OpenMP

Aula 08

OpenMP

- Devido a falta de padrão nas diretivas (normas) de compartilhamento
- Desenvolvido e mantido pelo grupo OpenMP ARB (Architecture Review Board)
- Teve início por volta de 1997 em Fortran
- Para C e C++ em 1998
- Open + MP ->padrão aberto + Multi Processing

OpenMP

- Programação de computadores paralelos com memória compartilhada
- A facilidade principal é a existência de um único espaço de endereçamento através de todo o sistema de memória

OpenMP

- Não é uma linguagem de programação
- É um padrão
- É uma API e um conjunto de diretivas
  - Criar programas paralelos
  - Memória compartilhada
  - Criação de conjunto de threads de forma automática e otimizada

OpenMP

- Objetivos
  - Ser o padrão de programação para arquiteturas de memória compartilhada
  - Estabelecer um conjunto simples e limitado de diretivas de programação
  - Possibilitar a paralelização incremental de programas sequenciais
  - Permitir implementações eficientes em problemas de granularidade fina, média e grossa

OpenMP

- Componentes
  - Bibliotecas de Funções
    - Omp_get_num_threads()
  - Diretivas de Compilação
    - #pragma omp parallel
  - Variáveis de Ambiente
    - OMP_NUM_THREADS

Diretivas e Sentinelas

- É uma linha especial de código fonte com significado especial, apenas para determinados compiladores
- Uma diretiva se distingue pela existência de uma sentinela no começo da linha
- As sentinas do OpenMP são
  - Fortran
    - !$OMP ou C$OMP ou *$OMP
  - C/C++
    - #pragma omp

Categorias de Diretivas

- Regiões Paralelas – Construtor Paralelo
  - Omp parallel
- Compartilhamento dos dados
  - Omp shared, private, ...
- Distribuição do trabalho – Instrutor Paralelo
  - Omp for
- Sincronizações
  - Omp atomic, critical, barrier,....
- Runtime funções e variáveis de ambiente
  - Omp_set_num_threads(), omp_set_lock()
  - Omp_schedule, ...

Pragma

- Instruções especiais de pré-processamento
- Permitem às aplicações comportamentos que não pertencem a especificação básica de C
- O compilador que não suporta pragmas ignora elas

Modelo de Programação

- Paralelismo Explícito
  - Cabe ao programador anotar as tarefas para execução paralela
  - Definir pontos de sincronização
  - Anotações feitas por meio de diretivas de programação embarcadas no código do programa

Modelo de Programação

- Multithread Implícito
  - Um processo é visto por um conjunto de threads que se comunicam por meio da utilização de variáveis compartilhadas
  - Feito de forma implícita pelo ambiente de execução sem que o programador tenho que se preocupar com tais detalhes
  - Espaço de endereçamento global compartilhado entre as threads

Modelo de Programação

- Variáveis podem ser compartilhadas ou privadas para cada thread
- Controle, manuseio e sincronização das variáveis envolvidas nas tarefas paralela é transparente ao programador

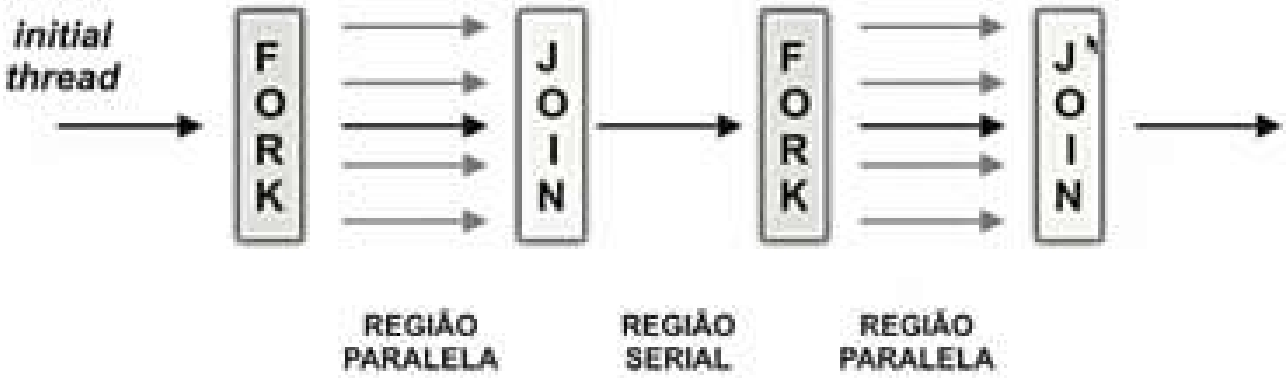
Modelo de Execução

- Todos os programas iniciam sua execução com um processo mestre
- A master thread executa sequencialmente até encontrar um construtor paralelo, momento em que cria um team de thread
- O código delimitado pelo construtor paralelo é executado em paralelo pelo master thread e pelo team de threads
- Quando completa a execução paralela, o team de threads sincroniza em uma barreira implícita com o master thread

Modelo de Execução

- O team de threads termina a sua execução e o master thread continua a execução sequencialmente até encontrar um novo construtor paralelo
- A sincronização é utilizada para evitar a competição entre as threads
- Modelo de programação conhecido como fork-join (SO)

Modelo de Execução



Exemplo

- A ordem da execução não é determinístico
- Para alterar a quantidade de threads pode-se alterar a variável de ambiente
  - export OMP_NUM_THREADS=8

Por que usar OpenMP?

- Vantagens
  - Facilidade de conversão de programas sequenciais em paralelos
  - Maneira simples de explorar o paralelismo
  - Fácil compreensão e uso das diretivas
  - Minimiza a interferência na estrutura do algoritmo
  - Possibilita o ajuste dinâmico do número de threads
  - Compila e executa em ambientes paralelos e sequenciais

Construtor Paralelo

- É a diretiva mais importante do OpenMP
- Responsável pela indicação da região do código que será executada em paralelo
- Caso não seja especificado, o programa será executado de forma sequencial

SINTAXE:

```
#pragma omp parallel [cláusula,...] novalinha
   Instrução
```

Construtor Paralelo

- Uma região paralela é chamada de inativa
  - Quando é executada por apenas uma thread
- E ativa quando possui mais de uma
- Pode-se ativar uma região paralela aninhada
  - omp_set_nested()
- No final de toda região paralela existe uma barreira implícita(join)
- Faz com que as threads esperem até que todas as threads cheguem naquele ponto
- Entretanto há uma cláusula que pode ser usada para que o programador decida sobre a existência da barreira

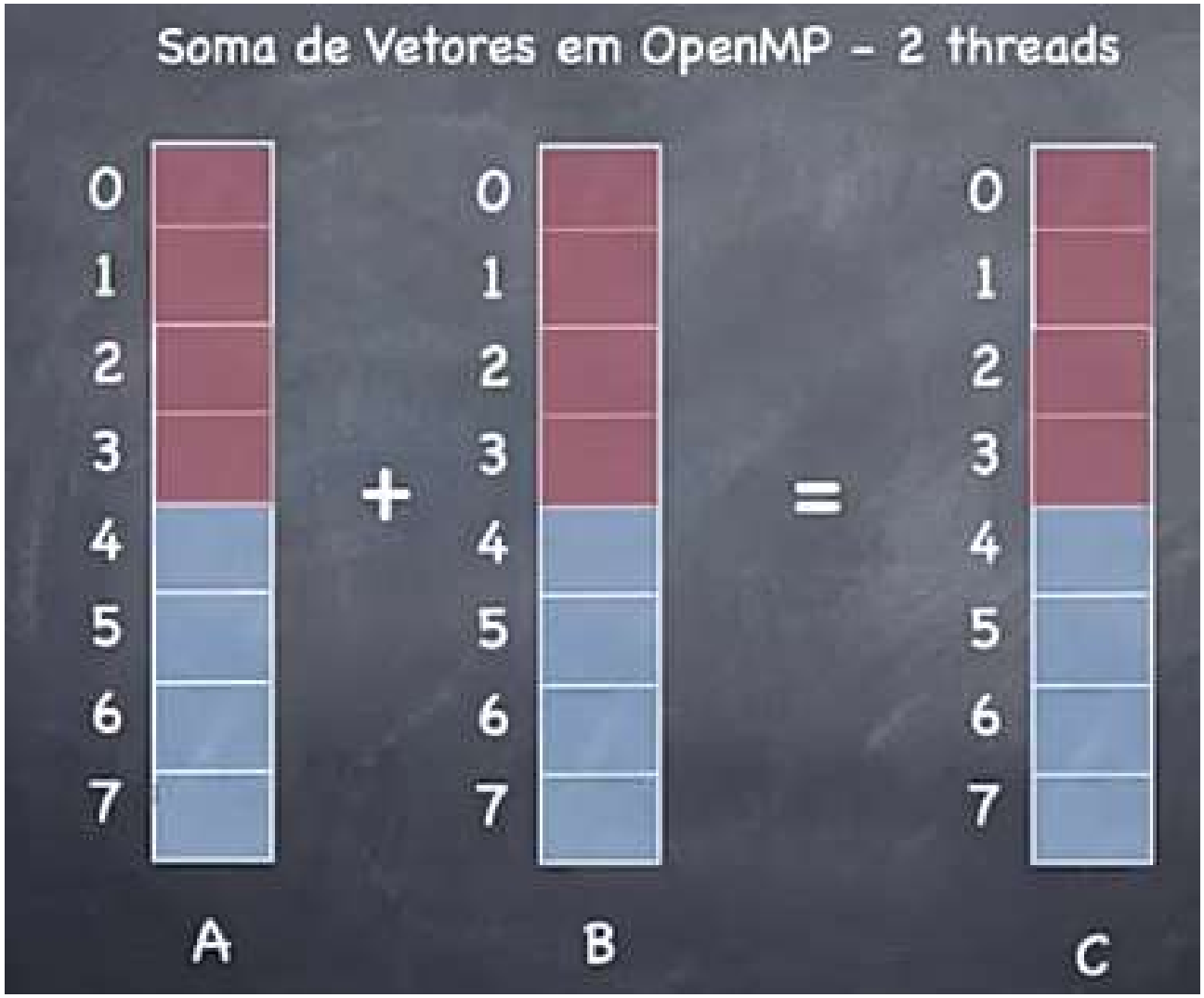
Construtor Paralelo

- As cláusulas que podem ser utilizadas nesse construtor
  - If(expressão lógica)
  - Private (lista de variáveis)
  - Shared (lista de variáveis)
  - Firstprivate (lista de variáveis)
  - Default (shared |none)
  - Copyin (lista de variáveis)
  - Reduction (operador: lista de variáveis)
  - Num_threads(variável inteira)

Estudo de Caso

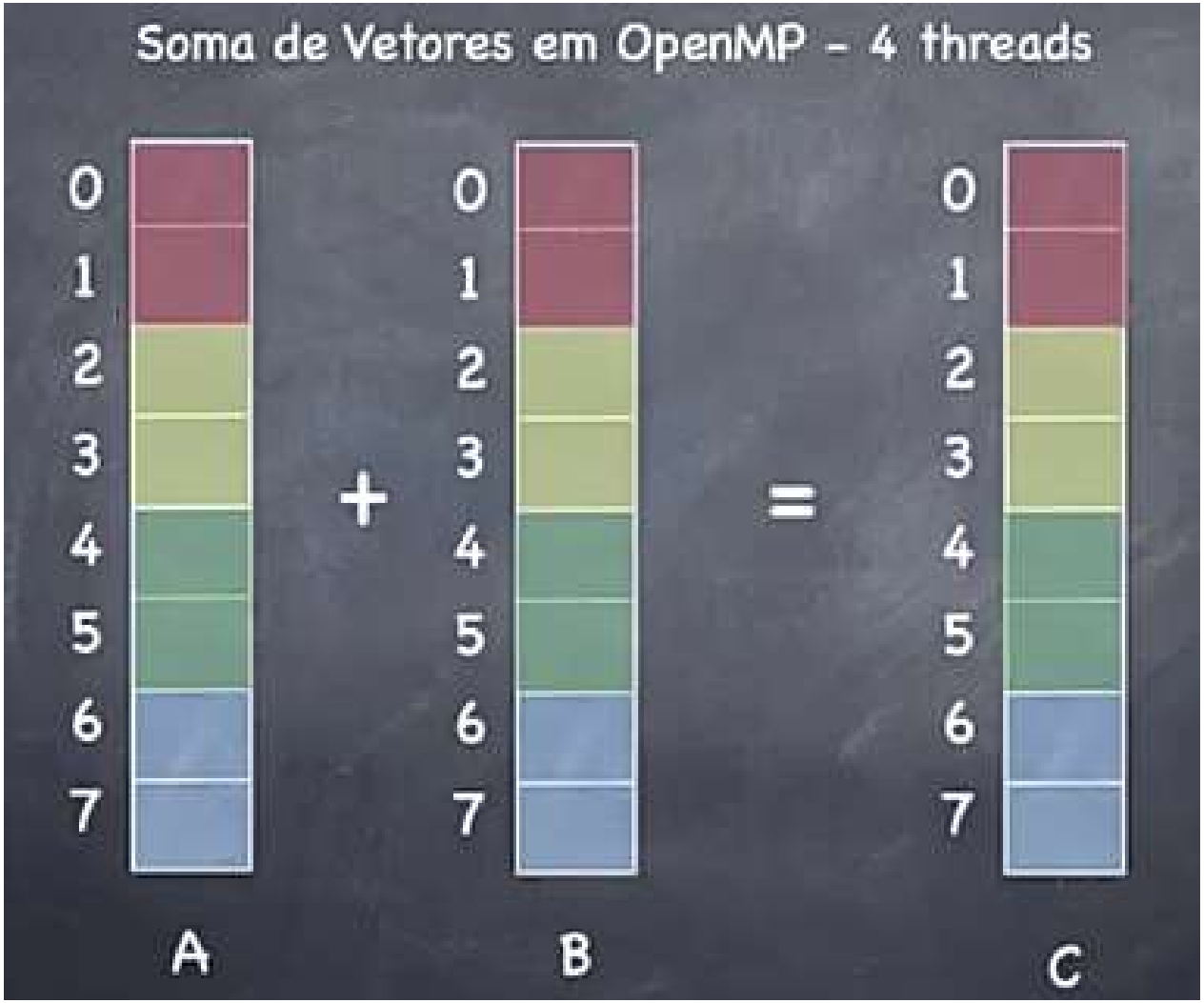
- Soma de Vetores
- Soma de Matrizes
- Não me preocupo com recursos compartilhados
- Possui a soma de cada posição não interfere em outras posições
- Torna o paralelismo bastante simples

Estudo de caso



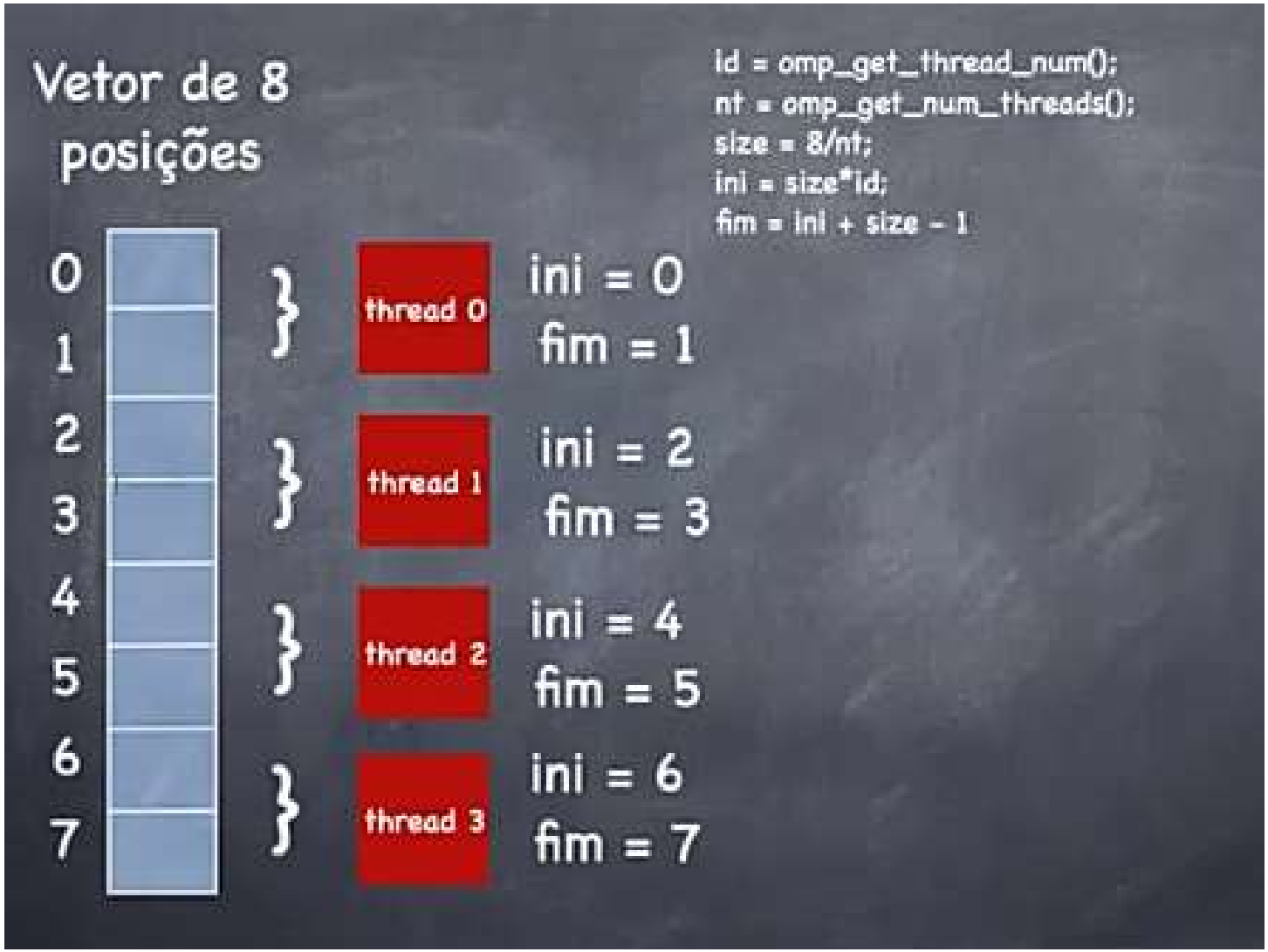
Speed up, teoricamente seria de 2X

Estudo de caso

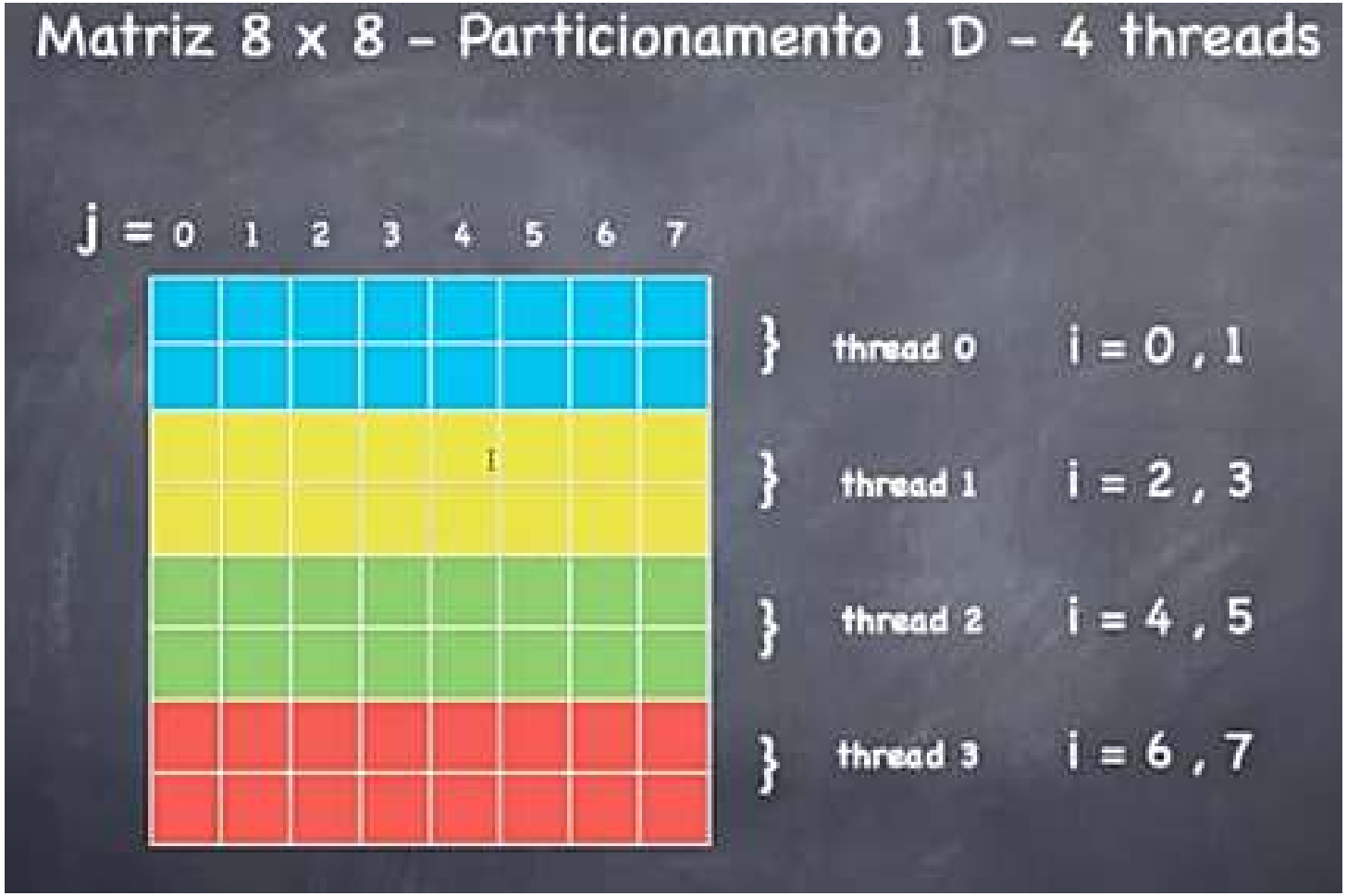


Speed up, teoricamente seria de 4X

Estudo de Caso



Estudo de Caso - Matriz



Construtores de Compartilhamento de Trabalho

- Responsáveis pela distribuição de trabalho entre as threads
- O trabalho não precisa mais ser dividido manualmente
- Indicam a maneira como o trabalho será dividido entre as threads
- Porém, não criam novas threads

Construtores de Compartilhamento de Trabalho

- Deve estar dentro de uma região paralela
- Existe, também, uma barreira implícita
- Em C/C++ existem três tipos de construtores de compartilhamento
  - Diretiva for
  - Diretiva sections
  - Diretiva single

Construtores de Compartilhamento de Trabalho

- Alguns cuidados
  - Cada construtor deve ser encontrado por todas as threads ou por nenhuma delas
  - A sequencia de construtores e barreiras deve ser a mesma para todas as threads do grupo

#pragma omp for

SINTAXE:

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

#pragma omp for

- As cláusulas que podem ser utilizadas
  - Private (lista de variáveis)
  - Firstprivate (lista de variáveis)
  - Lastprivate (lista de variáveis)
  - Reduction (operador: lista de variáveis)
  - Schedule(tipo, [tamanho])
  - Nowait
  - ordered

#pragma omp for

- Em C/C++, o construtor for só pode ser usado em estruturas de repetição no qual o número de iterações é previamente conhecido e não sofre alteração durante a execução
- O construtor for implementa SIMD

#pragma omp sections

SINTAXE:

```
#pragma omp sections[cláusula,...] novalinha
{
  #pragma omp section novalinha
  instrução
  #pragma omp section novalinha
  instrução
}
```

#pragma omp sections

- Além do construtor, que indica que cada thread vai executar um bloco diferente
- É necessário incluir a diretiva
- #pragma omp section
  - Para indicar qual instrução cada thread irá executar

#pragma omp sections

- As cláusulas que podem ser usadas
  - Private(lista de variáveis)
  - Firstprivate (lista de variáveis)
  - Lastprivate (lista de variáveis)
  - Reduction(operador: lista de variáveis)
  - nowait

#pragma omp sections

- Quando houver mais blocos do que threads
- Por outro lado, se houver mais threads
- Se houver apenas 1 thread
- Implementa MIMD

#pragma omp single

- Indica que o trecho de código abaixo desta diretiva deve ser executado apenas por uma thread
- As outras threads esperam em uma barreira implícita, no final do construtor single
- Até que a thread que encontrou o construtor termine a execução

#pragma omp single

SINTAXE:

```
#pragma omp single [cláusula,...]
    instrução ...
```

- As cláusulas que podem ser usadas no construtor
  - Private (lista de variáveis)
  - Firstprivate (lista de variáveis)
  - Copyprivate (lista de variáveis)
  - nowait

Construtores Combinados

SINTAXE:

```
#pragma omp parallel for   [cláusula,...]
      for-loop

#pragma omp parallel sections   [cláusula,...]
      novalinha
{
      #pragma omp section novalinha
            instrução
      #pragma omp section novalinha
          instrução
}
```

Diretivas de Sincronização

- Uma variável pode ser do tipo private ou shared
- Uma vez que as variáveis são visíveis à todas as threads
- O acesso a elas devem acontecem de forma sincronizada e organizada
- Garantir a não aparição da condição de corrida

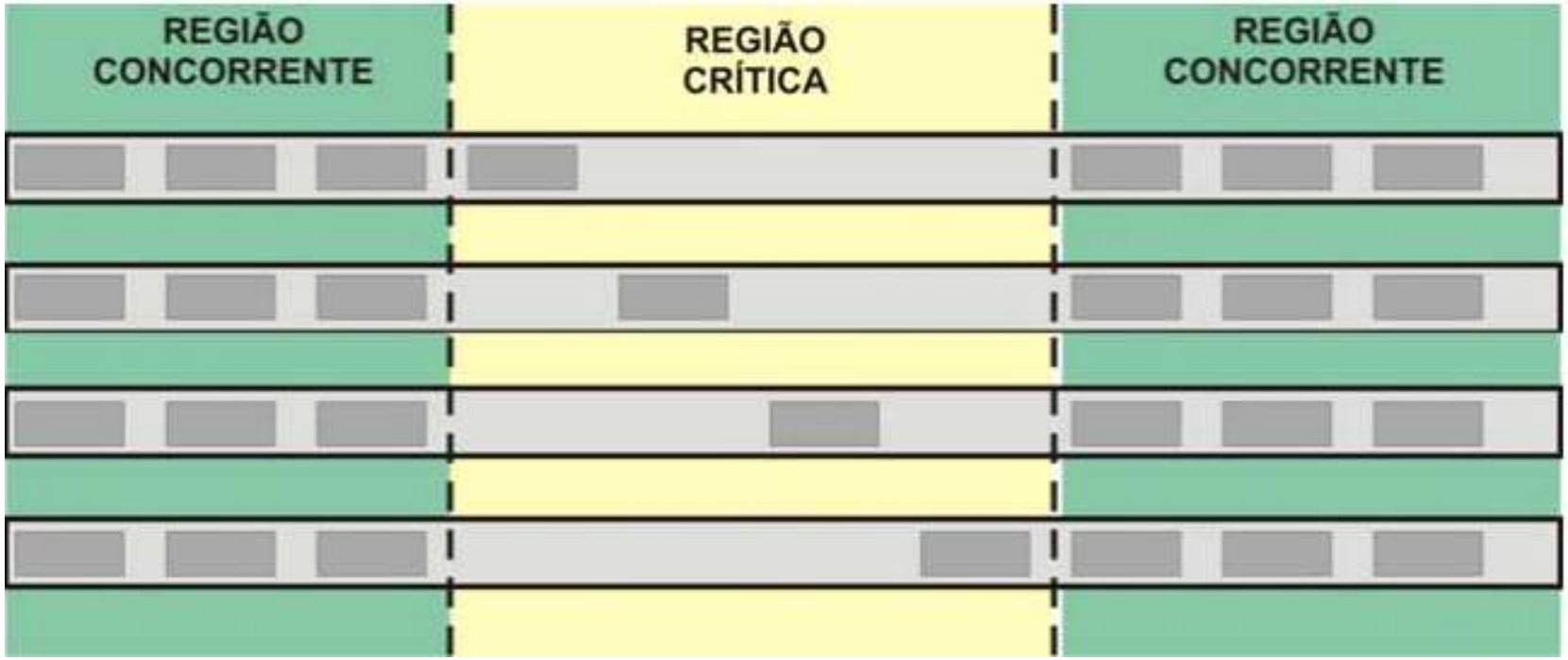
Critical

- O construtor restringe a execução de uma determinada tarefa a apenas uma thread por vez

SINTAXE:

```
#pragma omp critical [(nome)] novalinha
    Instrução
```

Funcionamento



Atomic

- As atualizações para variáveis compartilhadas não são atômicas

- O construtor atomic permite que certa região de memória seja atualizada atomicamente

Atomic

- Permite várias threads atualizarem dados compartilhados sem interferência

SINTAXE:

```
#pragma omp atomic
        Instrução
```

Critical vs Atomic

Parâmetro	Critical	Atomic
Escopo	Bloco de Instruções	Uma instrução
Instruções	Gerais	Atribuição de variáveis
Nível	Sistema Operacional	Processador
Eficiência	Menor	Maior

- Podem ser utilizados para substituir a cláusula reduction
- Entretanto sua eficiência é menor

Barrier

- Utilizada para sincronizar todas as threads em um determinado ponto do código

SINTAXE:

```
#pragma omp barrier
    novalinha
```

Barrier

- Principais utilizações é evitar condição de corrida

Flush

- É utilizada para garantir que todas as threads tenham acesso ao valor correto de uma variável compartilhada que foi recentemente atualizada
- O padrão OpenMP específica que todas as modificações de variáveis compartilhadas sejam escritas na memória principal em pontos de sincronização específicos

Flush

- Porém, nas regiões localizadas entre os pontos de sincronização não há nenhuma garantia de que as variáveis sejam atualizadas instantaneamente na memória principal

SINTAXE:

```
#pragma omp flush [(lista de variáveis)] novalinha
```

Flush

- Existe alguns flush's específicos em alguns lugares específicos das diretivas
  - Em todas as barreiras explícitas e implícitas
  - Na entrada e saída de uma região crítica
  - Na entrada e na saída de uma diretiva **ordered**
  - Na entrada e na saída de rotinas **lock**
  - Na saída de um construtor de divisão de trabalho

Ordered

- O construtor permite que um laço seja executado na ordem sequencial

SINTAXE:

```
#pragma omp ordered  novalinha
        Instrução
```

Master

- O construtor define um bloco de código que será executado apenas pela thread master

SINTAXE:

```
#pragma omp master  novalinha
    Instrução
```

threadprivate

- Específica quais variáveis serão privadas em todo o escopo do código

SINTAXE:

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

threadprivate

- Cada cópia da variável deste tipo é inicializada uma vez
- Não podem estar presentes em nenhuma lista de variáveis de cláusulas
- Exceto nas listas
  - Copyin
  - Copyprivate
  - Schedule
  - Num_threads e
  - if

threadprivate

- Pode-se garantir que o valor da cópia destas variáveis (das outras threads, exceto a mestre) irão persistir entre duas ou mais regiões paralelas consecutivas, apenas se todas as condições a seguir forem satisfeitas
  - Não houver nenhuma região paralela aninhada
  - O número de threads usado na execução de ambas as regiões paralelas for o mesmo

Cláusulas

- Definem o comportamento dos construtores aos quais estão associadas
- E das variáveis envolvidas na execução da região paralela
- Definem quais variáveis são compartilhadas entre as threads e quais são privadas

Cláusulas

- A maioria das cláusulas são aplicadas as variáveis presentes na lista de variáveis
- Uma variável pode estar presente em apenas uma cláusula por diretiva
  - Exceto as cláusulas **firstprivate** e **lastprivate**
- Nem todas as cláusulas aplicam-se em todas as diretivas

shared

- Indica quais variáveis terão endereço de memória acessível a todas as threads dentro da região paralela

SINTAXE:

```
shared (lista de variáveis)
```

shared

- As diretivas que podem utilizar essa cláusula
  - #pragma omp parallel
  - #pragma omp parallel for
  - #pragma omp parallel sections

private

- Indica que as variáveis presentes na lista são do tipo privada

SINTAXE:

```
private (lista de variáveis)
```

- Observações
  - Variável criada dentro de uma região paralela, por definição, é private
  - Por definição, a variável que registra as iterações do laço é private

private

- As diretivas que podem utilizar essa cláusula
  - #pragma omp parallel
  - #pragma omp for
  - #pragma omp section
  - #pragma omp single
  - #pragma omp parallel for
  - #pragma omp parallel sections

firstprivate

- As variáveis deste tipo entram na região paralela com os valores que possuíam antes de encontrar o construtor ao qual foi associado

SINTAXE:

```
firstprivate (lista de variáveis)
```

firstprivate

- As diretivas que podem utilizar essa cláusula
  - #pragma omp parallel
  - #pragma omp for
  - #pragma omp section
  - #pragma omp single
  - #pragma omp parallel for
  - #pragma omp parallel sections

lastprivate

- A variável sai da região definida pelo construtor com o último valor que foi atualizada

SINTAXE:

```
lastprivate (lista de variáveis)
```

lastprivate

- As diretivas nas quais essa cláusula pode ser utilizada
  - #pragma omp for
  - #pragma omp section
  - #pragma omp parallel for
  - #pragma omp parallel sections

default

- Define um padrão de compartilhamento de dados para as variáveis
- Para definir as exceções devem-se utilizar as cláusulas: private, lastprivate e firstprivate

SINTAXE:

```
default (none |shared)
```

Default(none)

- O programador deve especificar qual será o comportamento de cada uma das variáveis
- A cláusula só pode estar presente no construtor paralelo

Reduction

- Execução da cláusula reduction:
  - Ao entrar na região paralela uma cópia privada de cada variável da lista é criada em cada thread
  - As variáveis são inicializadas segundo o valor de inicialização definido pelo operador (tabela)
  - As variáveis são atualizadas localmente no interior da região paralela
  - Ao final do construtor paralelo associado as variáveis são atualizadas com a redução das cópias privadas utilizando o operador especificado.

SINTAXE:

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

Reduction

- A tabela a seguir mostra os operadores válidos e seus respectivos valores

Operador	Valor inicial
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

Reduction

- Algumas restrições
  - O tipo das variáveis presentes na lista devem ser compatíveis com o operador
  - Vetores, ponteiros e referências não podem aparecer na cláusula
  - Uma variável especificada no **reduction** não pode ser constante
  - Variáveis privadas que estão na cláusula **reduction** de uma diretiva paralela
    - Não pode estar especificada em uma cláusula **reduction** de uma diretiva de compartilhamento de trabalho inserida no construtor paralelo

Reduction

- As diretivas nas quais a cláusula pode estar presente
  - #pragma omp parallel
  - #pragma omp for
  - #pragma omp sections
  - #pragma omp parallel for
  - #pragma omp parallel sections

Copyin

- Permite que as cópias de uma variável definida como threadprivate sejam iniciadas com o mesmo valor

SINTAXE:

```
copyin (lista de variáveis)
```

Copyprivate

- É utilizada para transmitir o valor de uma variável privada de uma thread para as outras threads do grupo
- Só pode ser aplicado no construtor **single**

SINTAXE:

```
copyprivate (lista de variáveis)
```

Nowait

- Existe barreiras implícitas no final da maioria dos construtores
- Nesses casos pode-se utilizar o **nowait**
- Este faz com que as barreiras sejam ignoradas pelas threads

Nowait

- As diretivas nas quais a cláusula pode estar presente
  - #pragma omp for
  - #pragma omp sections
  - #pragma omp single
  - #pragma omp parallel for
  - #pragma omp parallel sections

SINTAXE:

```
nowait
```

Schedule

- É utilizado apenas no construtor for
- Controla a forma como as iterações são distribuídas entre as threads
- Há 4 tipos diferentes de **schedule** por padrão

SINTAXE:

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

static

- As iterações são divididas em pedaços de tamanho definido pelo segundo parâmetro
- A distribuição dos blocos de iteração entre as threads é feito de forma estática e cíclica
- Se a divisão do número de iterações não for exato, o último bloco terá um número menor de iterações
- Quando o segundo parâmetro não for especificado
  - Cada thread fica com um bloco de tamanho aproximadamente igual ao número de iterações dividido pelo número de threads

dynamic

- As iterações são distribuídas entre as threads à medida que elas solicitam mais iterações
- Cada thread executa um bloco de iterações, definida pelo segundo parâmetro
- Solicita um próximo bloco até que não existam mais blocos de iterações
- Quando o segundo parâmetro não é especificado, o padrão é 1

guided

- Indica o número mínimo de iterações a agrupar numa tarefa;
- O escalonamento dinâmico inicia com blocos de tamanho grande
- Com o passar do tempo, o tamanho do bloco diminui;
- Mas sempre maior do que o parâmetro“chunk”

runtime

- A decisão do schedule a ser executado é feito na hora da execução do código
- O tipo de schedule e o tamanho do bloco são definidos pela variável de ambiente
- OMP_SCHEDULE

Clausula	Quando usar
STATIC	Trabalho similar por iteração
DYNAMIC	Imprevisível, grande variação por iteração
GUIDED	Caso especial de dinâmico para evitar overhead de agendamento
RUNTIME	Usa a variável de ambiente OMP_SCHEDULE

Exemplo

```
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
    {
       if ( TestForPrime(i) )  gPrimesFound++;
    }
```

- Iterações são divididas em pedações de tamanho 8
- Se start = 3, então o primeiro pedaço é
- i= {3, 5, 7, 9, 11, 13, 15, 17}

IF

- Necessita avaliar como verdadeiro para que o team de threads seja criado
- Caso contrário, a execução será sequencial

Resumo

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
COPYPRIVATE				●		
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Diretivas que não aceitam Cláusulas

- Master
- Critical
- Barrier
- Atomic
- Flush
- Ordered
- Threadprivate

Funções de interface

- Disponibiliza 3 grupos de funções
  - Ambiente de execução
  - Sincronização do acesso aos dados
  - Medida de tempo das aplicações

Funções do ambiente de execução

- Omp_set_num_threads()
- Omp_get_num_threads()
- Omp_get_max_threads()
- Omp_get_thread_num()
- Omp_get_thread_limit()
- Omp_get_num_procs()
- Omp_in_parallel

Funções do ambiente de execução

- Omp_set_dynamic()
- Omp_get_dynamic()
- Omp_set_nested()
- Omp_get_nested()
- Omp_set_schedule()
- Omp_get_schedule()

Funções do ambiente de execução

- Omp_set_max_active_levels
- Omp_set_max_active_levels
- Omp_get_level
- Omp_get_ancestor_thread_num
- Omp_get_team_size
- Omp_in_final

Funções de Bloqueio

- São utilizadas para promover sincronização entre as threads
- Essas rotinas operam sobre variáveis de travamento do OpenMP
  - Denominadas de **lock**
- São do tipo
  - Omp_lock_t
  - Omp_nest_lock_t

Funções de Bloqueio

- Uma lock pode estar nos estados
  - Não inicializada
  - Desbloqueada
  - Bloqueada

Funções de Bloqueio

- Há dois tipos de lock
  - Lock simples
  - Lock aninhado
- As rotinas de bloqueio do OpenMP acessam os locks de tal forma que eles sempre lêem e atualizam o valor mais atualizado do lock

Funções de Bloqueio

- Omp_init_lock() e omp_init_nest_lock()
- Omp_destroy_lock() e omp_destroy_nested_lock()

SINTAXE:

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

SINTAXE:

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nested_lock(omp_nested_lock_t *lock);
```

Funções de Bloqueio

- omp_set_lock()
- omp_set_nested_lock()

SINTAXE:

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nested_lock(omp_nested_lock_t *lock);
```

Funções de Bloqueio

- Omp_unset_lock() e omp_unset_nested_lock()
  - Desbloqueiam um lock simples e decrementa o contador **nesting** de um lock aninhado
  - Quando o contador for igual à 0, então o lock aninhado é desbloqueado

Funções de Bloqueio

- Omp_test_lock() e omp_test_nested_lock()
  - Idem a rotina omp_set_lock()
  - Porém elas não bloqueiam a thread que está executando a rotina
  - Para um lock simples, a rotina retorna verdadeiro se o lock foi bloqueado com sucesso
  - Para o lock aninhado, retorna o novo contador **nesting** se o lock foi bloqueado com sucesso

Observação sobre Lock

- Sempre preferir os mecanismos abstratos do OPENMP
- Ao invés da utilização das funções de mais baixo nível disponibilizada pela API
- Pois implicam em overheads

TASK

- Outro forma de definir blocos que podem ser executados em paralelo
  - Omp task
- Pode utilizar, quando necessário, uma diretiva para bloquear a execução da aplicação até que a tarefa tenha sido concluída
  - taskwait

Funções de Tempo

- Omp_get_wtime()

```
#include <omp.h>
double start;
double end;
start = omp_get_wtime(void);
... work to be timed ...
end = omp_get_wtime(void);
printf("Work took %f sec. time \n", end-start);
```

Funções de Tempo

- Omp_get_wtick()

SINTAXE:

```
double omp_get_wtick(void);
```

Variáveis de Ambiente

- As variáveis de ambiente são
  - OMP_SCHEDULE
  - OMP_NUM_THREADS
  - OMP_DYNAMIC
  - OMP_NESTED
  - OMP_STACKSIZE
  - OMP_THREAD_LIMIT

OMP_SCHEDULE

- Especifica qual será o tipo do schedule e o tamanho do bloco (trechos de execução) que será utilizado pelos construtores **for** e **parallel for**
- Caso seja especificado nesses construtores essa variável será ignorada (cláusula schedule_runtime)
- Tamanho do bloco: int-> positivo e opcional

SINTAXE:

CSH:

```
setenv OMP_SCHEDULE "static"
setenv OMP_SCHEDULE "dynamic,6"
```

BASH

```
export OMP_SCHEDULE = "static"
export OMP_SCHEDULE = "dynamic,6"
```

OMP_NUM_THREADS

- Define o número de threads que será utilizado durante a execução da região paralela
- Deve ser sempre um valor inteiro positivo

SINTAXE:

CSH:

```
setenv OMP_NUM_THREADS 16
```

BASH:

```
export OMP_NUM_THREADS = 16
```

OMP_DYNAMIC

- Habilita ou desabilita o ajuste dinâmico do número de threads disponíveis para a execução da região paralela
- O valor desta variável deve ser true ou false
- Porém o ajuste pode ser feito em tempo de execução
  - Omp_set_dynamic()

SINTAXE:

CSH:

```
setenv OMP_DYNAMIC TRUE
```

BASH:

```
export OMP_DYNAMIC = TRUE
```

OMP_NESTED

- Habilita ou desabilita o paralelismo aninhado
- A menos que o mesmo seja habilitado ou desabilitado pela chamada da função
  - omp_set_nested()
- TRUE habilita
- FALSE desabilita

SINTAXE:

CSH:

```
setenv OMP_NESTED TRUE
```

BASH

```
export OMP_NESTED = TRUE
```

Integração Numérica – Fórmula dos Trapézios

- A integral de uma função f(x) pode ser calculada:

$$\int_a^b f(x)dx = \sum_{i=1}^{n} A_i = h\left[\frac{f(x_0)}{2} + \frac{f(x_n)}{2} + f(x_1) + f(x_2) + \ldots + f(x_{n-1})\right]$$

- Faça um código paralelo para computar a integral de $f(x) = x^2 + 1$ no intervalo [0,1]

Integração Numérica – Fórmula dos Trapézios