Introdução a OpenMP
MC-SD02-I
OpenMP Programação Avançada
MC-SD02-II

Carla Osthoff
LNCC/MCTI
osthoff@lncc.br

Material do curso:

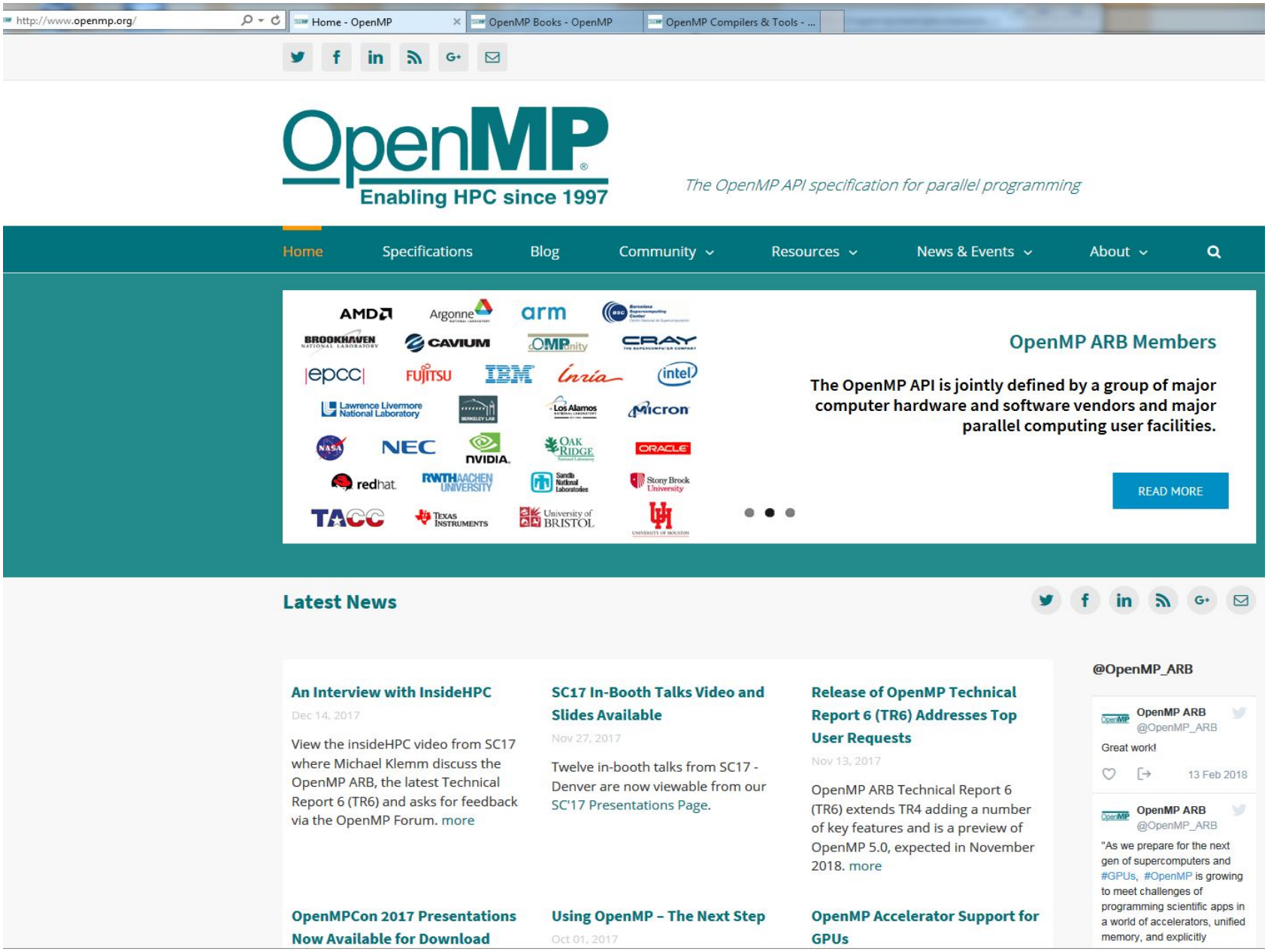
www.cenapad-rj.lncc.br/tutoriais/materiais-hpc/semana-sdumont

Link alternativo para o material:

https://www.dropbox.com/s/wnd0xl5s7w6zbcv/OpenMP2021.pdf?dl=0

https://www.dropbox.com/s/9iohyh3aftyszpz/OpenMP_2021_exemplos.tar?dl=0

www.openmp.org



OpenMP ARB releases OpenMP 5.1 with vital usability enhancements

While the primary focus has been enhancements, clarifications and corrections to the 5.0 specification, several useful new features have been added such as support for interoperability with lower level APIs like CUDA and HIP.

OpenMP @ SC20

We're gearing up for SC'20 and delighted to have three OpenMP tutorials included in the program: Advanced OpenMP: Host Performance and 5.0 Features, Programming your GPU with OpenMP: A hands-on Introduction, and The OpenMP Common Core: A hands-on Introduction.

ECP OpenMP Hackathons 2020

ECP SOLLVE, in conjunction with ORNL and NERSC, are organizing two OpenMP Hackathons in July and August. We encourage participation of teams interested in using OpenMP to port and optimize their applications on GPUs, and in using OpenMP for energy-efficient processor architectures.

OpenMP®

The OpenMP API specification for parallel programming

OpenMP Books

- OpenMP Common Core: Making OpenMP Simple Again – by Tim Mattson, Helen He, Alice Koniges (2019)
- Using OpenMP – The Next Step – by Ruud van der Pas, Eric Stotzer and Christian Terboven (2017)
- Using OpenMP – Portable Shared Memory Parallel Programming – by Chapman, Jost, and Van Der Pas (2007)
- Parallel Programming in OpenMP – by Rohit Chandra et al.
- Parallel Programming in C with MPI and OpenMP – by Michael J. Quinn.
- Parallel Programming Patterns: Working with Concurrency in OpenMP, MPI, Java, and OpenCL – by Timothy G Sanders
- An Introduction to Parallel Programming with OpenMP, PThreads and MPI – by Robert Cook
- The International Journal of Parallel Programming – Issues and articles devoted to OpenMP.

OpenMP Compilers & Tools

GNU	GCC C/C++/Fortran	The free and open-source GNU Compiler Collection (GCC) supports among others Linux, Solaris, AIX, MacOSX, Windows, FreeBSD, NetBSD, OpenBSD, DragonFly BSD, HPUX, RTEMS, for architectures such as x86_64, PowerPC, ARM, and many more. Code offloading to NVIDIA GPUs (nvptx) and the AMD Radeon (GCN) GPUs Fiji and Vega is supported on Linux. OpenMP 4.0 is fully supported for C, C++ and Fortran since GCC 4.9; OpenMP 4.5 is fully supported for C and C++ since GCC 6 and partially for Fortran since GCC 7. OpenMP 5.0 is partially supported for C and C++ since GCC 9 and extended in GCC 10. The next release, GCC 11, will fully support OpenMP 4.5 for Fortran and extend the OpenMP 5.0 support for C, C++ and Fortran; the devel/omp/gcc-10 (og10) branch augments the GCC 10 branch with OpenMP and offloading features, mostly from GCC 11 development branch. Compile with -fopenmp to enable OpenMP. GCC binary builds are provided by Linux distributions, often with offloading support provided by additional packages, and by multiple entities for other platforms – and you can build it from source. Releases and release notes: https://gcc.gnu.org/ OpenMP documentation: https://gcc.gnu.org/onlinedocs/libgomp/ Building and using GCC for offloading: https://gcc.gnu.org/wiki/Offloading

OpenMP Compilers & Tools

Intel	C/C++/Fortran	Windows, Linux, and MacOSX.

- OpenMP 3.1 C/C++/Fortran fully supported in version 12.0, 13.0, 14.0 compilers
- OpenMP 4.0 C/C++/Fortran supported in version 15.0 and 16.0 compilers
- OpenMP 4.5 C/C++/Fortran supported in version 17.0, 18.0, and 19.0 compilers
- OpenMP 4.5 and subset of OpenMP 5.0 in C/C++/Fortran compiler classic 2021.1
- OpenMP 4.5 and subset of OpenMP 5.1 supported in oneAPI DPC++/C++ compiler 2021.1 under -fiopenmp -fopenmp-targets=spir64
- OpenMP 4.5 and subset of OpenMP 5.1 supported in oneAPI Fortran compiler (Beta) under -fiopenmp -fopenmp-targets=spir64.

Compile with -Qopenmp on Windows, or just -qopenmp or –fiopenmp on Linux or Mac OSX

Compile with -fiopenmp -fopenmp-targets=spir64 on Windows and Linux for offloading support

OpenMP®

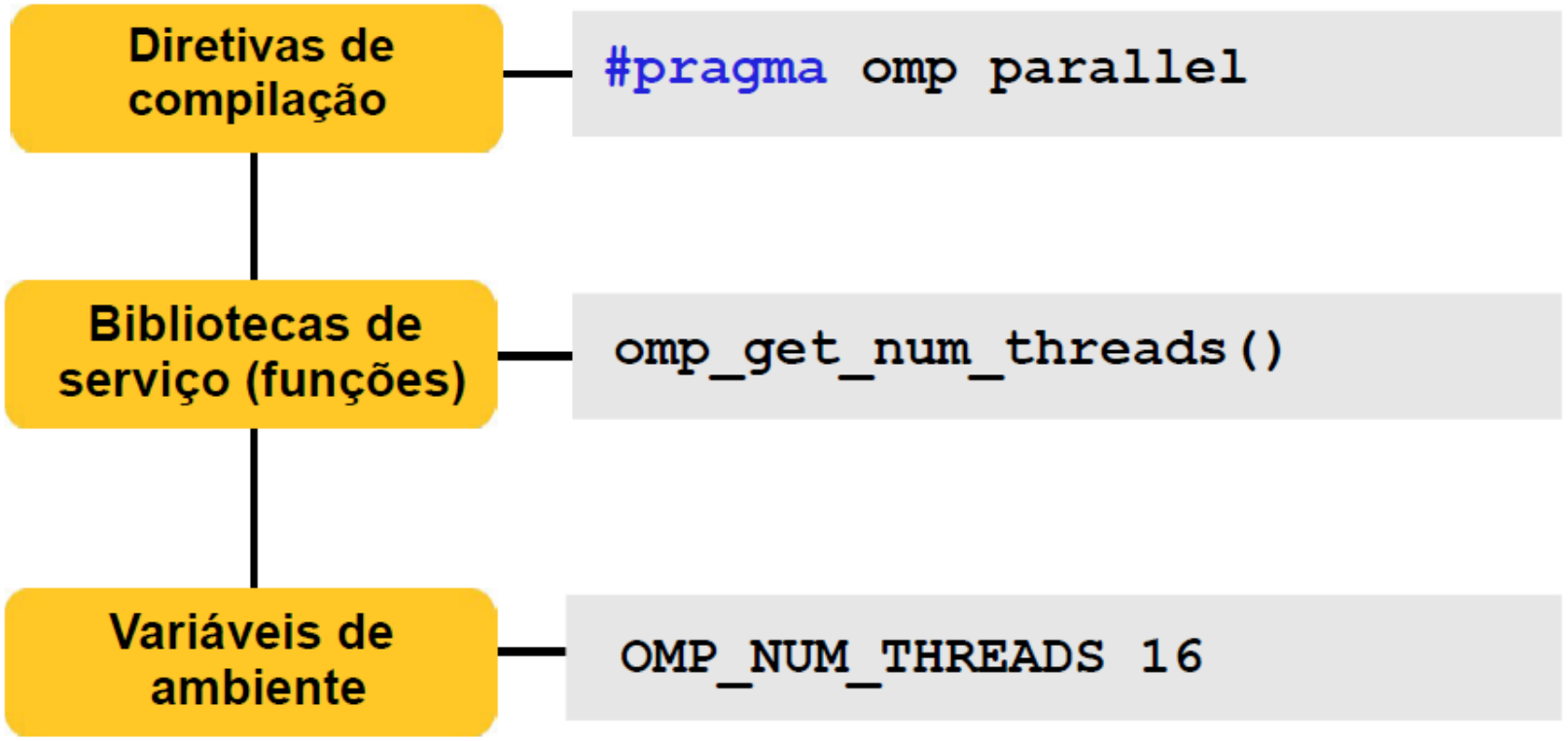
The OpenMP API specification for parallel programming

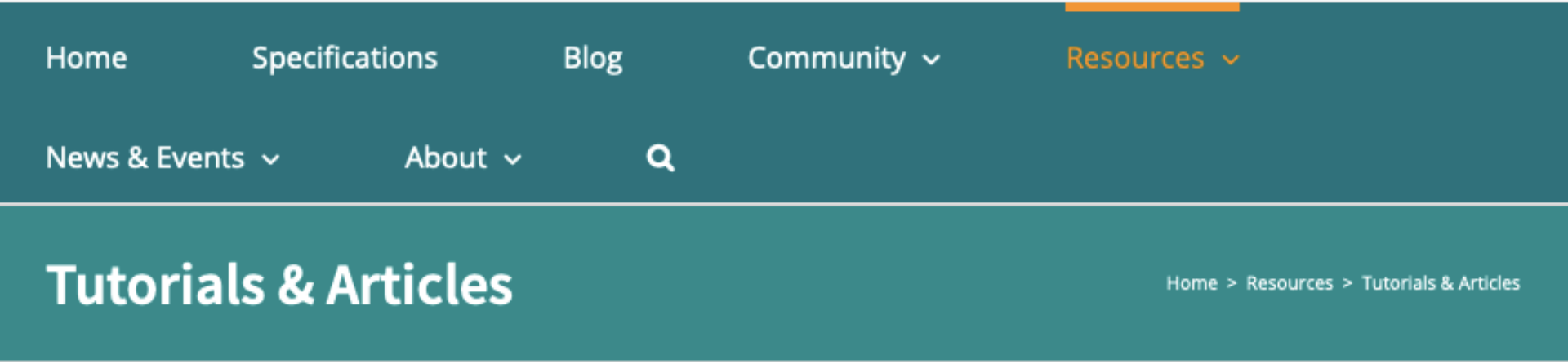
Home | Specifications | Blog | Community | Resources | News & Events

OpenMP Is:

- An Application Program Interface (API) that may be used to explicitly direct ***multi-threaded, shared memory*** parallelism.
- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- An abbreviation for: **Open Multi-Processing**

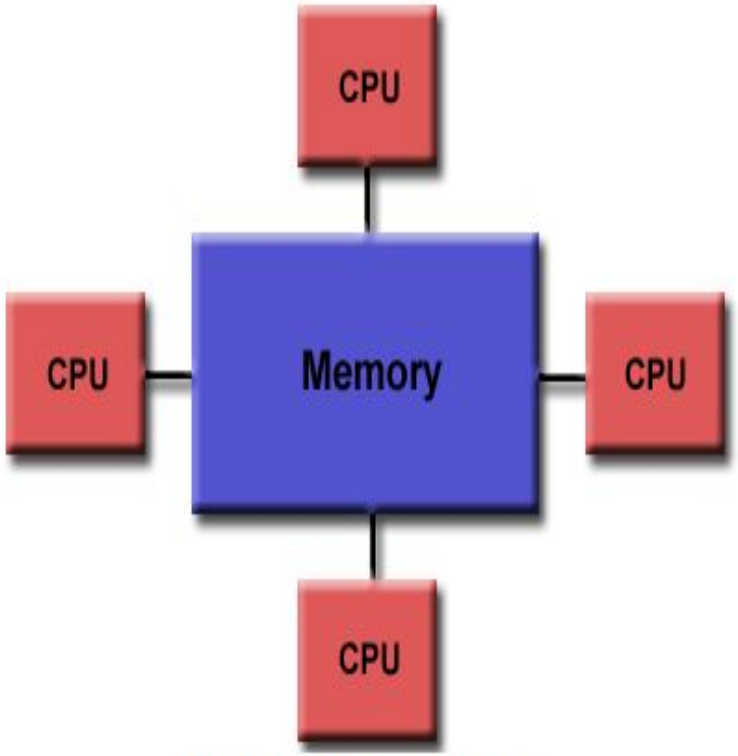
Estrutura do OpenMP API



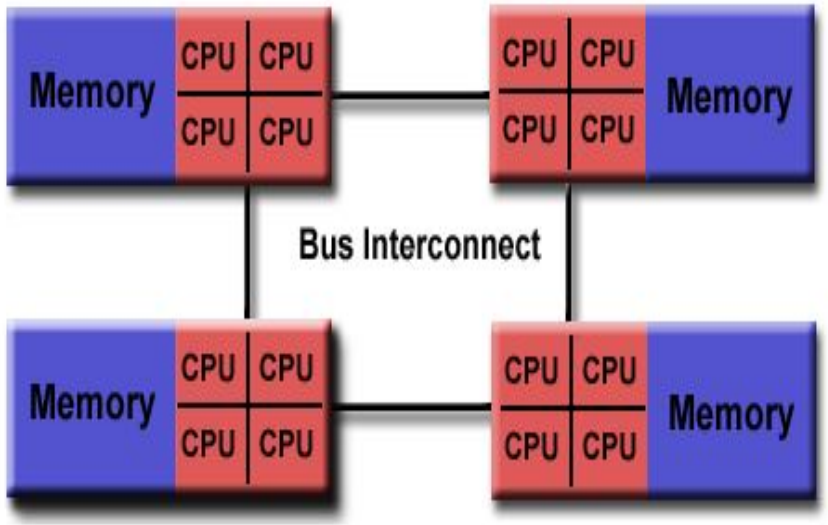


Tutorial:https://computing.llnl.gov/tutorials/openMP/

Modelo de Programação do OpenMP : Modelo de Memória Compartilhada

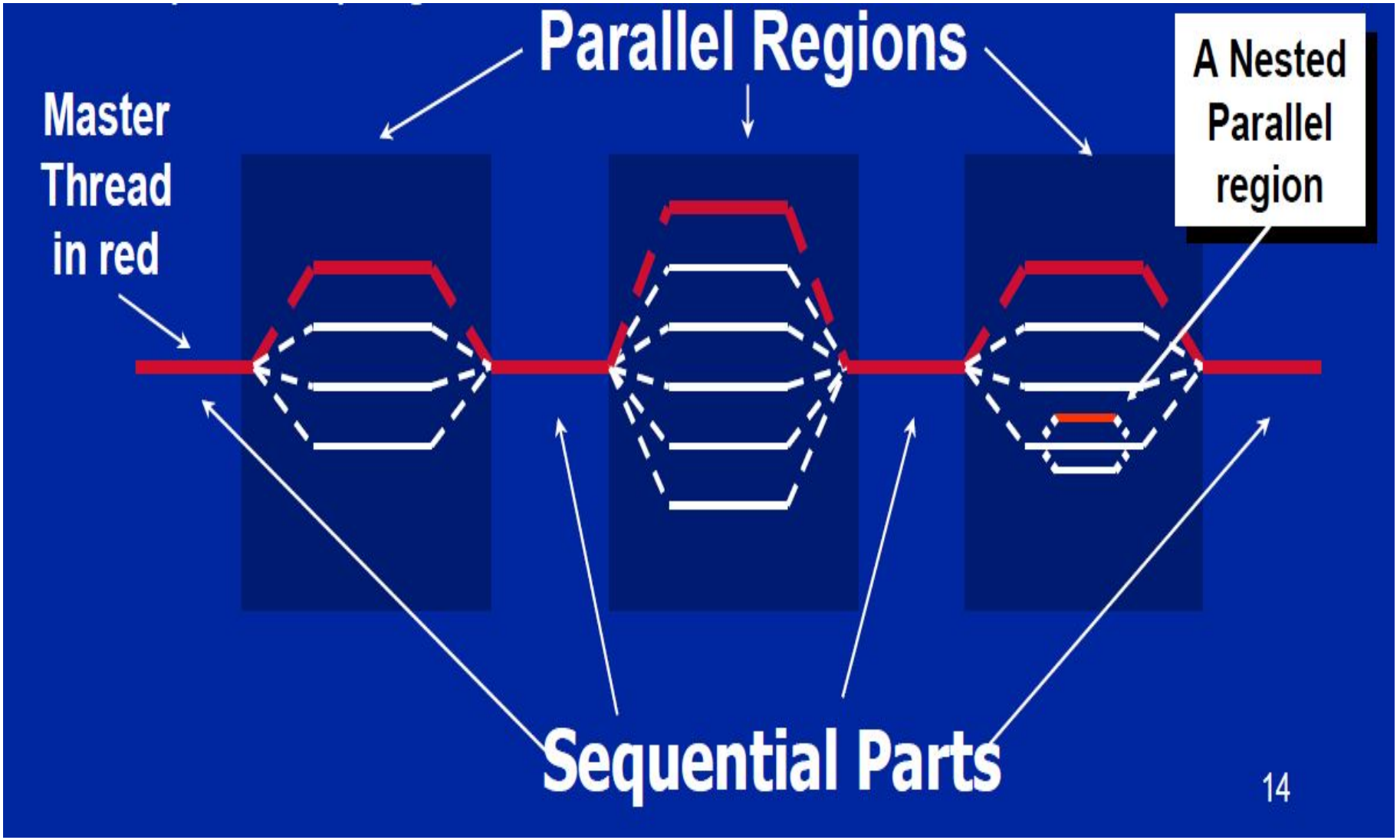


Uniform Memory Access



Non-Uniform Memory Access

OpenMP utiliza
Modelo de programação “Fork-Join”



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
```

Compiler / Platform	Compiler Commands	OpenMP Flag

Intel
Linux
	`icc`
	`icpc`
	`ifort`
		`-qopenmp`

GNU
Linux
IBM Blue Gene
Sierra, CORAL EA
	`gcc`
	`g++`
	`g77`
	`gfortran`
		`-fopenmp`

PGI
Linux
Sierra, CORAL EA
	`pgcc`
	`pgCC`
	`pgf77`
	`pgf90`
		`-mp`

Clang Linux Sierra, CORAL EA	`clang` `clang++`	`-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
Sierra, CORAL EA
	`xlc_r`
	`xlC_r, xlc++_r`
	`xlf_r`
	`xlf90_r`
	`xlf95_r`
	`xlf2003_r`
	`xlf2008_r`
		`-qsmp=omp`

Instalação do material na sua conta to SDUMONT:

$ sftp user@login.sdumont.lncc.br

$ put OpenMP_2021_exemplos.tar

$ exit

$ssh user@login.sdumont.lncc.br

$cp OpenMP_2021_exemplos.tar $SCRATCH/.

$cd $SCRATCH

$tar xvf OpenMP_2021_exemplos.tar

Instalação do material na maquina que você tem acesso:

$ sftp user@intelknl.lncc.br

$ put OpenMP_2021_exemplos.tar

$ exit

$ssh user@intelknl..lncc.br

$tar xvf OpenMP_2021_exemplos.tar

C / C++ - General Code Structure

```
#include <omp.h>

main ()  {

int var1, var2, var3;

Serial code
      .
      .
      .

Beginning of parallel region. Fork a team of threads.
Specify variable scoping

#pragma omp parallel private(var1, var2) shared(var3)
   {

   Parallel region executed by all threads
            .
   Other OpenMP directives
            .
   Run-time Library calls
            .
   All threads join master thread and disband

   }

Resume serial code
      .
      .
      .

}
```

- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel region. Only the master thread continues execution past this point.
- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

I - 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];
}
```

C/C++

```
#pragma omp parallel [clause ...]  newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)

   structured_block
```

Primeiro Exemplo

$ cd exemplo_1

$ gcc -fopenmp exemplo_01.c -o exemplo_01

$ ./exemplo_01

OBS:

O número de threads a serem gerados corresponde ao número de núcleos lógicos do processador . Você pode descobrir o número de núcleos lógicos através do comando:

$ cat /proc/cpuinfo

O número de threads a serem gerados pode ser alterado através coomando;

$ export OMP_NUM_THREADS=2

Execução do exemplo_01 no SDumont

- OpenMP "script" para compilar
  - **$./compilar.sh**
- **OpenMP** "script" para submeter na fila de execução:
  - **$sbatch sub.sh 10 exemplo_01**
    - **$squeue -u $USER (para visualizar o job na fila de execução**
    - **$scancel JOB_ID (para cancelar o job da fila de execução, caso seja necessário)**
    - **$cat slurm-JOB_ID.out**

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

```
#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_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()
```

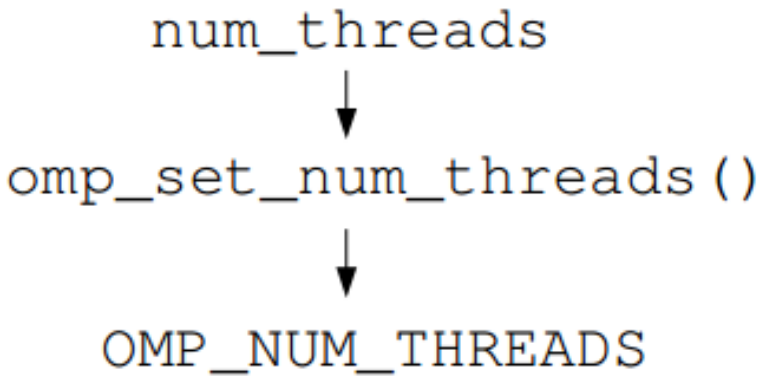
- Retorna o número de *threads* disponíveis para executar a região paralela.

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

- Define o número de *threads* que irá executar as regiões paralelas.

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

- Diagrama de Precedência:



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());
     }
    }
}
```

II- CONSTRUTORES DE TRABALHO

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

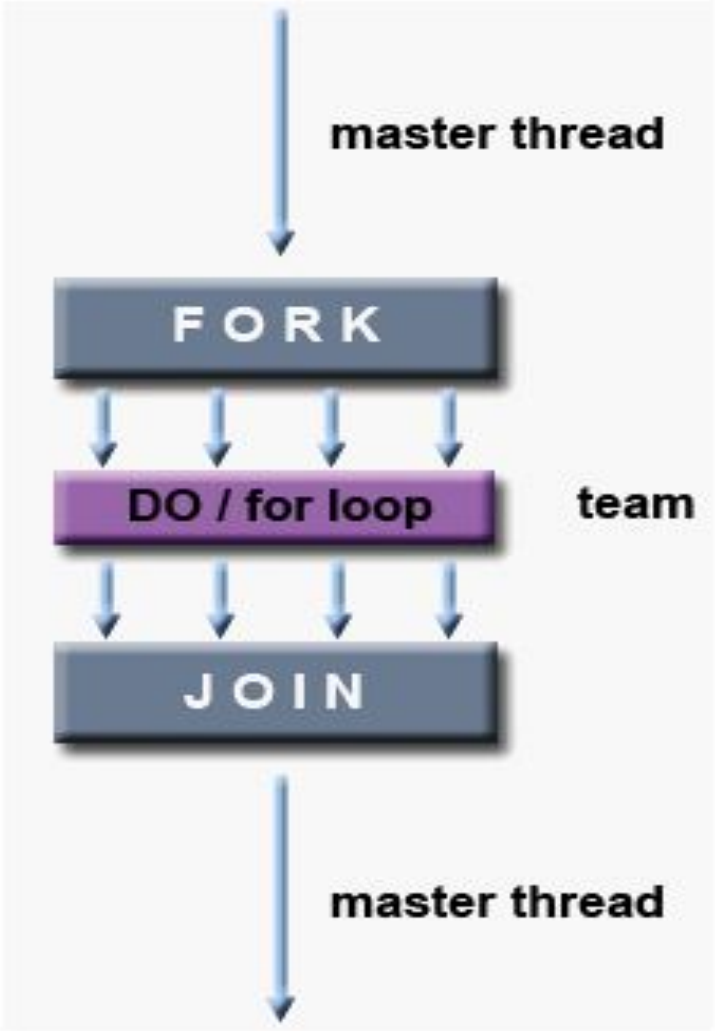
```
#pragma omp for

#pragma omp sections

#pragma omp single
```

II.1- CONSTRUTOR DE TRABALHO FOR

```
#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.

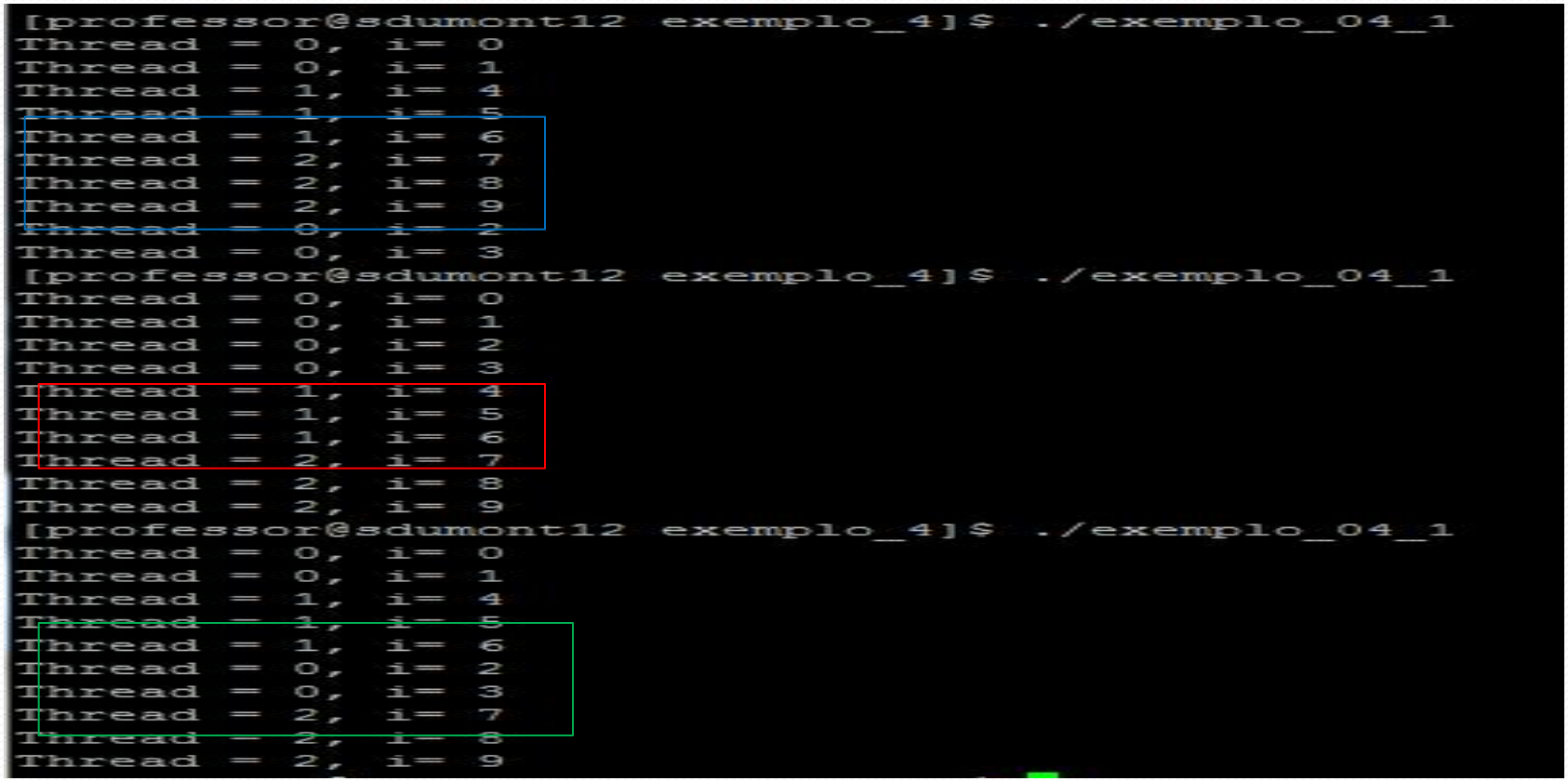
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".



exemplo_04_1.c

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

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

    n=10;
    m=10;
    #pragma omp parallel
    {
       #pragma omp for
        for (i = 0; i < n; i++)
        {
          id= omp_get_thread_num();
        printf("Thread = %d, i= %d , m=%d\n",id,i,m);
        }
    }
```

Observem que a thread 0 sempre pega as primeiras iterações



As cláusulas servem para alterar a execução padrão:

```
#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];
    }
}
```

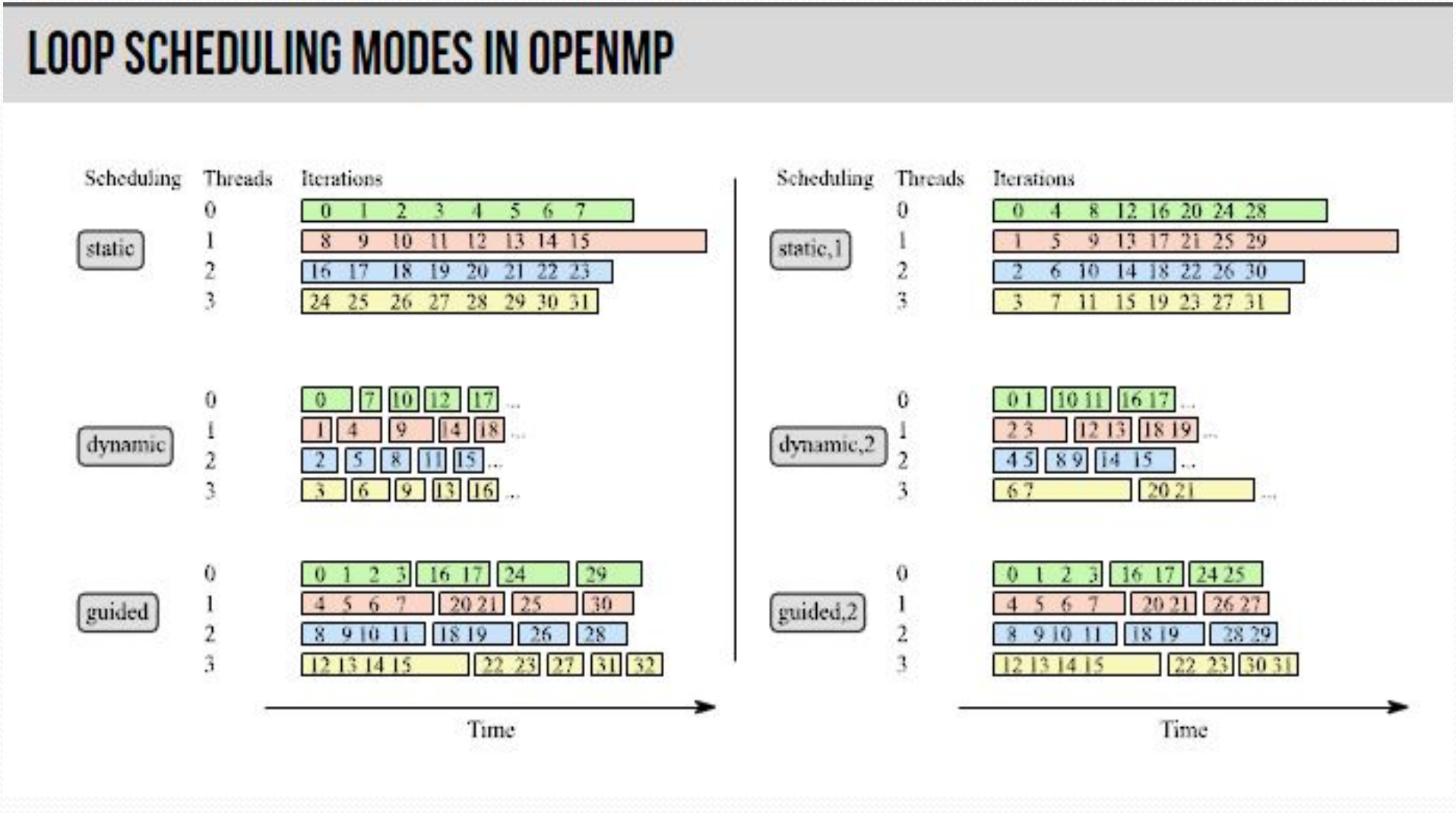
```
if(n>100*var)
  região paralela ativa
else
  região paralela inativa
```

Cláusula schedule do construtor de trabalho FOR

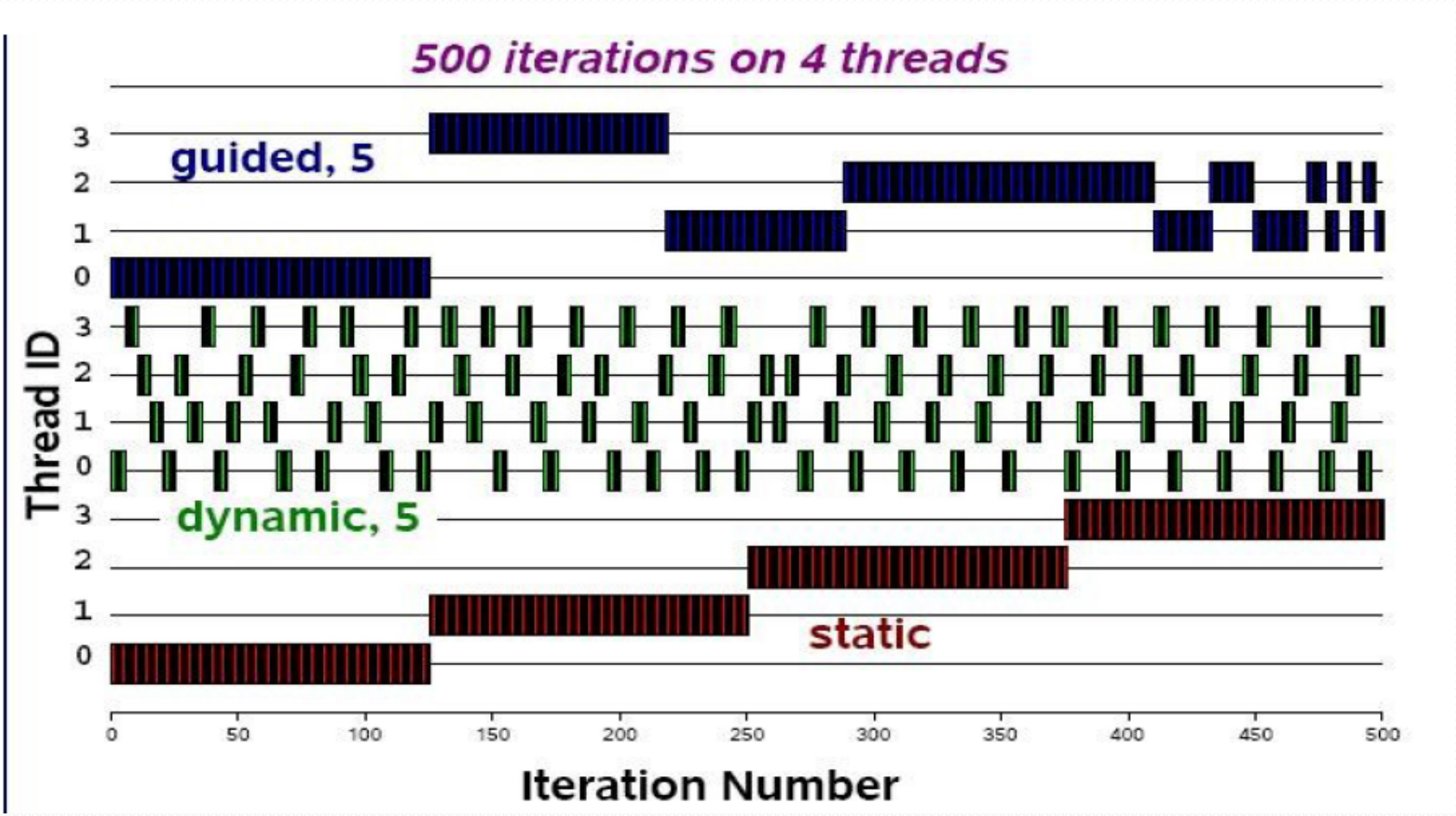
```
schedule(tipo,[tamanho])
```

- É utilizada apenas no construtor for e controla a forma com as iterações são distribuídas entre as *threads*.
- **`schedule`** pode ser do tipo:
  - `static`
  - `dynamic`
  - `guided`
  - `runtime`

static aumenta “desbalanceamento de carga”
dynamic aumenta “gasto com busca”
guided mistura dos dois



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)

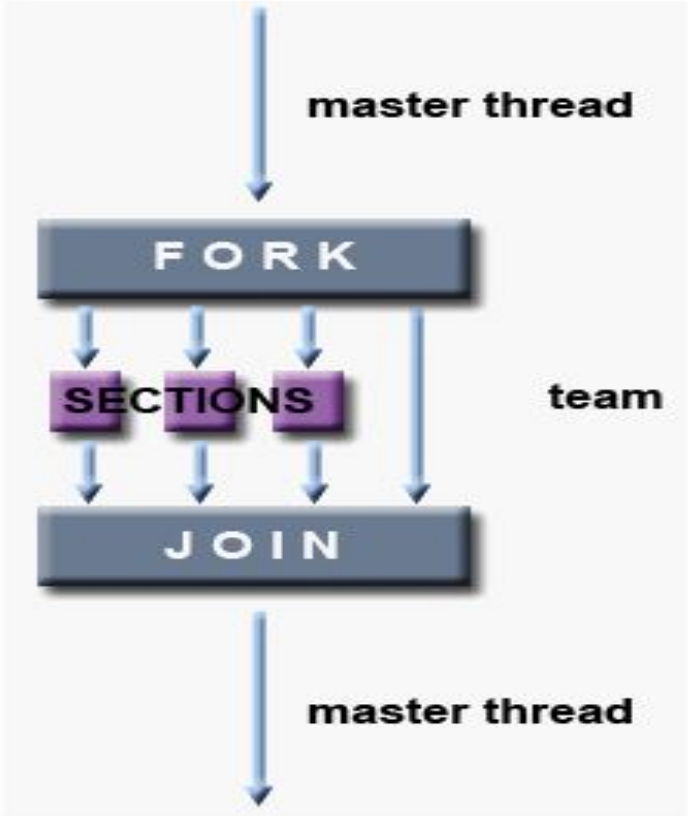
$time ./exemplo_04_3 (runtime: usa variável de ambiente)

$export OMP_NUM_THREADS=x

$export OMP_SCHEDULE="mode, chunk”

II.2- 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`**

Cláusulas para alterar a execução padrão

```
#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

  }
```

-exemplo_06.c :

(3 threads trabalham em paralelo)

-exemplo_06_1.c :

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

(Apenas duas threads recebem trabalho no “sections” e o terceiro thread tem que esperar o trabalho das outras duas acabar observe que neste caso um mesmo thread pode executar uma das duas na sections e depois no single)

Exemplos para Construtor de Trabalho Sections

exemplo_05_corrida.c: (as variáveis "id“ e "i" são globais e são alteradas simultaneamente pelas threads .)

Observem que cada vez que se executa aparece uma sáida diferente. Isto se deve porque a variável id e i estão sendo alteradas simultaneamente pelas threads da section vetor B e section vetor A. Para corrigir, tem que transformar estas variáveis em “private".

exemplo_05.c (cada thread executa uma section e escreve em variáveis privativas)

Observação:

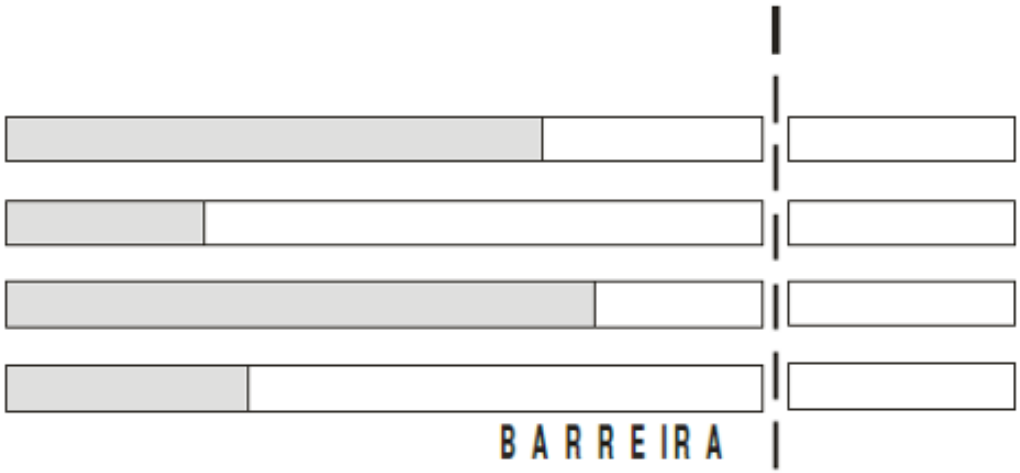
- Apenas os índices do loop dos construtores de trabalho “for” são “private”

- Os índices dos demais construtores de trabalho necessitam ser declarados como “private”

Sincronizador barrrier:

```
#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.`



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
```

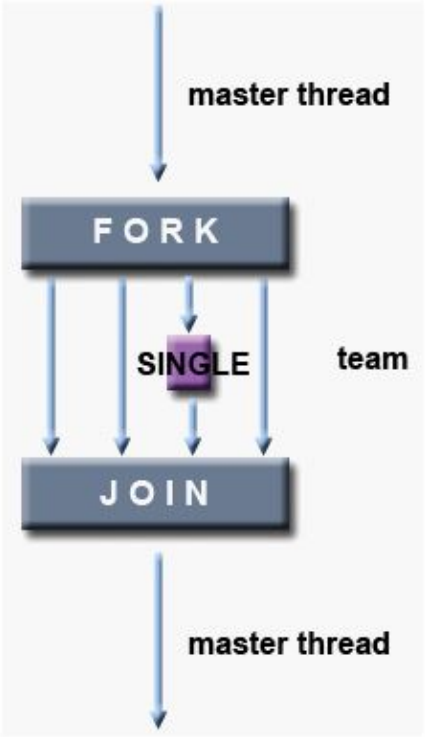
exemplo_5

- Exemplo_05.c: cada section é executada com uma thread distinta

- exemplo_05_nowait.c (a section single pode ser executada por um thread que havia executado outra section e já terminou.)
- exemplo_05_sem_nowait.c : a thread single executará após que todas as threads da section anterior tiverem terminado

II.3- CONSTRUTOR DE TRABALHO SINGLE

SINGLE - serializes a section of code



exemplo_01_set_num_threads.c
exemplo_01_num_threads.c

```
#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

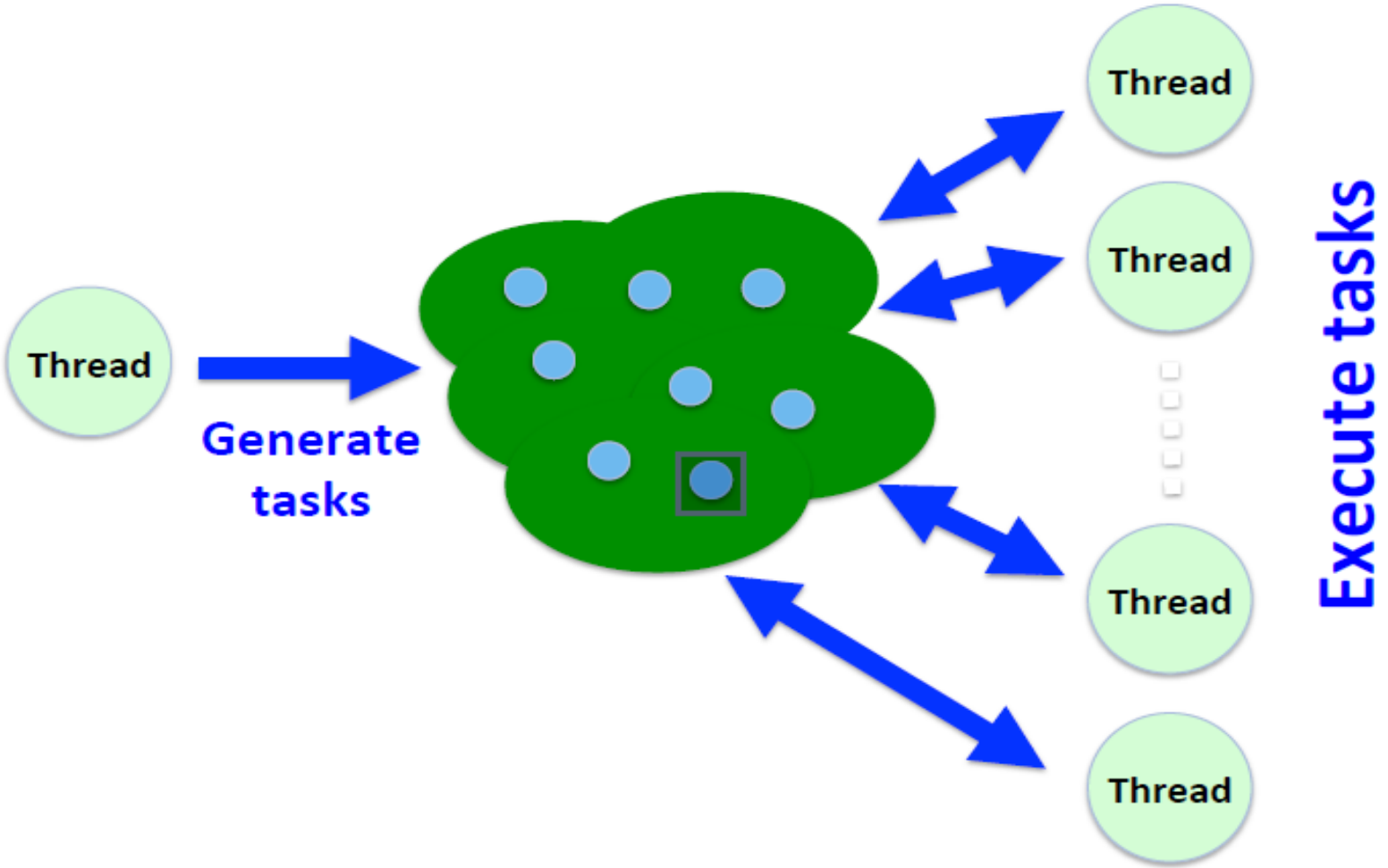
#pragma omp master

- 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

(só será executada pela “thread 0”, inclusive quando as outras estiverem livres

II.4- CONSTRUTOR DE TRABALHO TASK



Diretivas do 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”

Diretivas do 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

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

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

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

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

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

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

    #pragma omp parallel
    {
      #pragma omp single
       {
            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
```

```
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, cha
   #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
```

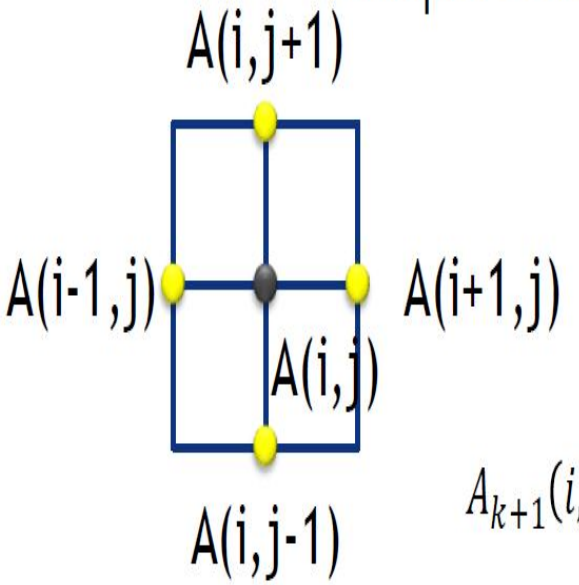
OpenMP Task is specified here (executed in parallel)

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
Apresenta uma paralelização com o construtor "FOR"

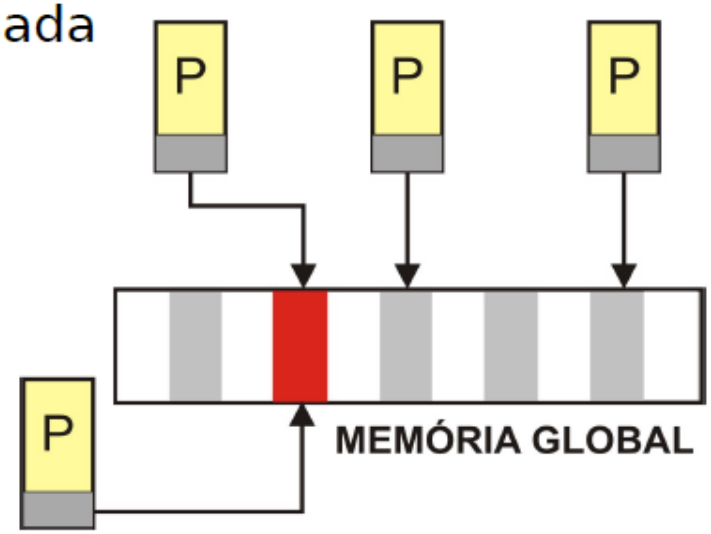
Exemplo laplace2d_task.c
Apresenta uma paralelização com o construtor "task"

Podemos observar que o construtor “task" introduz mais gastos de execução e deve ser implementado para tasks com “granularidade grossa”

III- CONDIÇÃO DE CORRIDA

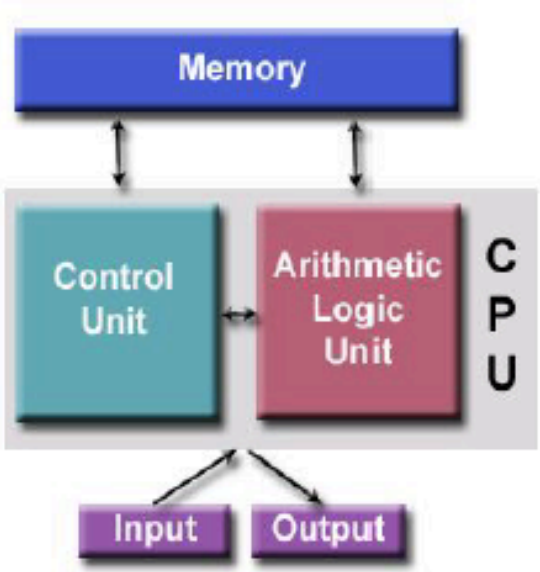
- Acontece quando duas ou mais *threads* tentam atualizar, ao mesmo tempo, uma mesma variável ou quando uma *thread* atualiza uma variável e outra *thread* acessa o valor dessa variável ao mesmo tempo.

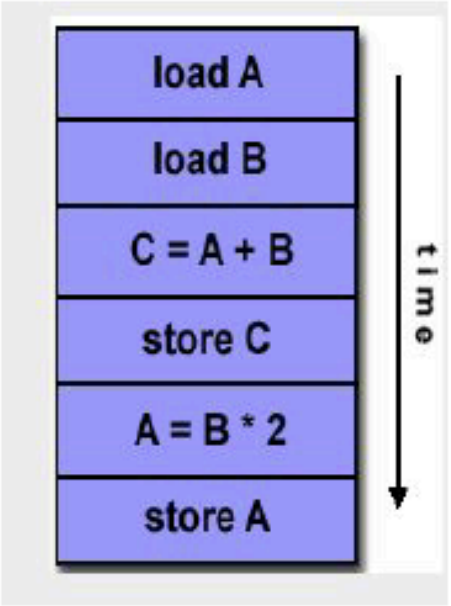
- Dessa forma, as diretivas de sincronização garantem que o acesso ou atualização de uma determinada variável compartilhada aconteça no momento certo.



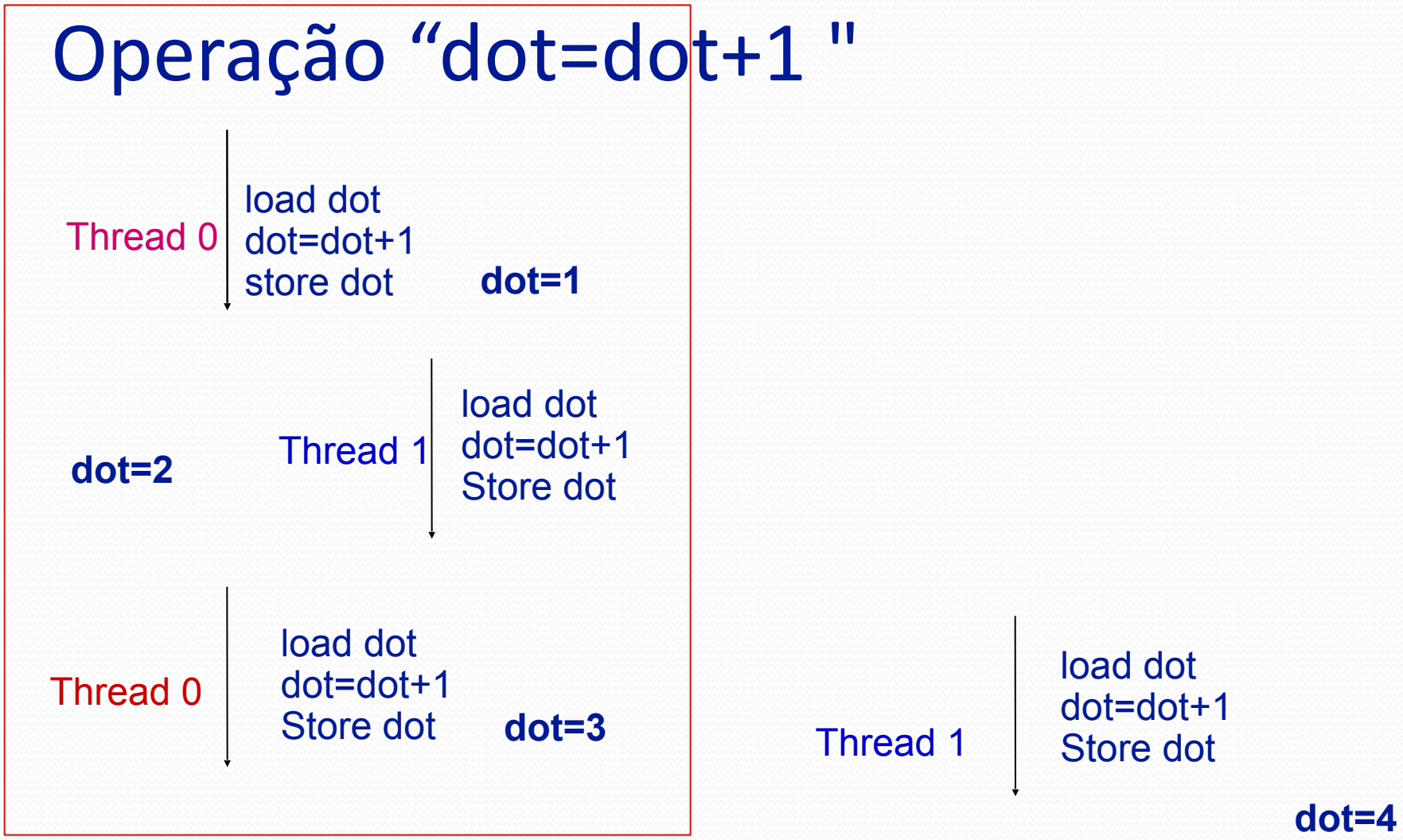
Arquitetura Von Neuman 1945
(Processador Sequencial)

Operações: C=A+B
A= B*2



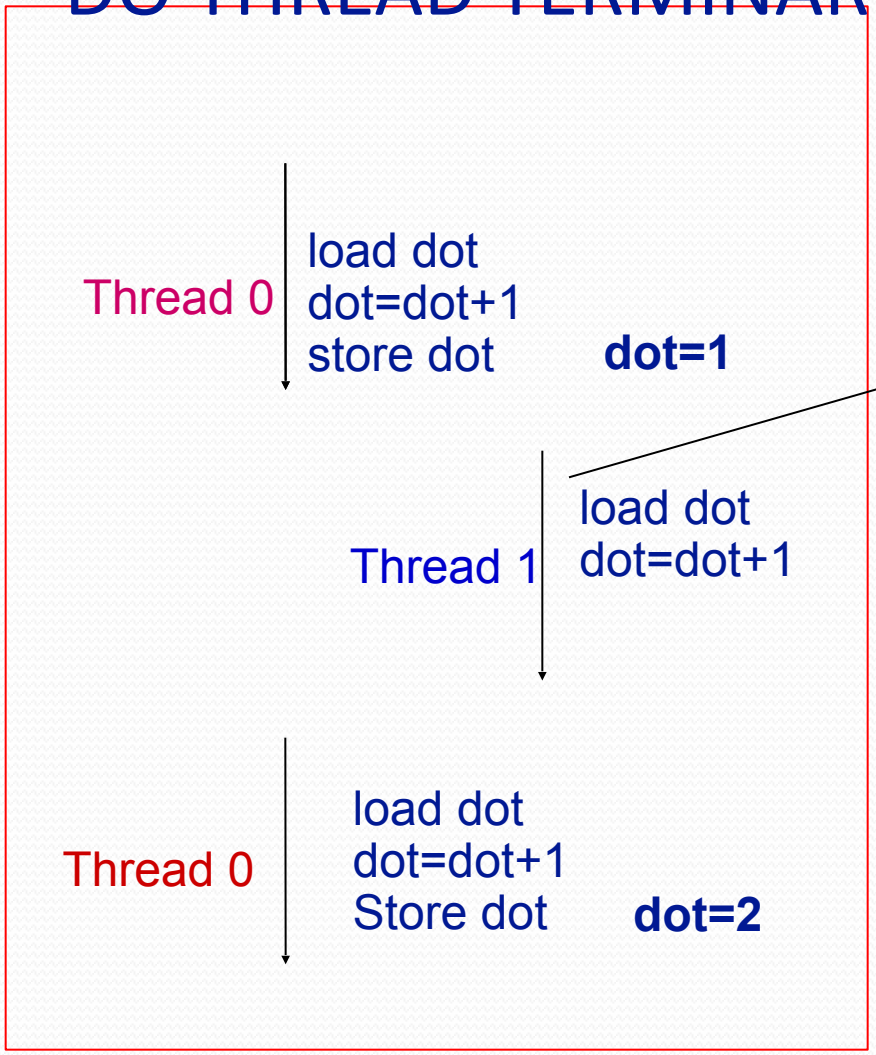


Exemplo_08.c:
Operação “dot=dot+1 "

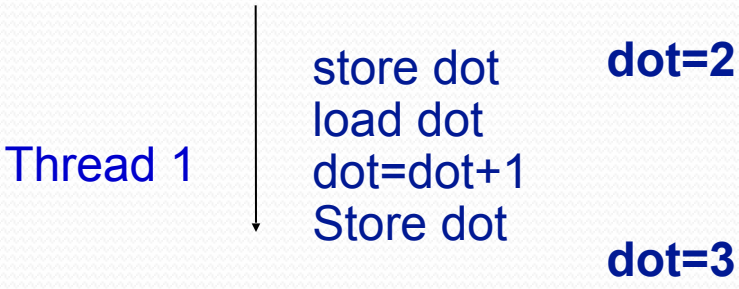


Para dot=0 no inicio no final de 4 execuções temos dot=4

PROBLEMA 1: INTERRUPÇÃO DO RUNTIME ANTES DO THREAD TERMINAR A OPERAÇÃO DE ESCRITA

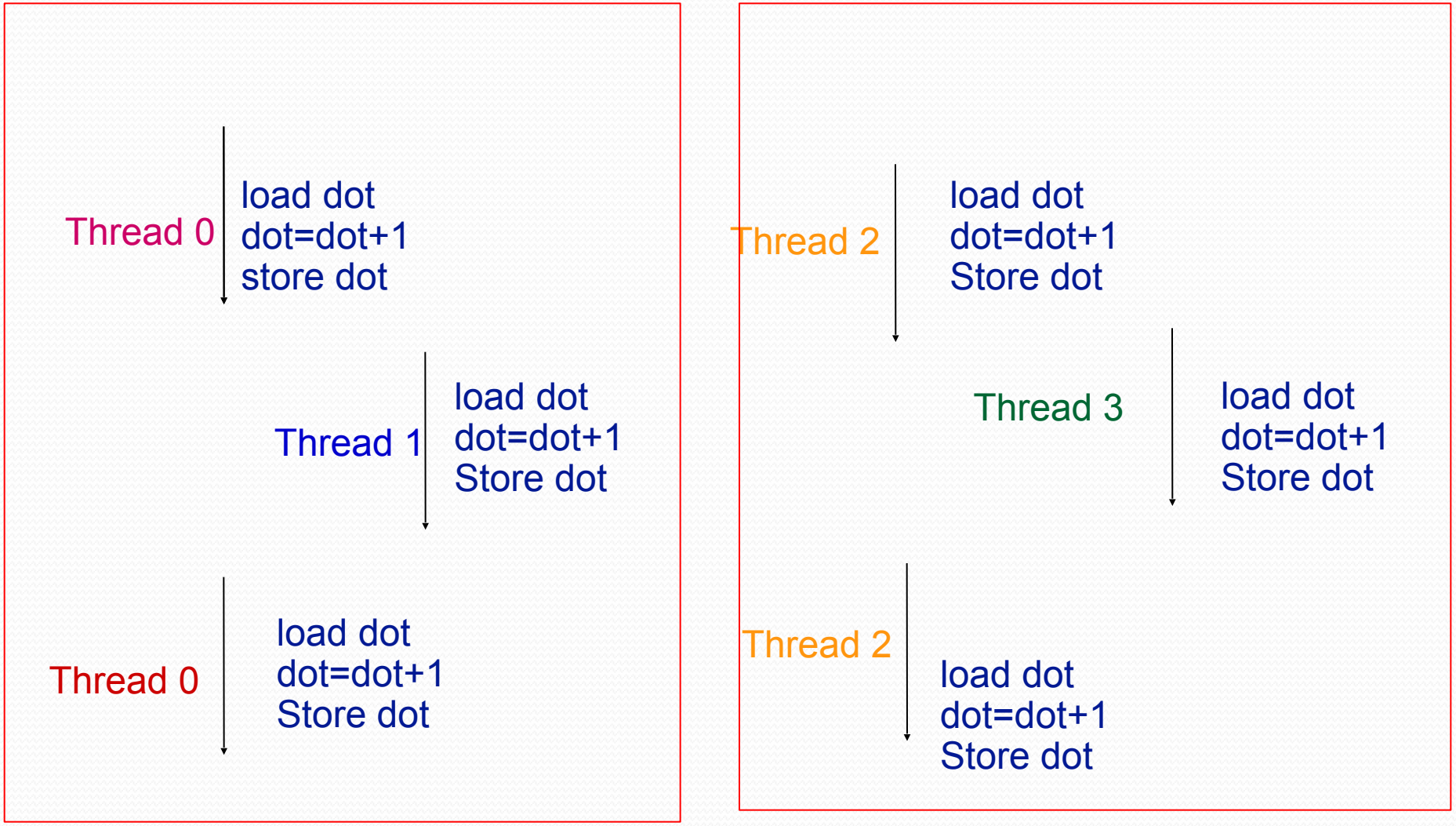


GERA PROBLEMA QUANDO O RUNTIME INTERROMPE A EXECUÇÃO E TROCA DE THREAD ANTES DA EXECUÇÃO DA OPERAÇÃO 'STORE"



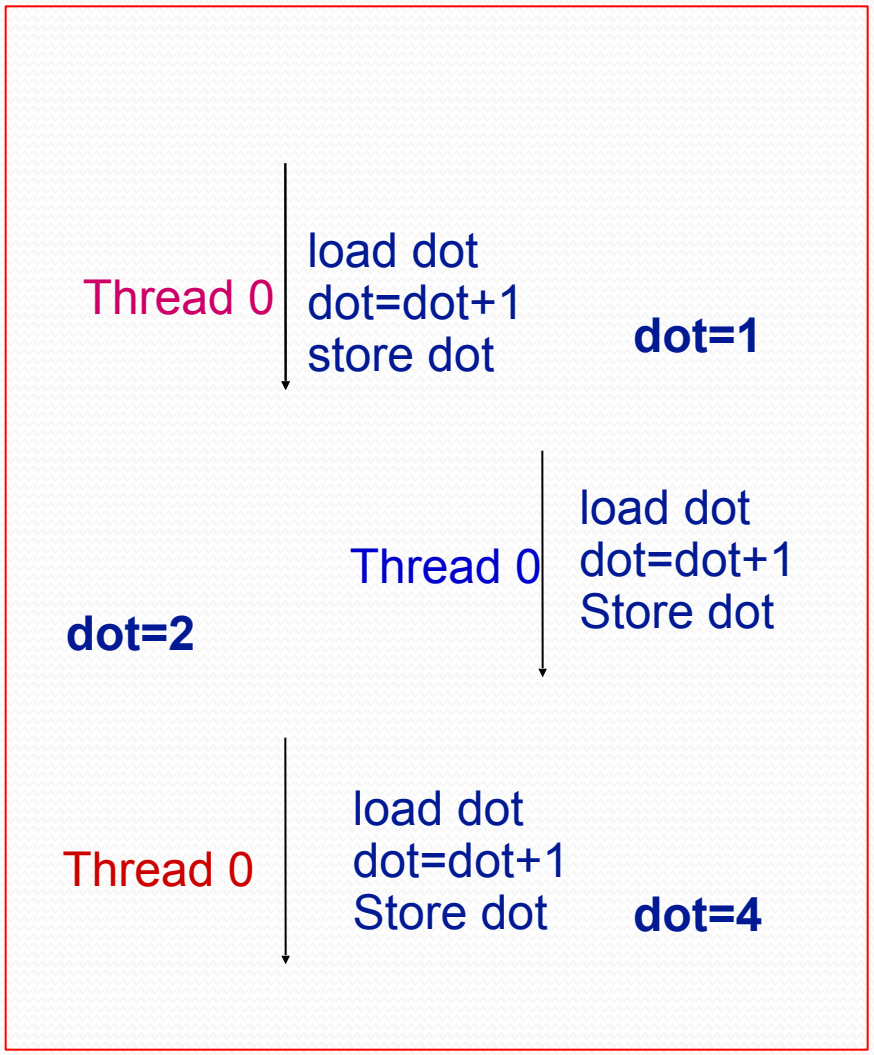
Para dot=0 no inicio no final de 4 execuções temos dot=3

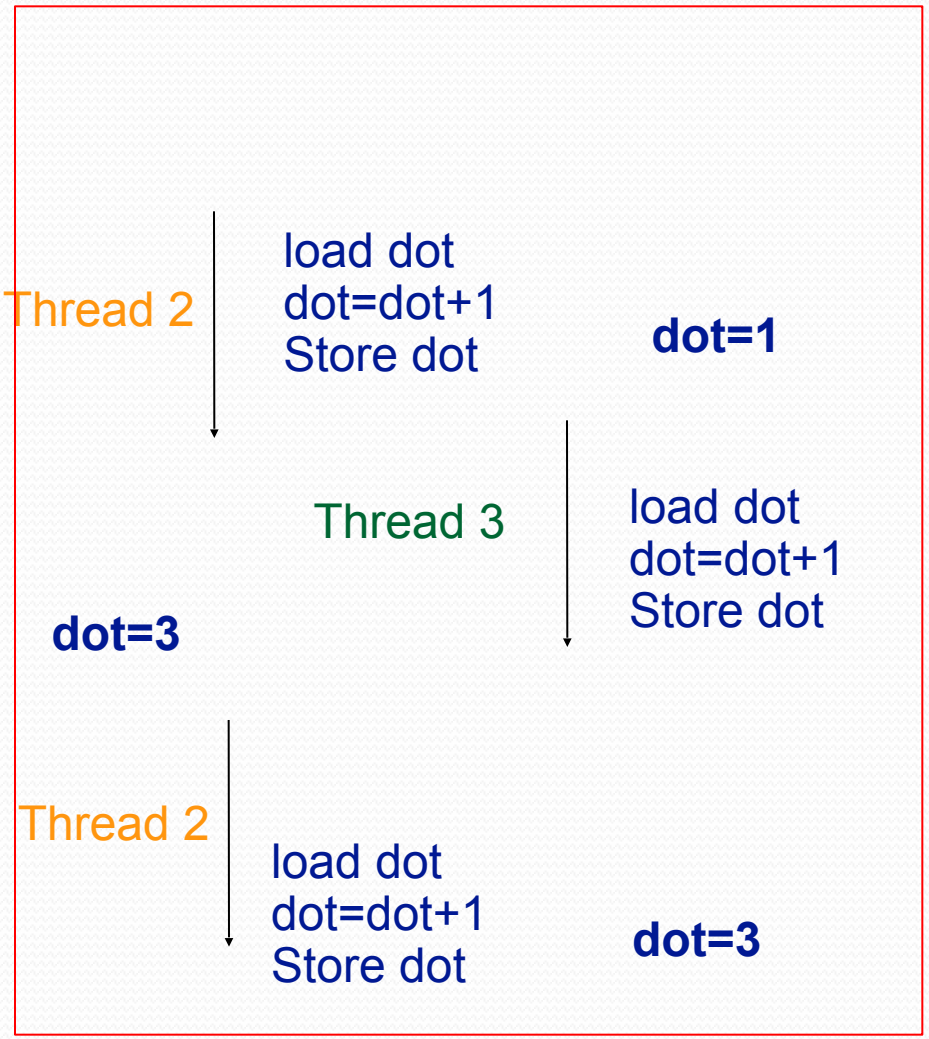
OPERAÇÃO DE ESCRITA POR DIVERSAS THREADS EM UMA MESMA VARIÁVEL GLOBAL



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

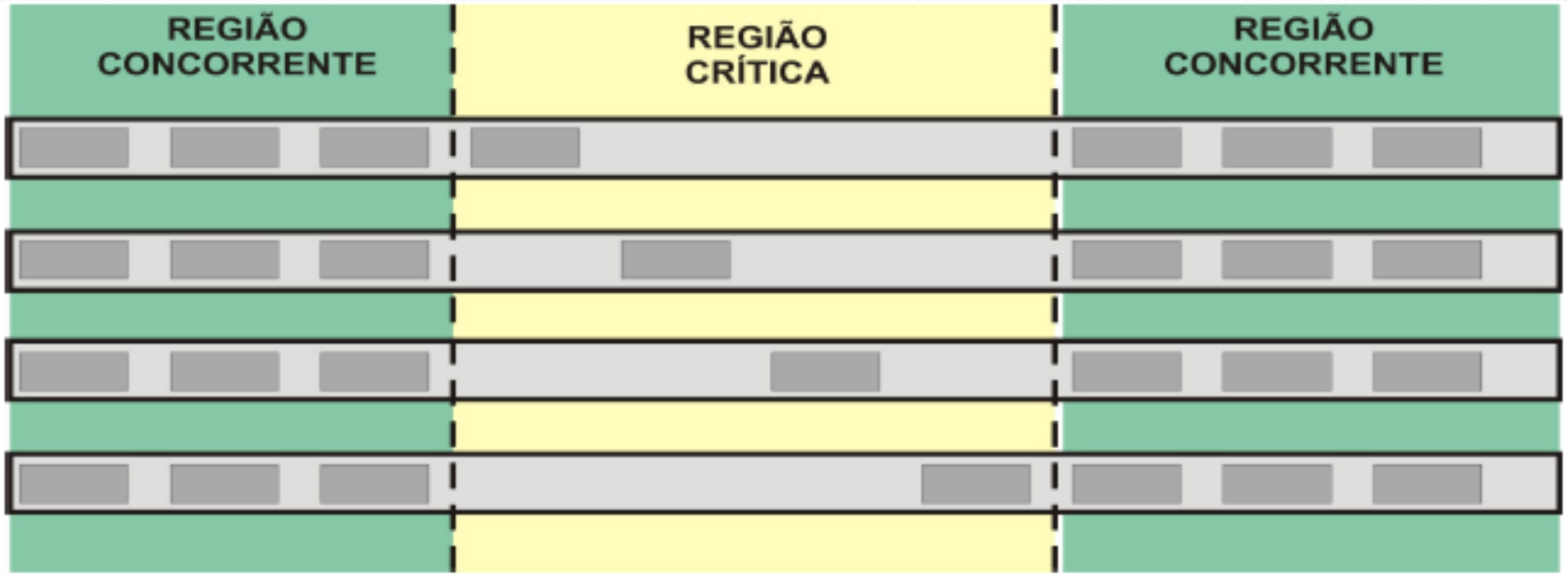
PROBLEMA 2: ESCRITADAS DESORDENADAS NA VARIÁVEL GLOBAL





Para dot=0 no inicio no final de 6 execuções temos dot=3

Operações em regiões com “Condição de corrida” (escrita simultânea em uma mesma variável global por diversas threads) devem ser executadas serialmente



Diretório exemplo_8

- $ ./exemplo_08_corrida.c 100

- Observe que a cada execução do código acima o resultado de saída é diferente, ou seja, o código gera um resultado “inconsistente”, devido a condição de corrida gerada pela escrita simultânea de diversas thread na variável global dos.
-

exemplo_08_sem_corrida_critical.c

```
#pragma omp critical
```

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

Observe que o resultado passou a ser consistente, ou seja, será o mesmo.

exemplo_08_sem_corrida.c

```
#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.

Permite a execução de apenas uma operação atômica

LIMITATIONS OF ATOMIC OPERATIONS

Read : operations in the form `v = x`

Write : operations in the form `x = v`

Update : operations in the form `x++`, `x--`, `--x`, `++x`, `x` *binop*`=` *expr*
and `x = x` *binop expr*

Capture : operations in the form `v = x++`, `v = x-`, `v = -x`, `v = ++x`,
`v = x` *binop expr*

- Here x and v are scalar variables
- *binop* is one of `+`, `*`, `-`, `-` `/`, `&`, `^`, `|`, `«`, `»`.
- No "trickery" is allowed for atomic operations:
  - no operator overload,
  - no non-scalar types,
  - no complex expressions.

exemplo_08.c

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

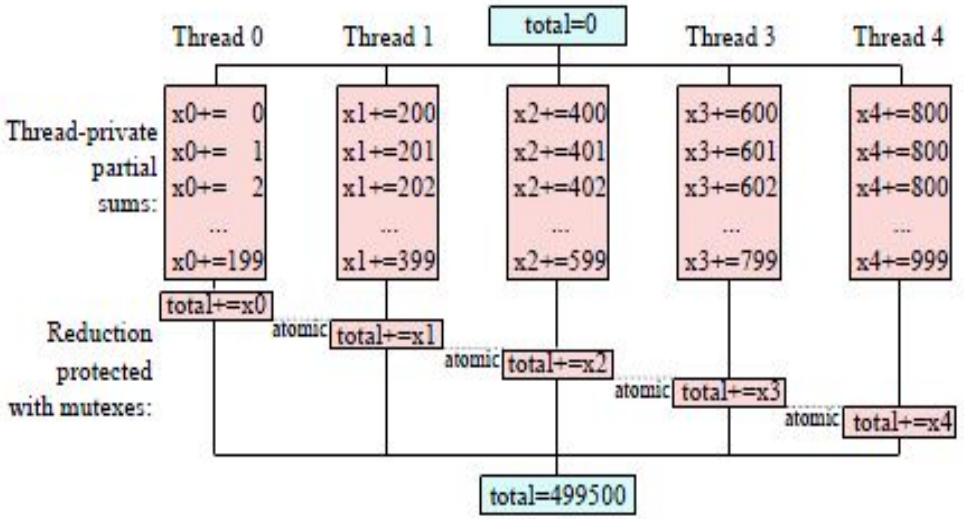
Operador	Valor Inicial
&	~0
|	0
&&	1
||	0

exemplo_08_critical_reduction.c
exemplo_08_atomic_reduction.c

AVOIDING RACES WITH THREAD-PRIVATE STORAGE

Correct and efficient code:

```
1  int total = 0;
2  #pragma omp parallel
3  {
4    int total_thr = 0;
5  #pragma omp for
6    for (int i=0; i<n; i++)
7      total_thr += i;
8
9  #pragma omp atomic
10   total += total_thr;
11
12 }
```



exemplo_08_critical_reduction.c

```
//Inicilializao da variavel dot
        dot = 0;
//Calculo do produto escalar
        #pragma omp parallel
         {
          int aux_dot=0;
        #pragma omp for
           for(i=0;i<num;i++)
                aux_dot += V1[i]*V2[i];

                #pragma omp critical
                {
                dot+=aux_dot;
                }
         }
    printf(" \n\n End Of Execution ::: dot = %d\n\n\n",dot);
        return 0;
```

exemplo_08_atomic_reduction.c

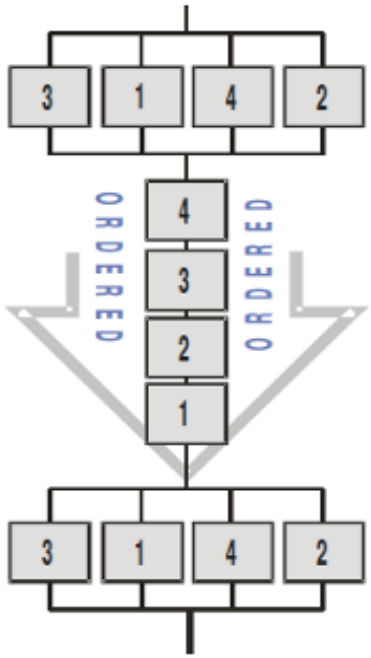
```
//Inicilializao da variavel dot
        dot = 0;
//Calculo do produto escalar
        #pragma omp parallel
         {
          int aux_dot=0;
        #pragma omp for
           for(i=0;i<num;i++)
                aux_dot += V1[i]*V2[i];

                #pragma omp atomic
                dot+=aux_dot;
         }
    printf(" \n\n End Of Execution ::: dot = %d\n\n\n",dot);
        return 0;
```

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++)
```



VARIÁVEIS “PRIVATE”
Só existem dentro da região paralela e não são inicializadas.

- **exemplo_04_1.c** (id e m são variáveis globais na região paralela)
- **exemplo_04_1_private** (id e m são variáveis private, não são inicializadas na região paralela)
- **exemplo_04_firstprivate** (inicializa a variável privativa m com valor global que tinha antes da região paralela)
- **exemplo_04_lastprivate** (“i” armazena na variável global o último valor dentro da região paralela)

Observe que ao retirar o lastprivate da variável i ela volt a ter o valor da variável global i.

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 privativas (var) das outras threads.
-

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);
```

VI-DIRETIVAS DE VETORIZAÇÃO

SIMULTANEOUS THREADING AND VECTORIZATION

Sometimes the compiler may need a little help:

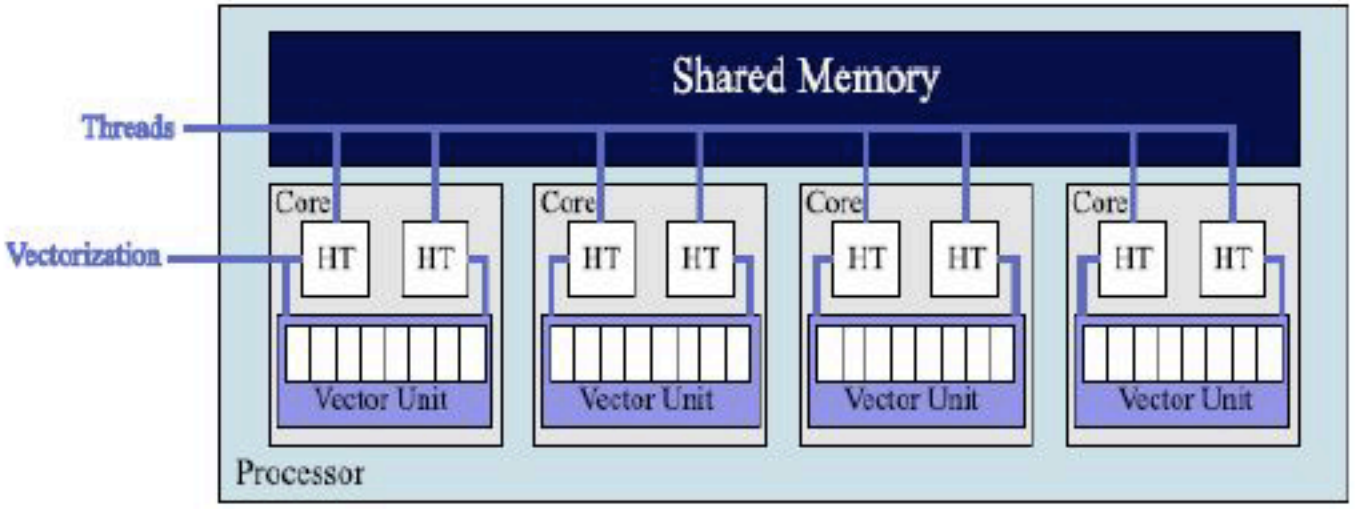
```
1  const int STRIP_SIZE = 128; // A multiple of vector length
2  const int nTrunc = n - n%STRIP_SIZE; // A multiple of vector length
3
4  #pragma omp parallel for
5  for (int ii = 0; ii < nTrunc; ii += STRIP_SIZE) // Thread parallelism in outer
6  #pragma simd
7    for (int i = ii; i < ii + STRIP_SIZE; i++) // Vectorization in inner loop
8      DoSomeWork(A[i]);
9
10 // Remainder loop:
11 for (int i = nTrunc; i < n; i++)
12   DoSomeWork(A[i]);
```

Existem diversos níveis de paralelismo em um processador multicore

PARALLELISM

CORES – multiple instructions on multiple data elements (MIMD)

VECTORS – single instruction on multiple data elements (SIMD)

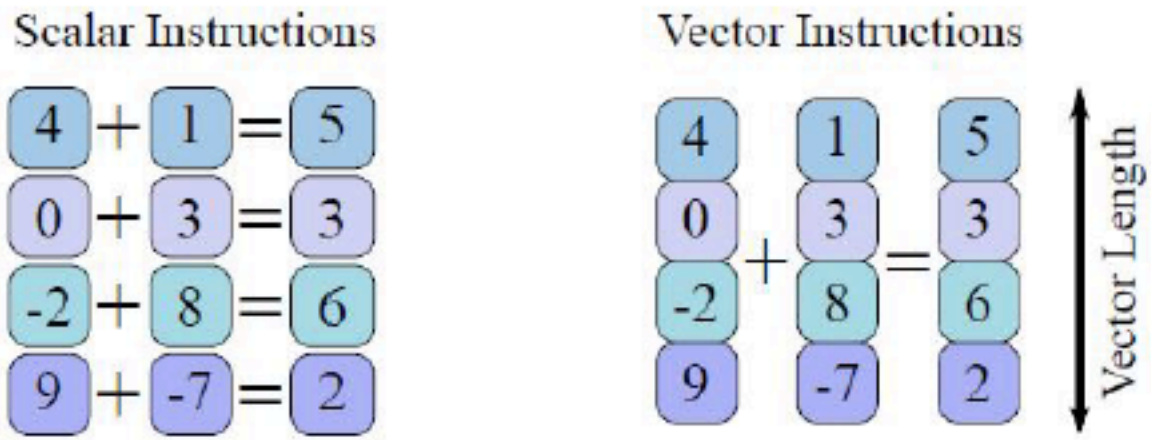


Unbounded growth opportunity, but **not automatic**

Vetorização: permite a execução de uma mesma instrução em vários dados

SHORT VECTOR SUPPORT

Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.



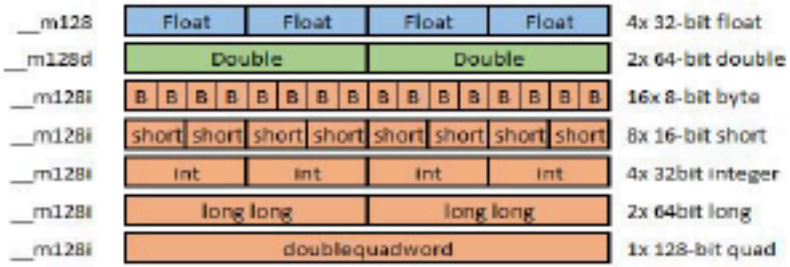
SSE & AVX Registers

SSE and AVX have 16 registers each. On SSE they are referenced as XMM0-XMM15, and on AVX they are called YMM0-YMM15. XMM registers are 128 bits long, whereas YMM are 256bit.

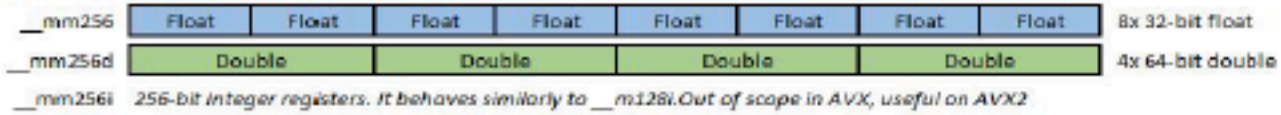
SSE adds three typedefs: `__m128` , `__m128d` and `__m128i` . Float, double (d) and integer (i) respectively.

AVX adds three typedefs: `__m256` , `__m256d` and `__m256i` . Float, double (d) and integer (i) respectively.

SSE Data Types (16 XMM Registers)



AVX Data Types (16 YMM Registers)



> **NOTE:** XMM and YMM overlap! XMM registers are treated as the lower half of the corresponding YMM register. This can introduce some performance issues when mixing SSE and AVX code.

Floating point datatypes (__m128, __m128d, __m256 and __m256d) have only one kind of data structure. Because of this, GCC allows for access to data components as an array. I.e: This is valid:

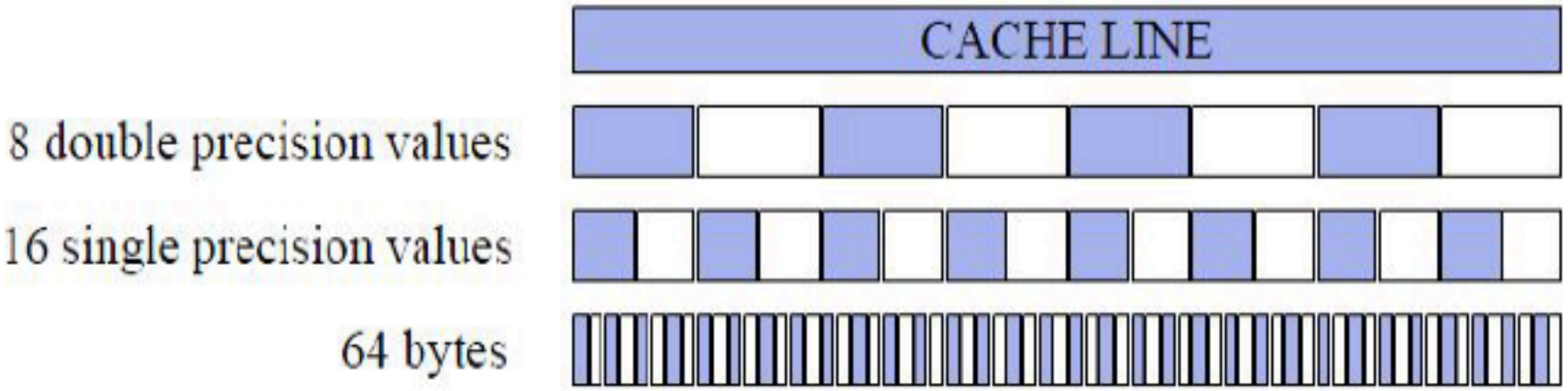
```
__m256 myvar = _mm256_set1_ps(6.665f); //Set all vector values to a single float
myvar[0] = 2.22f;                      //This is valid in GCC compiler
float f = (3.4f + myvar[0]) * myvar[7]; //This is valid in GCC compiler
```

__m128i and __m256i are unions, so the datatype needs to be referenced. I have not found a proper way to get the union declaration, so I use _mm_extract_epiXX() functions to retrieve individual data values from integer vectors.

A transferência de um bloco de dados na arquitetura Intel é 64 bytes

CACHE LINES

- Minimal block of data transferred between memory and cache
- 64 bytes long in Intel Architecture
- Aligned on 64-byte boundaries in memory



Para obter informações sobre a arquitetura do processador:

DETECTING AVAILABLE INSTRUCTIONS

In the OS:

```
[student@cdt ~]% cat /proc/cpuinfo
...
fpu_exception   : yes
cpuid level     : 11
wp              : yes
flags           : fpu vme de pse tsc msr pae mce
cx8 apic mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 ss ht syscall nx lm constant_tsc
unfair_spinlock pni ssse3 cx16 sse4_1 sse4_2
x2apic popcnt aes hypervisor lahf_lm fsgsbase
bogomips        : 5985.17
clflush size    : 64
cache_alignment: 64
address sizes   : 46 bits physical, 48 bits virtual
...
```

In code (see also):

```
1  // Intel compiler
2  // preprocessor macros:
3
4  #ifdef __SSE__
5  // ...SSE code path
6  #endif
7
8  #ifdef __SSE4_2__
9  // ...SSE code path
10 #endif
11
12 #ifdef __AVX__
13 // ...AVX code path
14 #endif
```

Diretivas de compilação para fornecer a arquitetura do processador

TARGETING A SPECIFIC INSTRUCTION SET

`-x[code]` to target specific processor architecture

`-ax[code]` for multi-architecture dispatch

code	Target architecture
MIC-AVX512	Intel Xeon Phi processors (KNL)
CORE-AVX512	Future Intel Xeon processors
CORE-AVX2	Intel Xeon processor E3/E5/E7 v3, v4 family
AVX	Intel Xeon processor E3/E5 and E3/E5/E7 v2 family
SSE4.2	Intel Xeon processor 55XX, 56XX, 75XX and E7 family
host	architecture on which the code is compiled

A diretiva –qopt-report gera um arquivo com informações sobre a compilação, para verificar se ocorreu compilação automática

AUTOMATIC VECTORIZATION OF LOOPS

```
1  #include <cstdio>
2
3  int main(){
4    const int n=1024;
5    int A[n] __attribute__((aligned(64)));
6    int B[n] __attribute__((aligned(64)));
7
8    for (int i = 0; i < n; i++)
9      A[i] = B[i] = i;
10
11   // This loop will be auto-vectorized
12   for (int i = 0; i < n; i++)
13     A[i] = A[i] + B[i];
14
15   for (int i = 0; i < n; i++)
16     printf("%2d %2d %2d\n",i,A[i],B[i]);
17 }
```

```
vega@lyra% icpc autovec.cc -qopt-report
vega@lyra% cat autovec.optrpt
...
LOOP BEGIN at autovec.cc(12,3)
remark #15399: vectorization support:
unroll factor set to 2 [autovec.cc(12,3)]
remark #15300: LOOP WAS VECTORIZED
[autovec.cc(12,3)]
LOOP END
...
vega@lyra% ./a.out
0  0  0
1  2  1
2  4  2
3  6  3
4  8  4
...
```

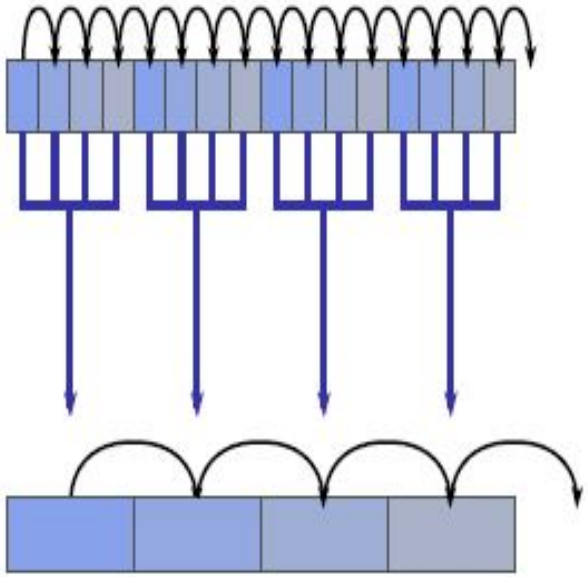
(Linha, coluna)

Situações que implementam vetorização automática

LIMITATIONS ON AUTOMATIC VECTORIZATION

- Innermost loops*
- Known number of iterations
- No vector dependence
- Functions must be SIMD-enabled

* `#pragma omp simd` to override



Condicionais dentro de um enlace diminuem o desempenho da vetorização: são executadas várias versões

MOVE BRANCHES OUTSIDE OF LOOPS

```
1  // Elegant, but bad for performance
2  for (i = 0; i < n; i++) {
3    if (i == 0) {
4      // Absorbing boundary
5      B[i] = 0.0;
6    } else if (i == n - 1) {
7      // Injection at boundary
8      B[i] = A[i] + 1.0;
9    } else {
10     // Diffusion between boundaries
11     B[i] = 0.25*(A[i-1] +
12                  2.0*A[i] + A[i+1]);
13   }
14 }
```

```
1  // Moving branches out of loops
2
3
4  // Absorbing boundary
5  B[i] = 0.0;
6
7  for (i = 1; i < n - 1; i++) {
8    // Diffusion between boundaries
9    B[i] = 0.25*(A[i-1] + 2.0*A[i] +
10                              A[i+1]);
11 }
12
13 // Injection at boundary
14 B[n-1] = A[n-1] + 1.0;
```

É mais eficiente fazer código “redundante” para retirar a condicional do enlace e permitir a execução de apenas uma versão e aumentar desempenho da vetorização

REDUNDANT CODE IS OK

```
// Elegant, but bad for performance
for (ii = 0; ii < n; ii+=16) {
  for (i = ii; i < ii+16; i++)
    // Branch causes unnecessary
    // masking of vector iterations
    if (i < n) {
      A[k*n + i] = ...
    }
}
```

```
// Redundant code, but faster
const int nTrunc = n - n%16;
for (ii = 0; ii < nTrunc; ii+=16) {
  for (i = ii; i < ii+16; i++)
      A[k*n + i] = ...

for (i = nTrunc; i < n; i++)
  A[k*n + i] = ...
}
```

Diretiva SIMD

VECTORIZE MORE LOOPS: `#pragma omp simd`

Used to "enforce vectorization of loops", which includes:

- Loops with SIMD-enabled functions
- Second innermost loops
- Failed vectorization due to compiler decision
- Where guidance is required (vector length, reduction, etc.)

See OpenMP reference for syntax; `#pragma simd`

IV.I-Loop with SIMD-enable function

Diretivas para o compilador vetorizar funções é necessária quando a função está em arquivo separado do arquivo fonte (biblioteca).

SIMD-ENABLED FUNCTIONS

Define function in one file (e.g., library), use in another

```
1  // Compiler will produce 3 versions:
2  #pragma omp declare simd
3  float my_simple_add(float x1, float x2){
4    return x1 + x2;
5  }
```

```
1  // May be in a separate file
2  #pragma omp simd
3  for (int i = 0; i < N, ++i) {
4    output[i] = my_simple_add(inputa[i], inputb[i]);
5  }
```

I V. 2- Second Innermost loop

Vetorização automática é ineficiente porque o “inner loop” (i) possui “stride access”.

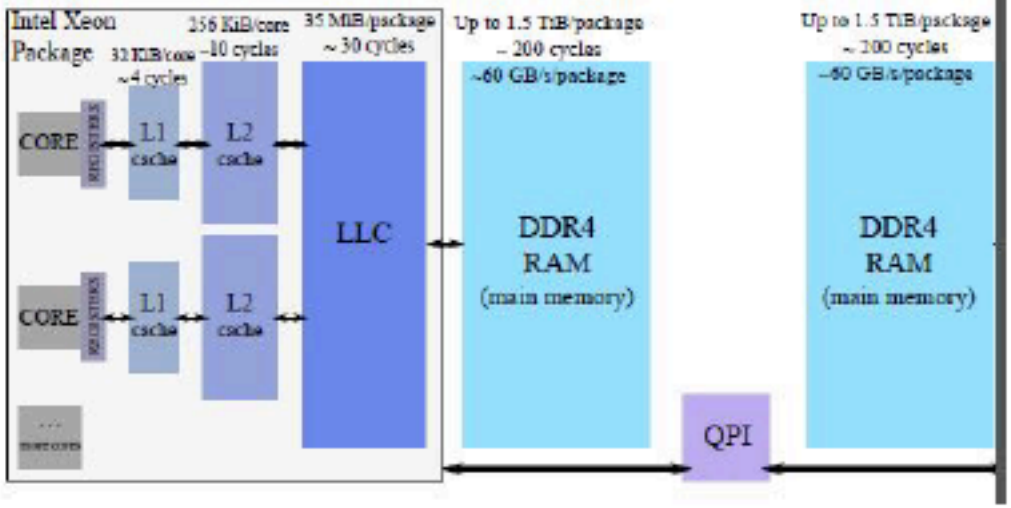
EXAMPLE FOR #pragma omp simd

```
1  const int N=128, T=4;
2  float A[N*N], B[N*N], C[T*T];
3
4  for (int jj = 0; jj < N; jj+=T) // Tile in j
5    for (int ii = 0; ii < N; ii+=T) // and tile in i
6  #pragma omp simd    // Vectorize outer loop
7    for (int k = 0; k < N; ++k) // long loop, vectorize it
8      for (int i = 0; i < T; i++) { // Loop between ii and ii+T
9        // Instead of a loop between jj and jj+T, unrolling that loop:
10       C[0*T + i] += A[(jj+0)*N + k]*B[(ii+i)*N + k];
11       C[1*T + i] += A[(jj+1)*N + k]*B[(ii+i)*N + k];
12       C[2*T + i] += A[(jj+2)*N + k]*B[(ii+i)*N + k];
13       C[3*T + i] += A[(jj+3)*N + k]*B[(ii+i)*N + k];
14 }
```

Hierarquia de Memória afeta o desempenho: é importante acessar o dado de forma a obter localidade temporal e local

INTEL XEON CPU: MEMORY ORGANIZATION

- Hierarchical cache structure
- Two-way processors have NUMA architecture



Acessos de memória com stride diminuem o desempenho:

UNIT-STRIDE ACCESS

Unit-stride access is optimal:

```
1 for (int i = 0; i < n; i++)
2   A[i] += B[i];
```

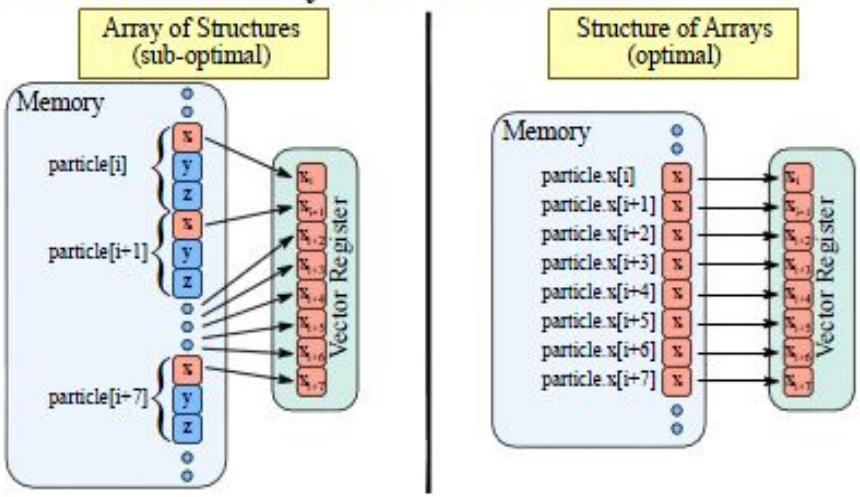
Non-unit stride is slower:

```
1 for (int i = 0; i < n; i++)
2   A[i*stride] += B[i];
```

Stochastic access may be vectorized (but not efficient):

```
1 for (int i = 0; i < n; i++)
2   A[offset[i]] += B[i];
```

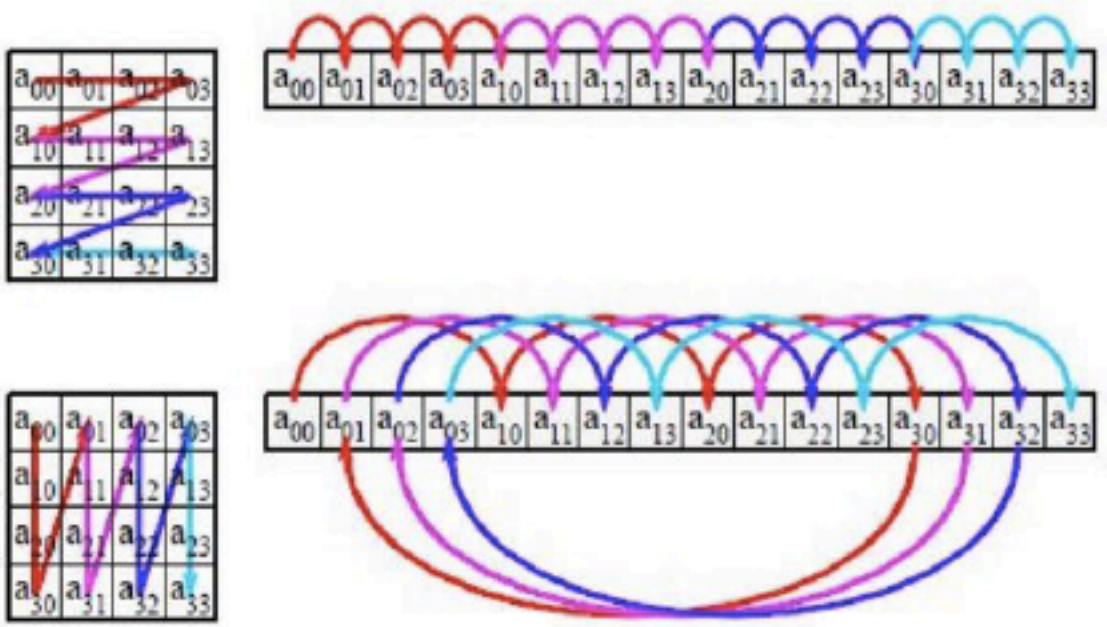
It may be a question of changing the order of loop nesting, but sometimes you need to modify data structures:



O loop deve percorrer a matriz evitando acessos de stride:

PRINCIPLE

Choose loop order to maintain unit-stride memory access



Compiler may or may not be able to automate loop permutation.

O “inner” loop passa para k, para percorrer a matriz sem “stride”

EXAMPLE: OVER-SIMPLIFIED MATRIX-MATRIX MULTIPLICATION

$$C = AB \quad \Leftrightarrow \quad C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$$

Before:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3   for (int j = 0; j < n; j++)
4 #pragma vector aligned
5     for (int k = 0; k < n; k++)
6       C[i*n+j]+=A[i*n+k]*B[k*n+j];
```

After:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3   for (int k = 0; k < n; k++)
4 #pragma vector aligned
5     for (int j = 0; j < n; j++)
6       C[i*n+j]+=A[i*n+k]*B[k*n+j];
```

sem_otimizacao.c

```
    //Aloca a memória necessária para as matrizes
    long int n = atol(argv[1]), i;
    printf("Allocating memory...\n");
    int *A = (int*) malloc(n * n * sizeof(int));
    int *B = (int*) malloc(n * n * sizeof(int));

    //Inicializa A e B com os valores 2 e 3
    printf("Initializing Matrices...\n");
    for(int i = 0;i < n;i++){
        for(int j = 0;i < n;i++){
            A[i*n + j] = 2;
            B[i*n + j] = 3;
        }
    }

    //Aloca memória para matriz C
    int *C = (int*) malloc(n * n * sizeof(int));

    //Multiplica A * B
# pragma omp parallel for
    for(int i = 0;i < n;i++){
        for(int j = 0;j < n;j++){
# pragma vector aligned
            for(int k = 0;k < n;k++){
                C[i*n +j] += A[i*n +k] * B[k*n +j];
            }
        }
    }
```

com_otimizacao.c

```
    //Aloca a memória necessária para as matrizes
    long int n = atol(argv[1]), i;
    printf("Allocating memory...\n");
    int *A = (int*) malloc(n * n * sizeof(int));
    int *B = (int*) malloc(n * n * sizeof(int));

    //Inicializa A e B com os valores 2 e 3
    printf("Initializing Matrices...\n");
    for(int i = 0;i < n;i++){
        for(int j = 0;i < n;i++){
            A[i*n + j] = 2;
            B[i*n + j] = 3;
        }
    }

    // Aloca memória para matriz C
    int *C = (int*) malloc(n * n * sizeof(int));

# pragma omp parallel for
    for(int i = 0;i < n;i++){
        for(int k = 0;k < n;k++){
# pragma vector aligned
            for(int j = 0;j < n;j++){
                C[i*n +j] += A[i*n +k] * B[k*n +j];
            }
        }
    }
```

I V.3 - Failed to vetorization due to Compiler decision

Situações que impedem vetorização automática: "dependência de dados"

TRUE VECTOR DEPENDENCE

- True vector dependence – vectorization impossible:

```
1 for (int i = 1; i < n; i++)
2   a[i] += a[i-1]; // dependence on the previous element
```

- Safe to vectorize:

```
1 for (int i = 0; i < n-1; i++)
2   a[i] += a[i+1]; // no dependence on the previous element
```

- May be safe to vectorize:

```
1 for (int i = 16; i < n; i++)
2   a[i] += a[i-16]; // no dependence if vector length <=16
```

Vetores que podem apontar para mesmo endereço, fazem o compilador gerar versão com e sem vetorização que causa “overhead” na execução.

ASSUMED VECTOR DEPENDENCE

Not enough information to confirm or rule out vector dependence:

```
1 void AmbiguousFunction(int n, int *a, int *b) {
2   for (int i = 0; i < n; i++)
3     a[i] = b[i];
4 }
```

- If a, b are not aliased or b>a, then safe to vectorize
- If a, b are aliased (e.g., b==a−1), requires scalar computation

Solução: Diretiva “ivdep” garante que os dois vetores não estarão apontando para mesmo endereço e torna execução mais eficiente

POINTER DISAMBIGUATION

Prevent multiversioning or allow vectorization with a directive:

```
1 #pragma ivdep
2   for (int i = 0; i < n; i++)
3     // ...
```

```
user@host% icpc -c code.cc -qopt-report -qopt-report-phase:vec
user@host% cat vdep.optrpt
...
LOOP BEGIN at code.cc(4,1)
   remark #25228: LOOP WAS VECTORIZED
LOOP END
...
```

Alternative: keyword `restrict` – more fine-grained, weaker.

Resumo das Diretivas de Vetorização :

VECTORIZATION DIRECTIVES

- `#pragma omp simd` Altera o loop a ser vetorizado
- `#pragma vector always` sempre vetorizar
- `#pragma vector aligned | unaligned` Garante que todo acesso a memória é alinhado
- `__assume_aligned` keyword
- `#pragma vector nontemporal | temporal` Stream store para evitar poluição de cache
- `#pragma novector` Impede vetorização
- `#pragma ivdep` Informa que os ponteiros não são ambíguos e pode ter ajuda do restrict
- `restrict` qualifier and `-restrict` command-line argument
- `#pragma loop count` Fornece informações sobre o número de interações a serem vetorizadas. Ajuda em loop unroll

https://colfaxresearch.com/how-series/

Research | Training | Services | About Us

Home » Training

Training

Browse our webinars and on-demand training on parallel programming, performance optimization for parallel processors, new technology and development tools. Some of our training programs feature remote access to training servers, original programming exercises and certificates of accomplishment.

"HOW" Series: Deep Dive

Learn More