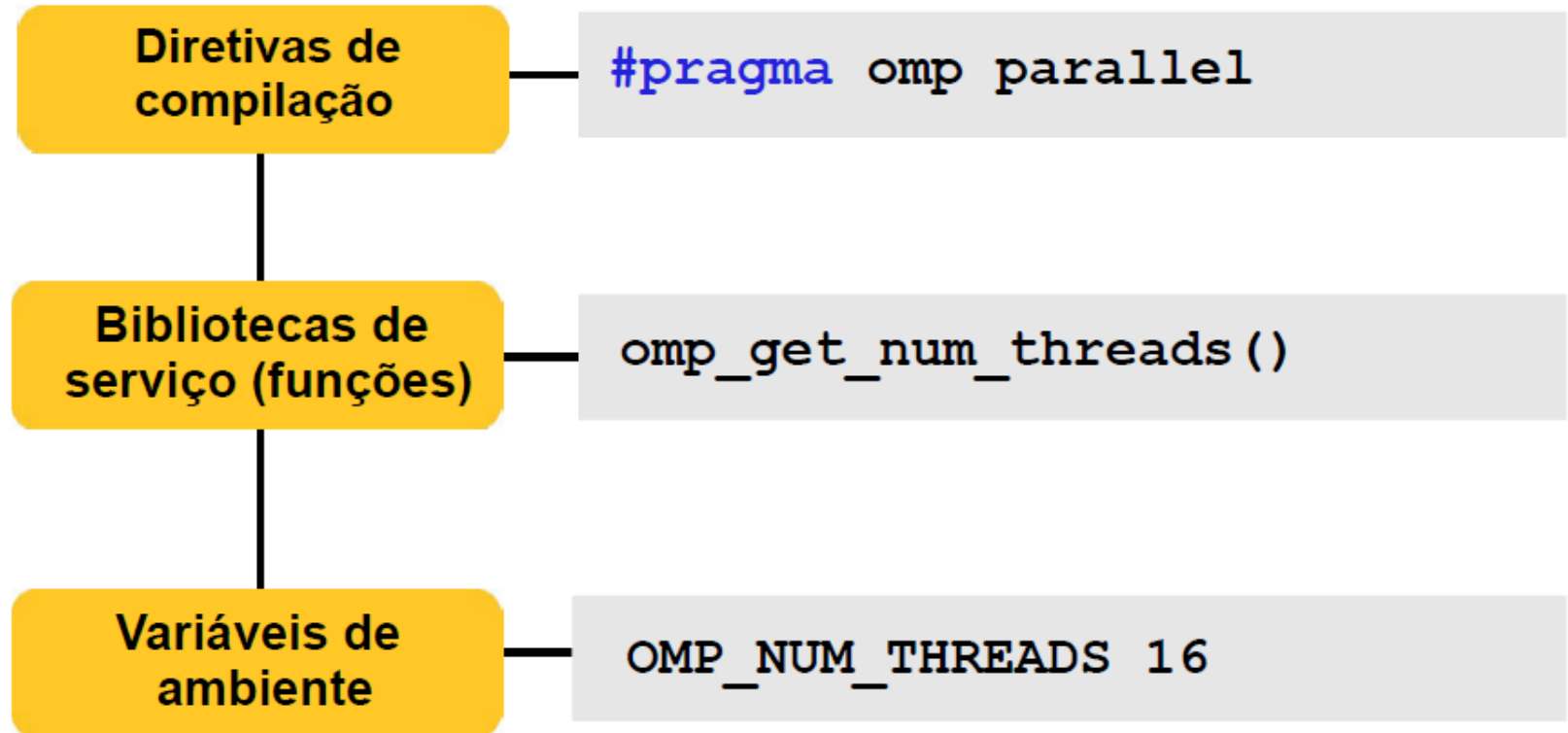
```
$ gcc -fopenmp exemplo_01.c -o exemplo_01
```

- Para executar:

```
$ ./exemplo_01
```

- Porque foram a saída gera 24 textos “Hello Word”?

Estrutura do OpenMP API



Variáveis de ambiente no notebook

```
$ ./exemplo_01
```

```
$ export OMP_NUM_THREADS= 3
```

Execução no SDumont

```
$cd $SCRATCH
```

```
$ ./compilar.sh
```

```
$ sbatch run.sh 5 exemplo_01  
  (submissão)
```

```
$ squeue -u professor
```

```
$ scancel JOBID
```

Funções de Ambiente de Execução biblioteca omp.h

```
void omp_set_num_threads(int num)
```

- Define o número de *threads* padrão a serem usadas na região paralela.

```
int omp_get_num_threads()
```

- Retorna o número de *threads* ativas na região paralela onde ela foi chamada.

```
int omp_get_max_threads()
```

- Retorna o número de *threads* disponíveis para executar a região paralela.

```
int omp_get_thread_num()
```

- Retorna o identificador da thread relativo ao grupo ao qual ela pertence.

```
int omp_get_num_procs()
```

- Retorna o número de processadores disponíveis no momento da chamada da função.

exemplo_01_num_threads.c

```
[professor@sdumont12 exemplo_1]$ cat exemplo_01_num_threads.c
```

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

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

```
int main()
```

```
{
```

```
    #pragma omp parallel num_threads(10)
```

```
    {
```

```
        printf("omp_get_thread_num = %d\n", omp_get_thread_num());
```

```
    #pragma omp single
```

```
        printf("omp_get_num_procs = %d\n", omp_get_num_procs());
```

```
    }
```

```
}
```

exemplo_01_parallel.c

```
int omp_in_parallel()
```

- Verifica se a região onde a função está sendo chamada é uma região paralela.
- Se a função for chamada dentro de uma região paralela, o retorno da função será um valor diferente de zero.
- Se a função for chamada dentro de uma região serial, o retorno da função será zero.

exemplo_02_get_nested.c

```
void omp_set_nested(int num)
```

- Habilita ou desabilita o paralelismo aninhado.
 - > num = 0, desabilita o paralelismo aninhado
 - > num \neq 0, habilita o paralelismo aninhado

```
int omp_get_nested(void)
```

- Retorna um valor que indica se o paralelismo aninhado está ativado ou não.
- Se o valor retornado for zero, o paralelismo aninhado está desativado. Se o valor retornado for diferente de zero, o paralelismo aninhado está ativado.

exemplo_01_omp_set_num_threads.c

```
void omp_set_num_threads(int num)
```

- Define o número de *threads* padrão a serem usadas na região paralela.

```
int omp_get_num_threads()
```

- Retorna o número de *threads* ativas na região paralela onde ela foi chamada.

```
int omp_get_max_threads()
```

- Retorna o número de *threads* disponíveis para executar a região paralela.

exemplo_01_set_num_threads.c

```
[professor@sdumont12 exemplo_1]$ cat exemplo_01_set_num_threads.c

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

int main()
{
    omp_set_num_threads(8);

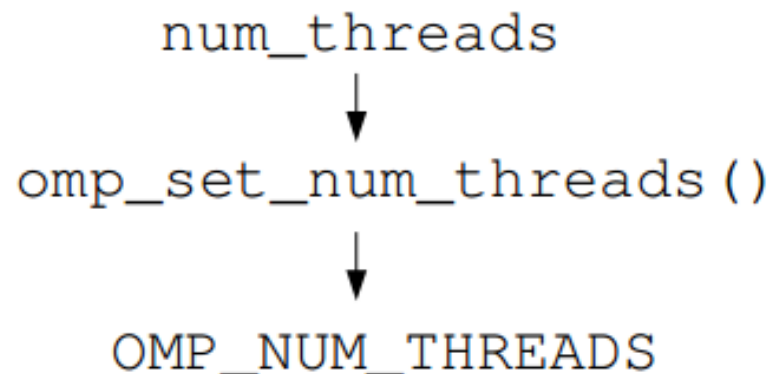
    #pragma omp parallel num_threads(4)
    {
        printf("Hello World\n");
        #pragma omp single
        {
            printf(" Numero de threads= %d\n", omp_get_num_threads());
            printf(" Numero max de threads= %d\n", omp_get_max_threads());
        }
    }
}
```

Alteração do padrão de execução: Cláusula, Função da OMP.h , Variável de ambiente ,

- Variável de ambiente:

```
BASH : export OMP_NUM_THREADS = 8  
CSH  : setenv OMP_NUM_THREADS 8
```

- Diagrama de Precedência:



Construtores de Trabalho

- São responsáveis pela distribuição de trabalho entre as *threads* e determinam como o trabalho será dividido.

```
#pragma omp for
```

```
#pragma omp sections
```

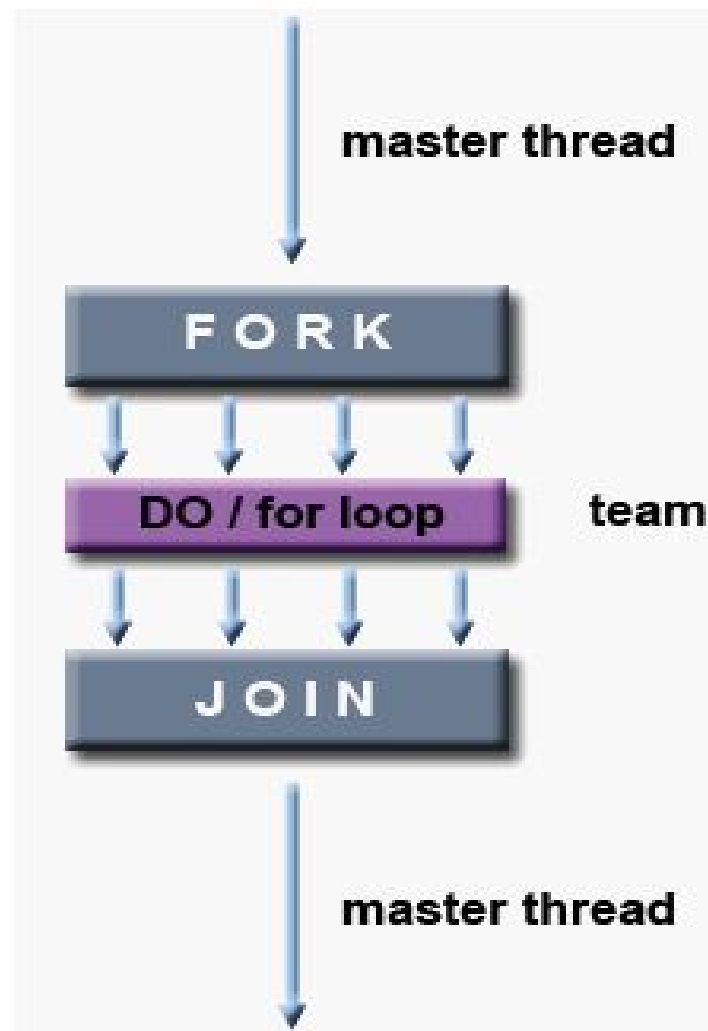
```
#pragma omp single
```

Construtor de Trabalho

```
#pragma omp for [cláusula, ...]
```

- Esse construtor é responsável pela divisão das iterações do laço a serem realizadas entre as *threads*.
- O número de iterações do laço deve ser previamente conhecido.

DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".



exemplo_04_1.c

```
[professor@sdumont12 exemplo_4]$ cat exemplo_04_1.c
```

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

int main (int argc, char * argv[])
{
    int n,id,i;

    n=10;

    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < n; i++)
        {
            id= omp_get_thread_num();
            printf("Thread = %d, i= %d\n",id,i);
        }
    }
}
```

exemplo_04.1.c

```
[professor@sdumont12 exemplo_4]$ ./exemplo_04_1
Thread = 0, i= 0
Thread = 0, i= 1
Thread = 1, i= 4
Thread = 1, i= 5
Thread = 1, i= 6
Thread = 2, i= 7
Thread = 2, i= 8
Thread = 2, i= 9
Thread = 0, i= 2
Thread = 0, i= 3
[professor@sdumont12 exemplo_4]$ ./exemplo_04_1
Thread = 0, i= 0
Thread = 0, i= 1
Thread = 0, i= 2
Thread = 0, i= 3
Thread = 1, i= 4
Thread = 1, i= 5
Thread = 1, i= 6
Thread = 2, i= 7
Thread = 2, i= 8
Thread = 2, i= 9
[professor@sdumont12 exemplo_4]$ ./exemplo_04_1
Thread = 0, i= 0
Thread = 0, i= 1
Thread = 1, i= 4
Thread = 1, i= 5
Thread = 1, i= 6
Thread = 0, i= 2
Thread = 0, i= 3
Thread = 2, i= 7
Thread = 2, i= 8
Thread = 2, i= 9
```

```
#pragma omp for [clause ...] newline
    schedule (type [,chunk])
    ordered
    private (list)
    firstprivate (list)
    lastprivate (list)
    shared (list)
    reduction (operator: list)
    collapse (n)
    nowait
```

for_loop

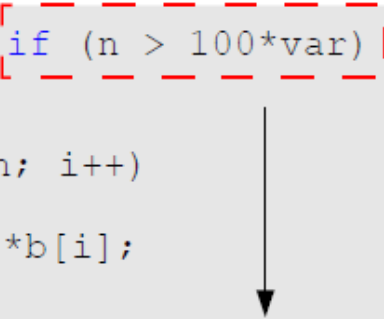
Cláusula IF : exemplo_03.c

`if` (expressão lógica)

- Se a expressão lógica for verdadeira a região paralela será executada por mais de uma *thread*.

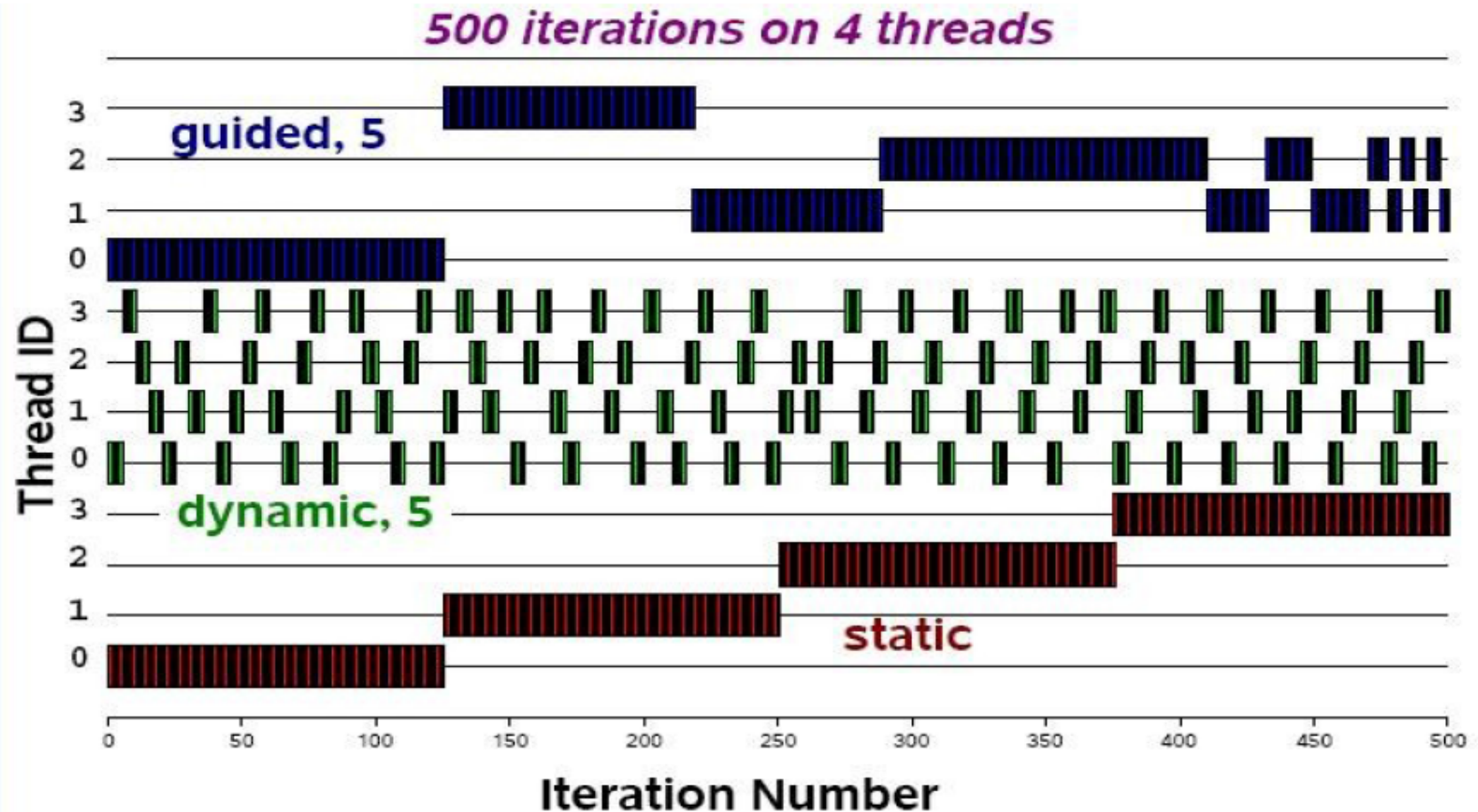
EXEMPLO:

```
#pragma omp parallel if (n > 100*var) |
{
    #pragma omp for
    for (i = 0; i < n; i++)
    {
        c[i] = a[i]*b[i];
    }
}
```



```
if (n > 100*var)
    região paralela ativa
else
    região paralela inativa
```

Escalonamento das cláusulas do divisor de trabalho FOR



Cláusula schedule

`$/exemplo_04_1` (executa configuração “default”)

`$/exemplo_04_2` (cláusula `schedule dynamic, chunk=5`)

`$/exemplo_04_4` (cláusula `schedule dynamic, chunk=1`)

`$time ./exemplo_04_3` (runtime: usa variável de ambiente)

`$export OMP_NUM_THREADS=x`

`$export OMP_SCHEDULE="mode, chunk"`

Clausula Reduction

```
reduction ( operador : lista de variáveis)
```

- Uma cópia de cada variável é criada para cada *thread*.
- Ao final da região paralela definida pelo construtor, a lista de variáveis original é atualizada com os valores da cópia privada de cada *thread* usando o operador especificado.

Operador	Valor Inicial
+	0
*	1
-	0
^	0

Operador	Valor Inicial
&	~0
	0
&&	1
	0

exemplo_8/exemplo_08.c

```
[professor@s dumont12 exemplo_8]$ cat exemplo_08.c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

////////////////////////////////////
//          PRODUTO ESCALAR
////////////////////////////////////

//Funcao responsavel pelo preenchimento dos vetores
void InicializaVetores(int num,int *V1,int *V2)
{
    int i;
    for(i=0;i<num;i++)
    {
        V1[i] = V2[i] = 1;
    }
}

int main (int argc, char * argv[])
{
    int *V1,*V2;
    int i,j,id;
    int num_threads;
    int par,num,dot;

    printf("\n\n===== \n");
    printf("\tPRODUTO ESCALAR\n");
    printf("===== \n\n");

    //Dimensao dos Vetores

    num = atoi(argv[1]);

    //Alocacao dinamica dos vetores
    V1=(int*)malloc(num*sizeof(int));
    V2=(int*)malloc(num*sizeof(int));

    //Inicializacao dos vetores
    InicializaVetores(num,V1,V2);

    //Inicilizacao da variavel dot
    dot = 0;
    //Calculo do produto escalar
    #pragma omp parallel
    {
        #pragma omp single
        printf("numero de threads = %d   id = %d\n",omp_get_num_threads(),omp_get_thread_num());
        #pragma omp for reduction(+:dot)
        for(i=0;i<num;i++)
            dot += V1[i]*V2[i];
        #pragma omp single
        printf(" \n\n End Of Execution ::: dot = %d\n\n\n",dot);
    }
    return 0;
}
```

exemplo_8/exemplo_08_corrida.c

```
[professor@sдумont12 exemplo_8]$ cat exemplo_08_corrida.c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

////////////////////////////////////////
//                                PRODUTO ESCALAR                                //
////////////////////////////////////////

//Funcao responsavel pelo preenchimento dos vetores
void InicializaVetores(int num,int *V1,int *V2)
{
    int i;
    for(i=0;i<num;i++)
    {
        V1[i] = V2[i] = 1;
    }
}

int main (int argc, char * argv[])
{
    int *V1,*V2;
    int i,j,id;
    int num_threads;
    int par,num,dot;

    printf("\n\n=====\\n");
    printf("\tPRODUTO ESCALAR\\n");
    printf("=====\\n\\n");

    //Dimensao dos Vetores
    num = atoi(argv[1]);

    //Alocacao dinamica dos vetores
    V1=(int*)malloc(num*sizeof(int));
    V2=(int*)malloc(num*sizeof(int));

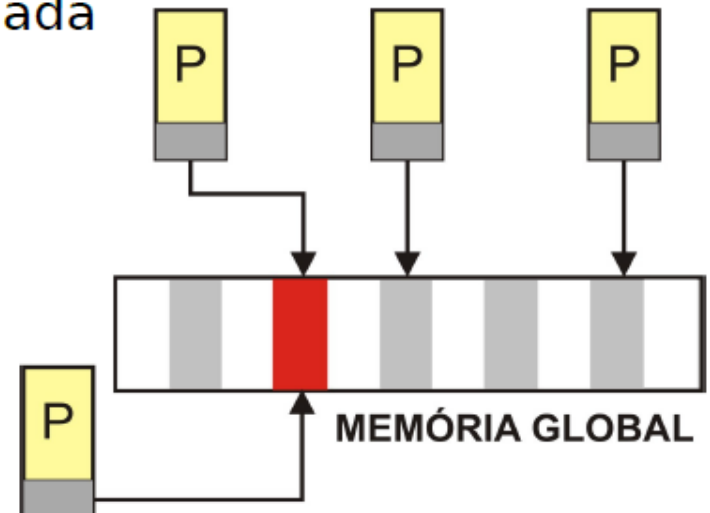
    //Inicializacao dos vetores
    InicializaVetores(num,V1,V2);

    //Inicilizacao da variavel dot
    dot = 0;

    //Calculo do produto escalar
    #pragma omp parallel
    {
        #pragma omp for
        for(i=0;i<num;i++)
            dot += V1[i]*V2[i];
    }
    printf(" \n\n End Of Execution ::: dot = %d\\n\\n\\n",dot);
    return 0;
}
```

Condição de corrida

- > Acontece quando duas ou mais *threads* tentam atualizar, ao mesmo tempo, uma mesma variável ou quando uma *thread* atualiza uma variável e outra *thread* acessa o valor dessa variável ao mesmo tempo.
- > Dessa forma, as diretivas de sincronização garantem que o acesso ou atualização de uma determinada variável compartilhada aconteça no momento certo.



Exemplo_08.c

Thread 0

load dot
dot=dot+1
store dot

Thread 1

load dot
dot=dot+1
Store dot

Thread 0

load dot
dot=dot+1
Store dot

Thread 2

load dot
dot=dot+1
Store dot

Thread 3

load dot
dot=dot+1
Store dot

Thread 2

load dot
dot=dot+1
Store dot

Para dot=0 no inicio no final de 6 execuções temos dot=6

Thread 0
↓
load dot
dot=dot+1
store dot

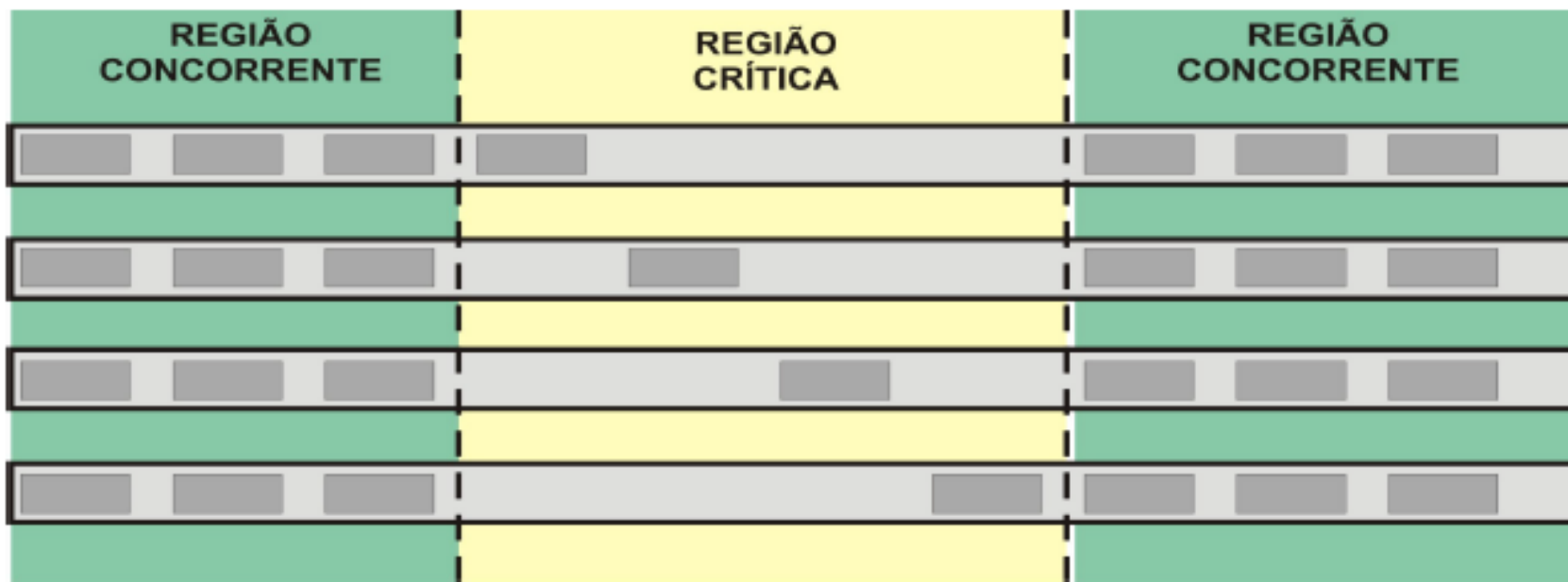
Thread 1
↓
load dot
dot=dot+1
Store dot

Thread 2
↓
load dot
dot=dot+1

Thread 3
↓
load dot
dot=dot+1
Store dot

Thread 2
↓
Store dot
load dot
dot=dot+1
Store dot

Condição de corrida:



exemplo_8/exemplo_08_sem_corrida.c

```
[professor@sдумont12 exemplo_8]$ cat exemplo_08_sem_corrida.c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

////////////////////////////////////
//          PRODUTO ESCALAR
////////////////////////////////////

//Funcao responsavel pelo preenchimento dos vetores
void InicializaVetores(int num,int *V1,int *V2)
{
    int i;
    for(i=0;i<num;i++)
    {
        V1[i] = V2[i] = 1;
    }
}

int main (int argc, char * argv[])
{
    int *V1,*V2;
    int i,j,id;
    int num_threads;
    int par,num,dot;

    printf("\n\n=====\\n");
    printf("\tPRODUTO ESCALAR\\n");
    printf("=====\\n\\n");

    //Dimensao dos Vetores

    num = atoi(argv[1]);

    //Alocacao dinamica dos vetores
    V1=(int*)malloc(num*sizeof(int));
    V2=(int*)malloc(num*sizeof(int));

    //Inicializacao dos vetores
    InicializaVetores(num,V1,V2);

    //Inicilizao da variavel dot
    dot = 0;
    int aux_dot=0;
    //Calculo do produto escalar
    #pragma omp parallel
    {
        #pragma omp for
        for(i=0;i<num;i++)
            #pragma omp atomic
            dot += V1[i]*V2[i];
    }
    printf(" \\n\\n End Of Execution ::: dot = %d\\n\\n\\n",dot);
    return 0;
}
```


exemplo_08

exemplo_08.c (reduction)

exemplo_08_corrida.c

exemplo_08_sem_corrida.c (atomic)

exemplo_08_sem_corrida_critical.c (critical)

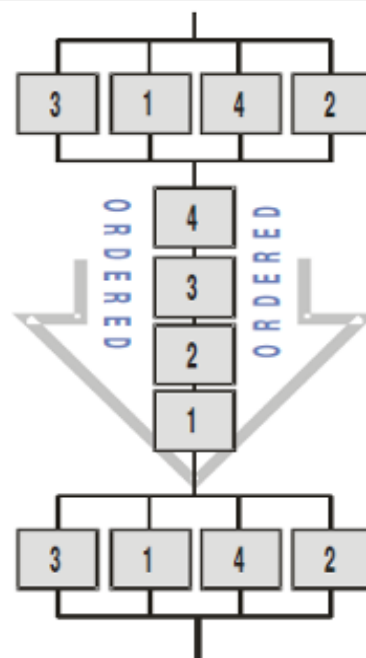
Sincronizador: Ordered

exemplo_04_ordered.c

```
#pragma omp ordered
```

- Garante que o loop será executado seqüencialmente
- Utilizado quando houver uma dependência dos dados atuais com as iterações anteriores.

```
#pragma omp parallel for ordered  
for(i = 1; i<= 4; i++)
```



UTILIZE VARIÁVEIS “PRIVATE”

- **exemplo_04_05.c** (“=var e id são globais)
- **exemplo_04_private** (var e id “private”)
- **exemplo_04_1_private** (var, id, m globais)
- **exemplo_04_firstprivate** (inicializa m com valor antes da região paralela)
- **exemplo_04_lastprivate** (“i” mantém último valor dentro da região paralela)

O que acontece se retirar o lastprivate?

Diretiva Threadprivate

- exemplo_09.c

```
#pragma omp threadprivate (lista de variáveis)
```

- Especifica quais variáveis serão privadas em todo o escopo do código.
- As variáveis serão privadas em todas as regiões paralelas.

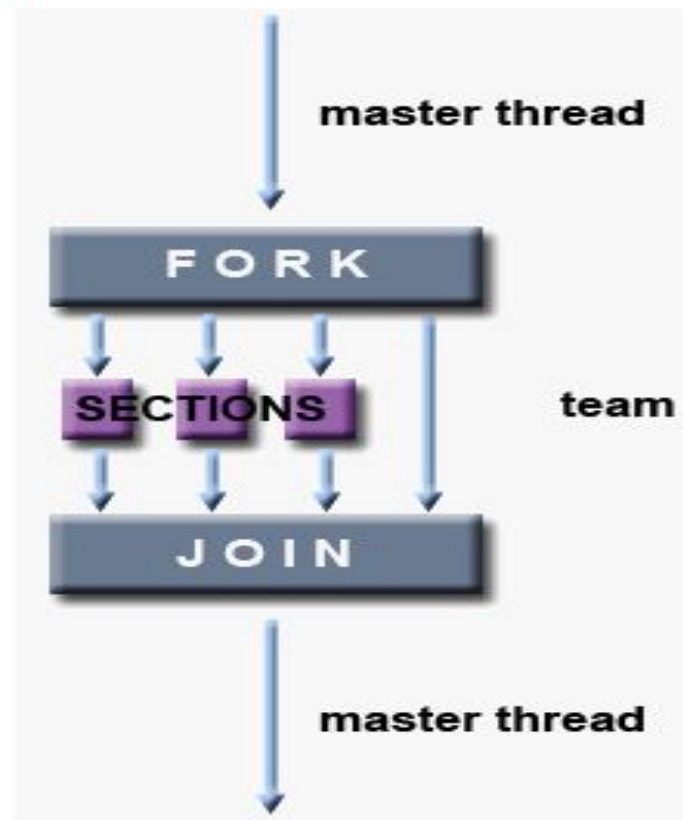
Diretiva Threadprivate : cláusula copyin

exemplo_09_copyin.c : permite que o valor da variável local (var) da thread master possa ser copiado para as variáveis privadas (var) das outras threads.

exemplo_10.c : permite que o valor da variável local (var) de uma thread seja transferida para as variáveis (var) das outras threads.

Construtor de Trabalho Sections

SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".



#pragma omp section

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        soma_vetores(a,b,c,n);
        #pragma omp section
        subtrai_vetores(a,b,d,n);
    }
}
```

- O construtor **section** define o bloco que será executado por uma *thread* dentro do construtor **sections**

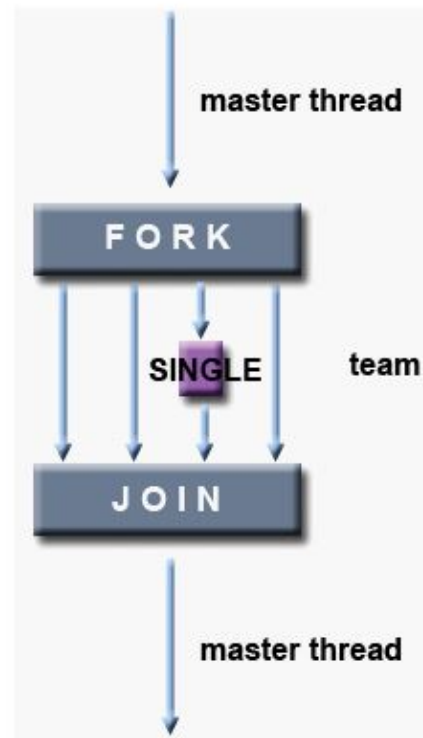
Estrutura do Sections

```
#pragma omp sections [clause ...] newline
    private (list)
    firstprivate (list)
    lastprivate (list)
    reduction (operator: list)
    nowait
{
    #pragma omp section    newline
        structured_block

    #pragma omp section    newline
        structured_block
}
```


Construtor de Trabalho Single

SINGLE - serializes a section of code



#pragma omp single [cláusula,...]

- Esse construtor indica que o bloco sintático localizado logo abaixo do mesmo deve ser executado por apenas uma *thread*.
- Não necessariamente a *thread master*

```
#pragma omp parallel
{
    #pragma omp single
    a = function(t);
    ...
}
```

CLÁUSULAS

- private
- firstprivate
- copyprivate
- nowait

-exemplo_06.c :

divisor de trabalho sections

-exemplo_06_1.c :

divisor de trabalho “single”, testar para:
export OMP_NUM_THREADS=3

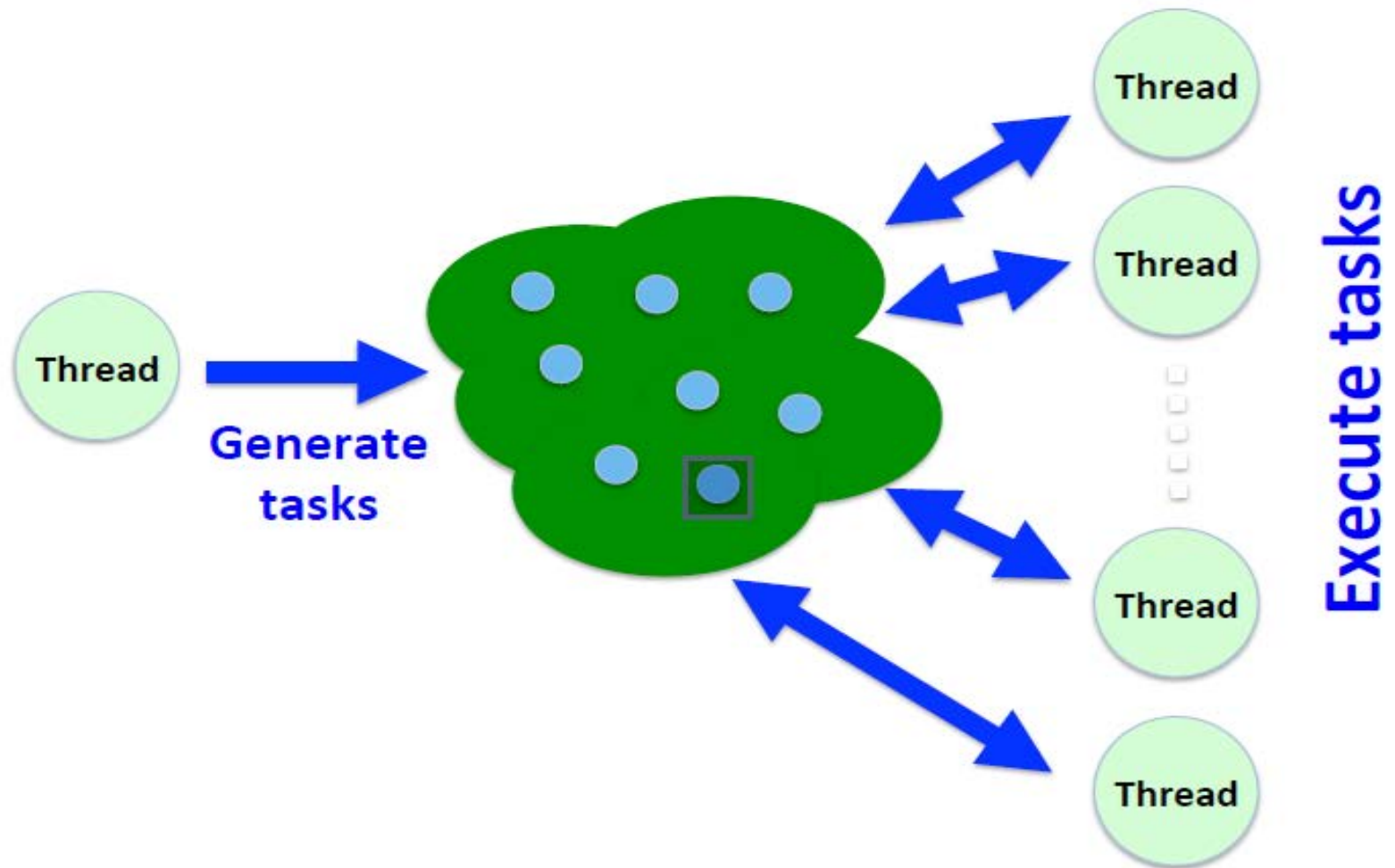
(o terceiro thread tem que esperar o
sections acabar e pode ficar ocioso)

```
#pragma omp master
```

- Define um bloco de código que será executado apenas pela *thread* mestre.
- Não possui barreira implícita, na entrada e na saída.

exemplo_5_nowait_master.c

Construtor de Trabalho TASK



Construtor de Trabalho TASK

C/C++

```
#pragma omp task [clause ...] newline  
    if (scalar expression)  
    final (scalar expression)  
    untied  
    default (shared | none)  
    mergeable  
    private (list)  
    firstprivate (list)  
    shared (list)
```

structured_block

- Quando uma thread encontra um construtor de task, uma nova task é gerada.
- A execução da task cabe ao sistema de runtime
- O término de uma task pode ser forçado através de um “task synchronization”

Construtor de Trabalho TASK

```
#pragma omp task
```

```
!$omp task
```

Defines a task

#pragma omp barrier

```
#pragma omp barrier
```

```
!$omp barrier
```

#pragma omp taskwait

```
#pragma omp taskwait
```

```
!$omp taskwait
```


exemplos/racecar/racecar.c

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

```
int main(int argc, char *argv[]) {
```

```
    printf("A ");
    printf("race ");
    printf("car ");
```

```
    printf("\n");
    return(0);
```

```
}
```

```
$ cc -fast hello.c
$ ./a.out
A race car
$
```

What will this program print ?

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

int main(int argc, char *argv[]) {

    #pragma omp parallel
    {
        printf("A ");
        printf("race ");
        printf("car ");

    } // End of parallel region

    printf("\n");
    return(0);
}
```

***What will this program print
using 2 threads ?***

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car A race car
```

Note that this program could (for example) also print

“A A race race car car” or

“A race A car race car”, or

“A race A race car car”, or

.....

But I have not observed this (yet)

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

```
int main(int argc
```

***What will this program print
using 2 threads ?***

```
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            printf("race ");
            printf("car ");
        }
    } // End of parallel region

    printf("\n");
    return(0);
}
```



```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
```

```

int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {printf("race ");}
            #pragma omp task
            {printf("car ");}
        }
    } // End of parallel region

    printf("\n");
    return(0);
}

```

***What will this program print
using 2 threads ?***

```

$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
$ ./a.out
A race car
$ ./a.out
A car race
$

```

```

int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {printf("race ");}
            #pragma omp task
            {printf("car ");}
            printf("is fun to watch ");
        }
    } // End of parallel region

    printf("\n");
    return(0);
}

```

***What will this program print
using 2 threads ?***

```

$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch car race
$

```

```

int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {printf("race ");}
            #pragma omp task
            {printf("car ");}
            printf("is fun to watch ");
        }
    } // End of parallel region

    printf("\n");
    return(0);
}

```

***What will this program print
using 2 threads ?***

```

$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch car race
$

```

```

int main(int argc, char
#pragma omp parallel
{
    #pragma omp single
    {
        printf("A ");
        #pragma omp task
        {printf("car ");}
        #pragma omp task
        {printf("race ");}
        #pragma omp taskwait
        printf("is fun to watch ");
    }
} // End of parallel region

printf("\n");return(0);
}

```

***What will this program
print using 2 threads ?***



```

$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
$
A car race is fun to watch
$ ./a.out
A car race is fun to watch
$ ./a.out
A race car is fun to watch
$

```



```
my_pointer = listhead;
```

```
#pragma omp parallel  
{
```

```
    #pragma omp single  
    {
```

```
        while(my_pointer)
```

```
            #pragma omp task firstprivate(my_pointer)  
            {
```

```
                (void) do_independent_work (my_pointer);
```

```
            }
```

```
            my_pointer = my_pointer->next ;
```

```
        }
```

```
    } // End of single
```

```
} // End of parallel region
```

*OpenMP Task is specified here
(executed in parallel)*

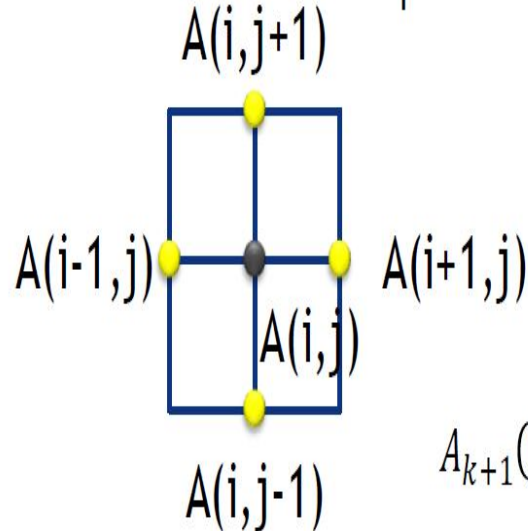


Example: Jacobi Iteration

Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.

Common, useful algorithm

Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$



$$A_{k+1}(i, j) = \frac{A_k(i-1, j) + A_k(i+1, j) + A_k(i, j-1) + A_k(i, j+1)}{4}$$

Exemplo: jacobi/laplace2d.c:

```
while ( error > tol && iter < iter_max )
{
    error = 0.0;

#pragma omp parallel for shared(Anew, A)
    for( int j = 1; j < n-1; j++)
    {
        for( int i = 1; i < m-1; i++ )
        {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

#pragma omp parallel for shared(Anew, A)
    for( int j = 1; j < n-1; j++)
    {
        for( int i = 1; i < m-1; i++ )
        {
            A[j][i] = Anew[j][i];
        }
    }

    if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

    iter++;
}

double runtime = GetTimer();

printf(" total: %f s\n", runtime / 1000);
```

Exemplo laplace2d_task.c

```
        error = 0.0;
#pragma omp parallel
{
    #pragma omp single
    {
        while ( error > tol && iter < iter_max )
        {

            for( int j = 1; j < n-1; j++)
            {
                for( int i = 1; i < m-1; i++ )
                {
                    #pragma omp task firstprivate (error)
                    Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                           + A[j-1][i] + A[j+1][i]);
                    error = fmax( error, fabs(Anew[j][i] - A[j][i]));
                }
            }
        }
    }
}
```

Diretivas de Sincronização:

- As diretivas de sincronização garantem que o acesso ou atualização de uma variável compartilhada ocorra no momento certo.

```
#pragma omp critical  
#pragma omp atomic  
#pragma omp barrier  
#pragma omp flush  
#pragma omp ordered  
#pragma omp threadprivate  
#pragma omp master
```

Sincronizador Critical: exemplo_07.c

```
#pragma omp critical
```

- Restringe a execução de uma determinada tarefa a uma *thread* (*threads* do mesmo grupo) por vez.
- Atualização no sistema operacional

Obs: Testar exemplo_07_corrida.c

sincronizador atomic

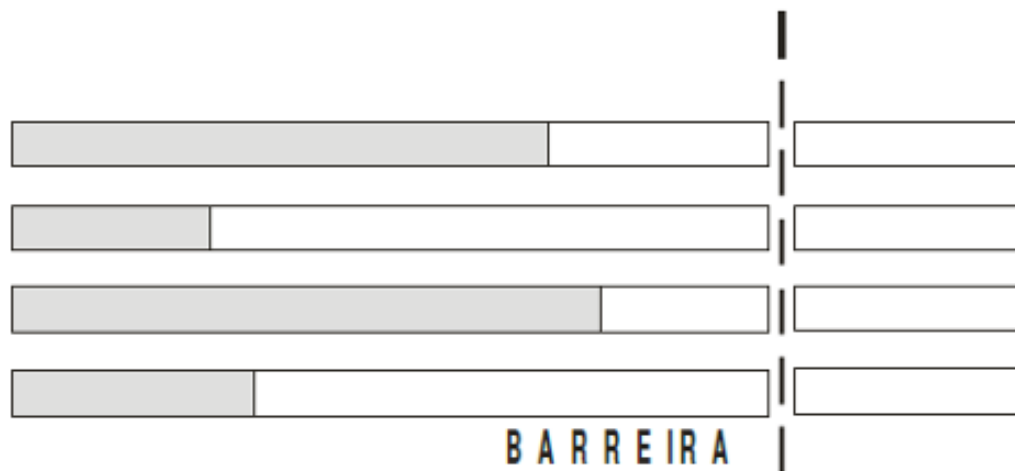
```
#pragma omp atomic
```

- Impedem que várias *threads* acessem essa variável ao mesmo tempo.
- Esse bloqueio é aplicado a todas as *threads* que executam o programa, não apenas as *threads* do mesmo grupo.
- Atualização a nível do processador

Sincronizador barrier:

```
#pragma omp barrier
```

- É utilizado para sincronizar todas as *threads* em determinado ponto do código.
- Em alguns construtores existe uma barreira implícita: “Na saída” `parallel`, `for`, `sections`, `single`.



Cláusula nowait

`nowait`

- Essa cláusula faz com que as *threads* ignorem as barreiras implícitas.

> **Que diretivas podem utilizar essa cláusula ?**

```
#pragma omp single  
#pragma omp for  
#pragma omp sections  
#pragma omp parallel for  
#pragma omp parallel sections
```

exemplo_5

`exemplo_05_nowait.c`

`exemplo_05_sem_nowait.c` : a thread single
será sempre a última a executar

Operações Implícitas de Flush

- entrada de uma barreira
- entrada/saída do loop parallel
- entrada/saída de um divisor de trabalho
- entrada/saída de uma região crítica

Sincronizador flush

exemplo_fush.c : exemplo de algoritmo produtor/consumidor

Obs:

section 2 espera a execução da section 1

section 3 é independente

section 4 espera a execução da section 2

Funções de Bloqueio

- Rotinas utilizadas para promover a sincronização entre *threads* através de uma implementação de mais “baixo-nível”.
- Restringem a execução de um determinado trecho de código a apenas uma thread por vez.
- Permitem maior flexibilidade de sincronização que o uso das diretivas `critical` e `atomic`.
- Operam sobre variáveis “lock” do tipo: `omp_lock_t` e `omp_nest_lock_t`

`omp_init_lock()` e `omp_init_nested_lock()`

- Inicializa uma “lock”

`omp_destroy_lock()` e `omp_destroy_nested_lock()`

- Altera o estado da “lock” para não-inicializado

`omp_set_lock()` e `omp_set_nested_lock()`

- Bloqueiam a thread até que a “lock” especificada esteja disponível (desbloqueada) e, então, bloqueiam essa variável

`omp_unset_lock()` e `omp_unset_nested_lock()`

- Desbloqueiam uma “lock” simples e decrementam o contador *nesting* de uma variável aninhada

`omp_test_lock()` e `omp_test_nested_lock()`

- Bloqueiam a “lock” quando a mesma está disponível sem bloquear a thread que executa a rotina

Procedimento para se utilizar as funções de Bloqueio:

- > Definir a variável “*lock*” (simples ou aninhada);
- > Inicializar a “*lock*” através da função `omp_init_lock()`;
- > Bloquear a “*lock*” através das funções `omp_set_lock()` e `omp_test_lock()`;
- > Desbloquear a “*lock*” através da função `omp_unset_lock()`;
- > Destruir a “*lock*” através da função `omp_destroy_lock()`.

exemplo_12.c

Funções de tempo:

- exemplo_13.c

```
double omp_get_wtime(void)
```

- Retorna o valor decorrido em segundos relativo a um tempo passado.

```
double start, end;  
start = omp_get_wtime();  
...  
end = omp_get_wtime();  
  
printf(Tempo decorrido%lf, end-start);
```


Desempenho

> *speed-up*

Representa o ganho de velocidade de processamento de uma aplicação quando executada com n processadores.

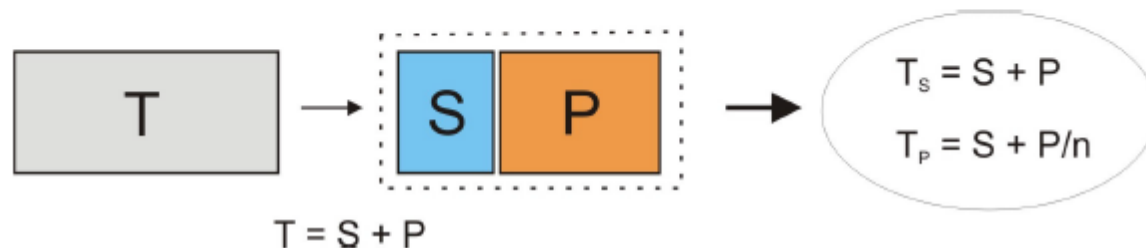
$$S_P = \frac{T_S}{T_P}$$

T_S - Tempo Serial

T_P - Tempo Paralelo

Lei de AMDHAL

- O fator *speed-up* é limitado por um valor máximo, decorrente da parcela serial do código, isso é conhecido com Lei de Amdahl.



- Combinando as expressões para tempo serial e tempo paralelo com a expressão do *speed-up*, têm-se:

$$S_P = \frac{1}{S + (1 - S)/n} \quad \text{onde } \lim_{n \rightarrow \infty} S_P = \frac{1}{S}$$

S – Fração serial do código
 n – Número de processadores

Ilustrando a Lei de Amdhal

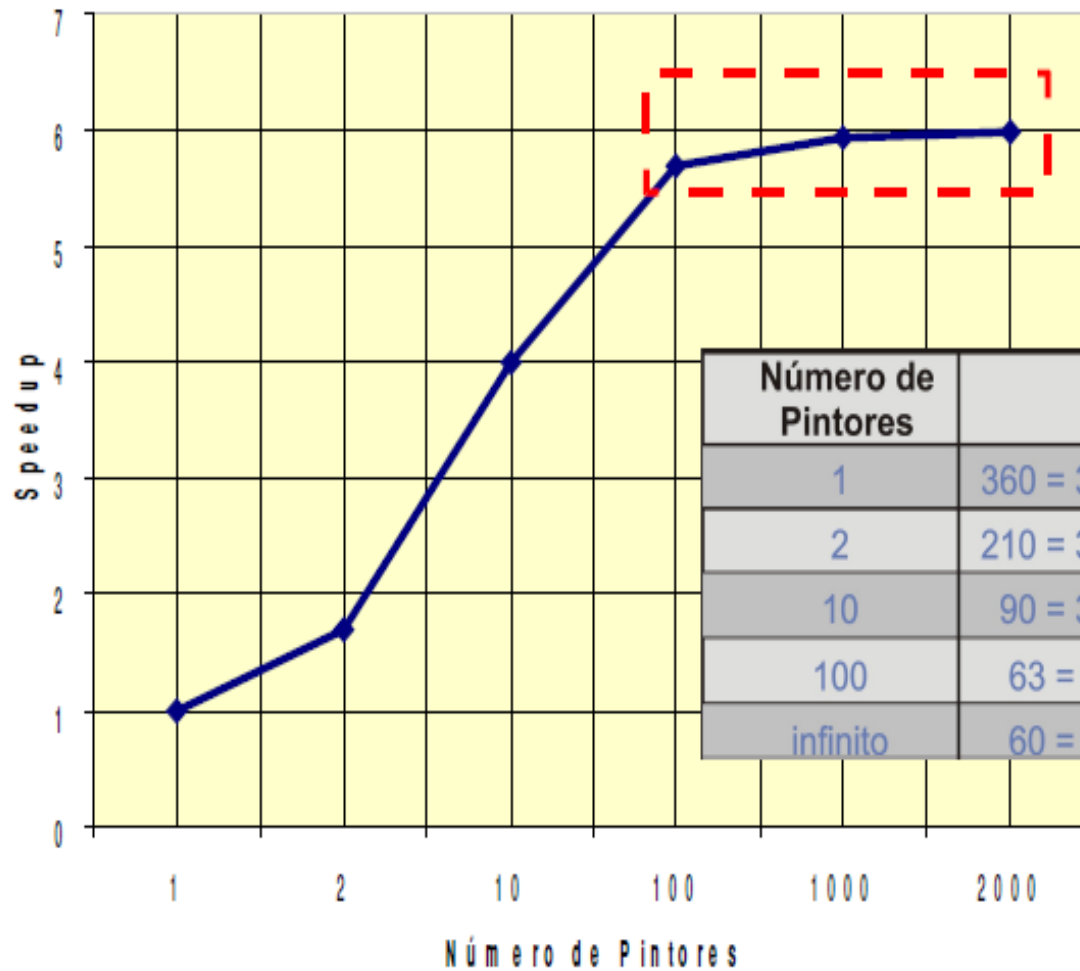
- Para pintar uma cerca



Preparo da tinta = 30 seg
Pintura das estacas = 5 min = 300 seg
Tempo para a tinta secar = 30 seg

Número de Pintores	Tempo	Speedup
1	$360 = 30 + 300 + 30$	1,0 x
2	$210 = 30 + 150 + 30$	1,7 x
10	$90 = 30 + 30 + 30$	4,0 x
100	$63 = 30 + 3 + 30$	5,7 x
infinito	$60 = 30 + 0 + 30$	6,0 x

Número de Pintores X Speedup



Número de Pintores	Tempo	Speedup
1	$360 = 30 + 300 + 30$	1,0 x
2	$210 = 30 + 150 + 30$	1,7 x
10	$90 = 30 + 30 + 30$	4,0 x
100	$63 = 30 + 3 + 30$	5,7 x
infinito	$60 = 30 + 0 + 30$	6,0 x