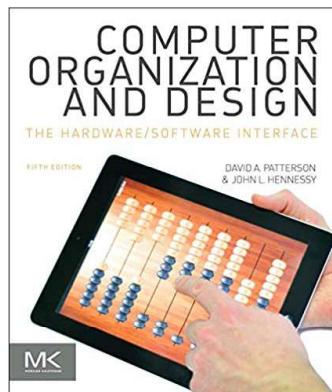


Aula 13 - Programação Paralela

Prof. Dr. Clodoaldo A. de Moraes Lima

Material baseado no livro “Patterson, David A., Hennessy, J. L. - Computer Organization And Design: The Hardware/Software Interface”



Introdução

- O desenvolvimento científico tem exigido das simulações numéricas resultados cada vez mais confiáveis e próximos da realidade.
- A computação científica se depara com o desafio de superar as diversas barreiras que surgem em virtude da limitação física dos computadores.

Computação de Alto Desempenho → Processamento Númerico → Armazenamento de dados → Visualização

Computação de Alto Desempenho

- É uma disciplina que tem como objetivo aumentar a velocidade de processamento de simulações numéricas e viabilizar a simulação de modelos muito grandes e/ou com elevado grau de detalhes



O processamento paralelo é uma das técnicas utilizadas na computação científica de alto desempenho

Memória Distribuída

- Sistema Computacional constituído de vários processadores dotados de recursos individuais
- Esse sistema se caracteriza pela troca de mensagem entre os processadores
- A troca de mensagem é feita por uma biblioteca de comunicação (MPI)

Modelo de programação baseado em MPI

Memória Compartilhada

- Sistema Computacional constituído de vários processadores que compartilham o mesmo recurso de memória
- É necessária a sincronização do acesso aos dados na memória compartilhada pelos processadores

OpenMP, Win32 Threads, Posix Threads

Arquiteturas UMA e NUMA

- UMA
 - Nas máquinas que possuem esse tipo de arquitetura, o tempo de acesso aos dados localizados em qualquer posição da memória é o mesmo para todos os processadores.
- NUMA
 - Nos computadores que possuem arquitetura NUMA, o tempo de acesso à memória depende da posição em que a mesma se encontra em relação ao processador.

Técnicas para programação modelos de memória compartilhada

- Programas construídos com esse modelo de programação requerem o uso de técnicas que possibilitem a criação de threads e que implementem mecanismos de sincronização entre elas.
- As ferramentas mais comuns para ambientes de memória compartilhada são baseadas em:
 - Threading Explícito
 - Diretivas de compilação
 - Troca de mensagem
 - Linguagens paralelas

Técnicas para programação modelos de memória compartilhada

- Threading Explicito
 - O programador cria explicitamente múltiplas threads dentro de um mesmo processo e divide também explicitamente o trabalho a ser realizado.
 - Posix Threads, Win 32 Threads
- Diretivas de compilação
 - O programador utiliza diretivas de compilação, que são inseridas no código sequencial, para informar ao compilador as regiões que devem ser paralelizadas
 - OpenMP

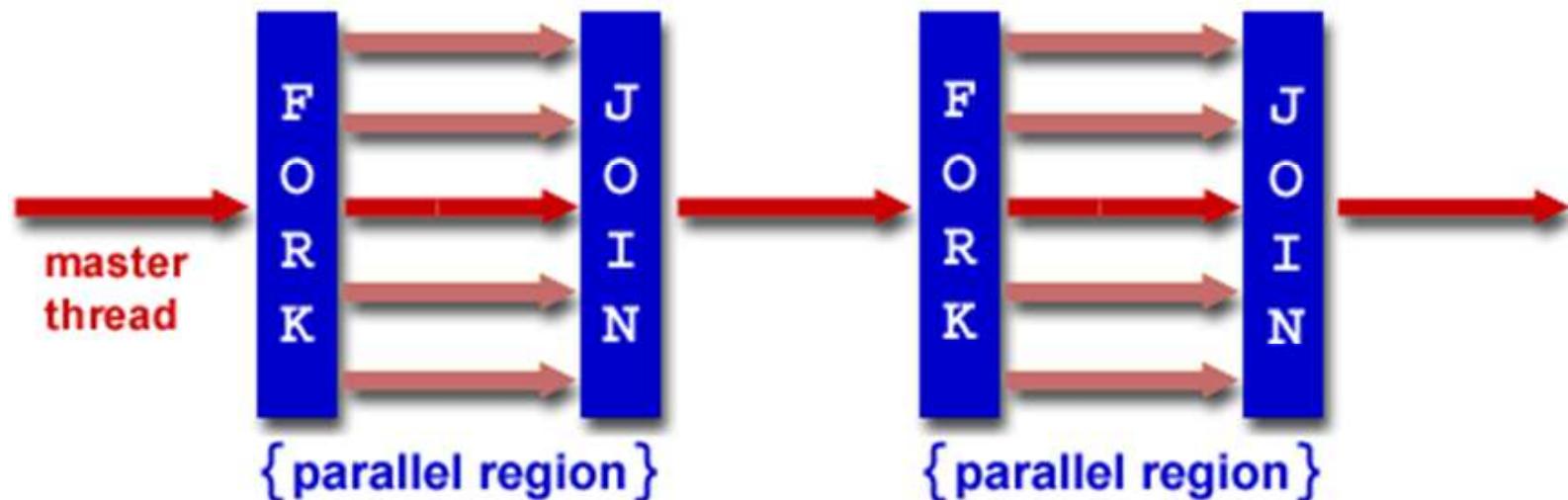
Técnicas para programação modelos de memória compartilhada

- Troca de Mensagem
 - A comunicação via troca de mensagem também é utilizada como técnica para programação em ambientes de memória compartilhada
 - MPI
- Linguagens Paralelas
 - São linguagens criadas especificamente para esse modelo de programação
 - HPF - High Performance Fortran

- Desenvolvido e mantido pelo grupo OpenMP ARB (Architecture Review Board)
- Teve início por volta de 1997 (Open + MP = padrão aberto + Multi Processing)
- Consiste em uma API e um conjunto de diretivas que permite a criação de programas paralelos com compartilhamento de memória, através da criação automática e otimizada de um conjunto de threads.
- Se encontra na versão 4.0 e está disponível nas linguagens Fortran 77, Fortran 90, C e C++
- OpenMP não é uma linguagem de programação, é um padrão que define como os compiladores devem gerar códigos paralelos de diretivas e funções

Modelo de Programação OpenMP

- Baseia-se na criação de várias threads (linhas de execução que compartilhamo mesmo recurso de memória)
- Modelo de programação fork-join: várias threads são criadas num dado ponto do código (fork) e, depois, num outro ponto, essas threads, exceto a thread inicial, deixam de existir (join)



Primeiro Código

```
#include <stdio.h>
#include <omp.h>
int main( int argc , char *argv[] ) {
    // Imprime a thread principal
    printf( "\nOla_1__Fora_da_Regiao_Paralela_...\n\n" )

    // Cria a regiao paralela
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    printf("Sou_a_thread_%d_de_um_total_%d\n", id, nt);
}
printf("\nOla_1__Fora_da_Regiao_Paralela_...\n\n");
return 0;
}
```

pragma omp parallel

- Diretiva paralela mais básica.
- O número de threads que executam o bloco que segue o pragma é determinado pelo sistema de runtime.

Execução

```
gcc -fopenmp teste.c -o teste  
./teste
```

Ola 1 – Fora da Regiao Paralela ...

```
Sou a thread 1 de um total de 8  
Sou a thread 2 de um total de 8  
Sou a thread 3 de um total de 8  
Sou a thread 4 de um total de 8  
Sou a thread 5 de um total de 8  
Sou a thread 6 de um total de 8  
Sou a thread 7 de um total de 8  
Sou a thread 0 de um total de 8
```

Ola 2 – Fora da Regiao Paralela ...

Nova Execução

```
gcc -fopenmp teste.c -o teste  
./teste
```

Ola 1 – Fora da Regiao Paralela ...

```
Sou a thread 1 de um total de 8  
Sou a thread 2 de um total de 8  
Sou a thread 3 de um total de 8  
Sou a thread 4 de um total de 8  
Sou a thread 6 de um total de 8  
Sou a thread 7 de um total de 8  
Sou a thread 5 de um total de 8  
Sou a thread 0 de um total de 8
```

Ola 2 – Fora da Regiao Paralela ...

Nova Execução

```
export OMP_NUM_THREADS=4  
./teste
```

Ola 1 – Fora da Regiao Paralela ...

Sou a thread 3 de um total de 4
Sou a thread 0 de um total de 4
Sou a thread 1 de um total de 4
Sou a thread 2 de um total de 4

Ola 2 – Fora da Regiao Paralela ...

Nova Execução

```
export OMP_NUM_THREADS=2  
./teste
```

Ola 1 – Fora da Regiao Paralela ...

Sou a thread 0 de um total de 2

Sou a thread 1 de um total de 2

Ola 2 – Fora da Regiao Paralela ...

Segundo Código

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    // Imprime a thread principal
    printf("\nOla_1_-Fora_da_Regiao_Paralela_...\n\n")

    omp_set_num_threads(8);

    //Cria a regiao paralela
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    printf("Sou_a_thread_%d_de_um_total_%d\n", id, nt);
}
printf("\nOla_2_-Fora_da_Regiao_Paralela_...\n\n");

    //Cria a regiao paralela
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    printf("Sou_a_thread_%d_de_um_total_%d\n", id, nt);
}

    return 0;
}
```

Nova Execução

```
export OMP_NUM_THREADS=16
gcc -fopenmp teste.c -o teste
./teste
```

Ola 1 – Fora da Regiao Paralela ...

```
Sou a thread 1 de um total de 8
Sou a thread 4 de um total de 8
Sou a thread 5 de um total de 8
Sou a thread 6 de um total de 8
Sou a thread 2 de um total de 8
Sou a thread 0 de um total de 8
Sou a thread 3 de um total de 8
Sou a thread 7 de um total de 8
```

Ola 2 – Fora da Regiao Paralela ...

```
Sou a thread 1 de um total de 8
Sou a thread 4 de um total de 8
Sou a thread 3 de um total de 8
Sou a thread 6 de um total de 8
Sou a thread 5 de um total de 8
Sou a thread 2 de um total de 8
Sou a thread 7 de um total de 8
Sou a thread 0 de um total de 8
```

Segundo Código

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    // Imprime a thread principal
    printf("\nOla_1_-Fora_da_Regiao_Paralela_...\n\n")

    omp_set_num_threads(8);

    //Cria a regiao paralela
#pragma omp parallel num_threads(4)
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    printf("Sou_a_thread_%d_de_um_total_%d\n", id, nt);
}
printf("\nOla_2_-Fora_da_Regiao_Paralela_...\n\n");

    //Cria a regiao paralela
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    printf("Sou_a_thread_%d_de_um_total_%d\n", id, nt);
}

    return 0;
}
```

Nova Execução

```
gcc -fopenmp teste.c -o teste  
./teste
```

Ola 1 – Fora da Regiao Paralela ...

```
Sou a thread 0 de um total de 4  
Sou a thread 1 de um total de 4  
Sou a thread 3 de um total de 4  
Sou a thread 2 de um total de 4
```

Ola 2 – Fora da Regiao Paralela ...

```
Sou a thread 1 de um total de 8  
Sou a thread 2 de um total de 8  
Sou a thread 3 de um total de 8  
Sou a thread 4 de um total de 8  
Sou a thread 5 de um total de 8  
Sou a thread 6 de um total de 8  
Sou a thread 0 de um total de 8  
Sou a thread 7 de um total de 8
```

Segundo Código

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    // Imprime a thread principal
    printf("\nOla_1__Fora_da_Regiao_Paralela_...\n\n")

    //Cria a regiao paralela
#pragma omp parallel num_threads(4)
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    printf("Sou_a_thread_%d_de_um_total_%d\n", id, nt);
}
printf("\nOla_2__Fora_da_Regiao_Paralela_...\n\n");

    //Cria a regiao paralela
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    printf("Sou_a_thread_%d_de_um_total_%d\n", id, nt);
}

    return 0;
}
```

Nova Execução

```
gcc -fopenmp teste.c -o teste  
./teste
```

Ola 1 – Fora da Regiao Paralela ...

```
Sou a thread 0 de um total de 4  
Sou a thread 1 de um total de 4  
Sou a thread 3 de um total de 4  
Sou a thread 2 de um total de 4
```

Ola 2 – Fora da Regiao Paralela ...

```
Sou a thread 1 de um total de 8  
Sou a thread 2 de um total de 8  
Sou a thread 3 de um total de 8  
Sou a thread 4 de um total de 8  
Sou a thread 5 de um total de 8  
Sou a thread 6 de um total de 8  
Sou a thread 0 de um total de 8  
Sou a thread 7 de um total de 8
```

Nova Execução

```
export OMP_NUM_THREADS=16
./teste
Ola 1 - Fora da Regiao Paralela ...

Sou a thread 1 de um total de 4
Sou a thread 0 de um total de 4
Sou a thread 2 de um total de 4
Sou a thread 3 de um total de 4

Ola 2 - Fora da Regiao Paralela ...

Sou a thread 1 de um total de 16
Sou a thread 5 de um total de 16
Sou a thread 2 de um total de 16
Sou a thread 6 de um total de 16
Sou a thread 10 de um total de 16
Sou a thread 4 de um total de 16
Sou a thread 12 de um total de 16
Sou a thread 3 de um total de 16
Sou a thread 7 de um total de 16
Sou a thread 11 de um total de 16
Sou a thread 13 de um total de 16
Sou a thread 8 de um total de 16
Sou a thread 9 de um total de 16
Sou a thread 14 de um total de 16
Sou a thread 0 de um total de 16
Sou a thread 15 de um total de 16
```

Nova Execução

```
export OMP_NUM_THREADS=2
```

```
./teste
```

```
Ola 1 - Fora da Regiao Paralela ...
```

```
Sou a thread 2 de um total de 4
```

```
Sou a thread 1 de um total de 4
```

```
Sou a thread 0 de um total de 4
```

```
Sou a thread 3 de um total de 4
```

```
Ola 2 - Fora da Regiao Paralela ...
```

```
Sou a thread 1 de um total de 2
```

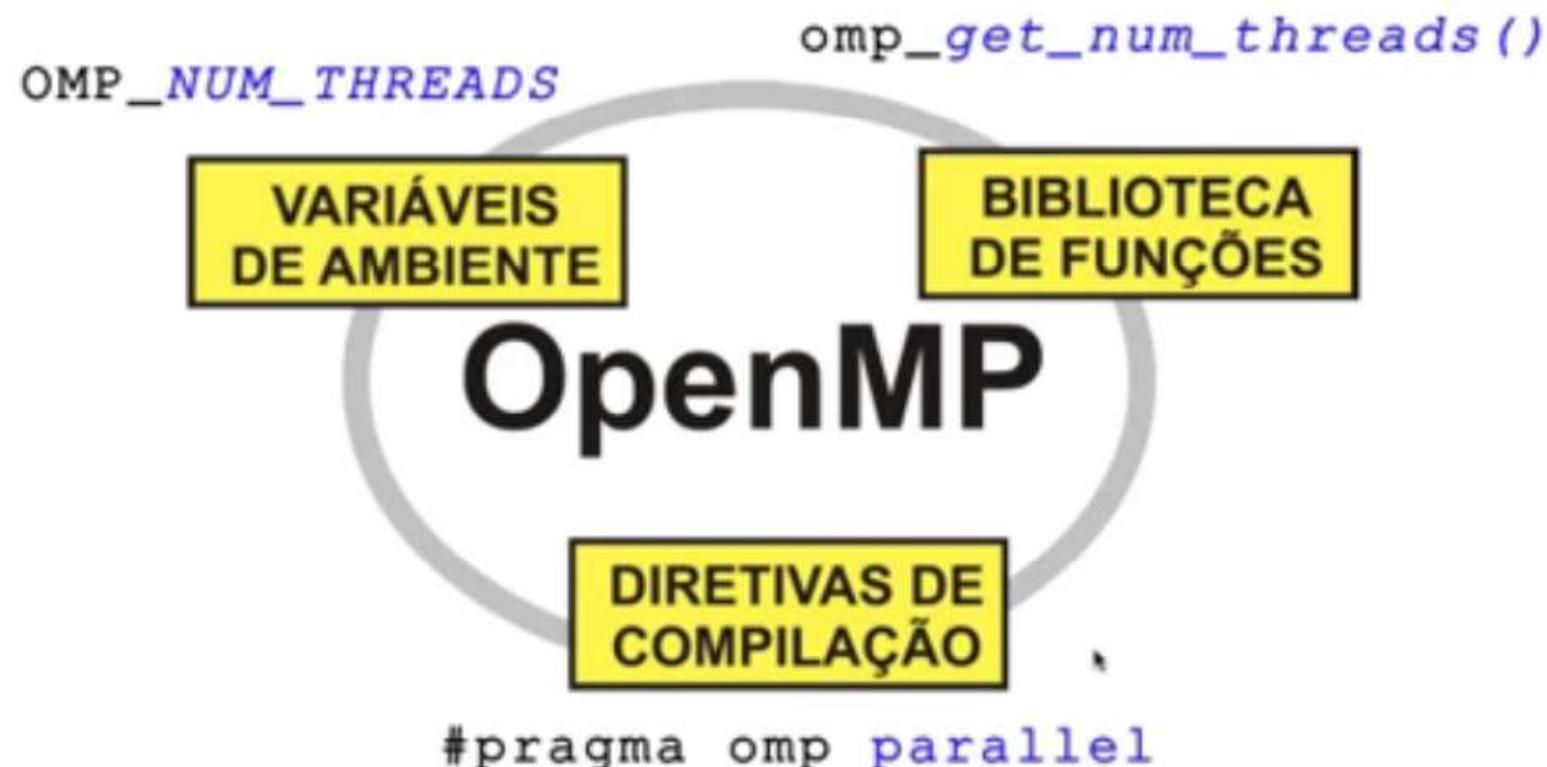
```
Sou a thread 0 de um total de 2
```

Cláusula

- Texto que modifica uma diretiva.
- A cláusula num-threads pode ser adicionada a uma diretiva paralela.
- Permite o programador especificar o número de threads que devem executar no bloco que segue o pragma.

```
# pragma omp parallel num-threads(4)
```

Elementos que constituem o OpenMP



Por que usar OpenMP?

- Facilidade de conversão de programas sequenciais em paralelos
- Maneira simples de explorar o paralelismo
- Fácil compreensão e uso de diretivas
- Minimiza a interferência na estrutura do algoritmo
- Compila e executa em ambientes paralelo e sequencial

OpenMP e Tecnologia MultiCore

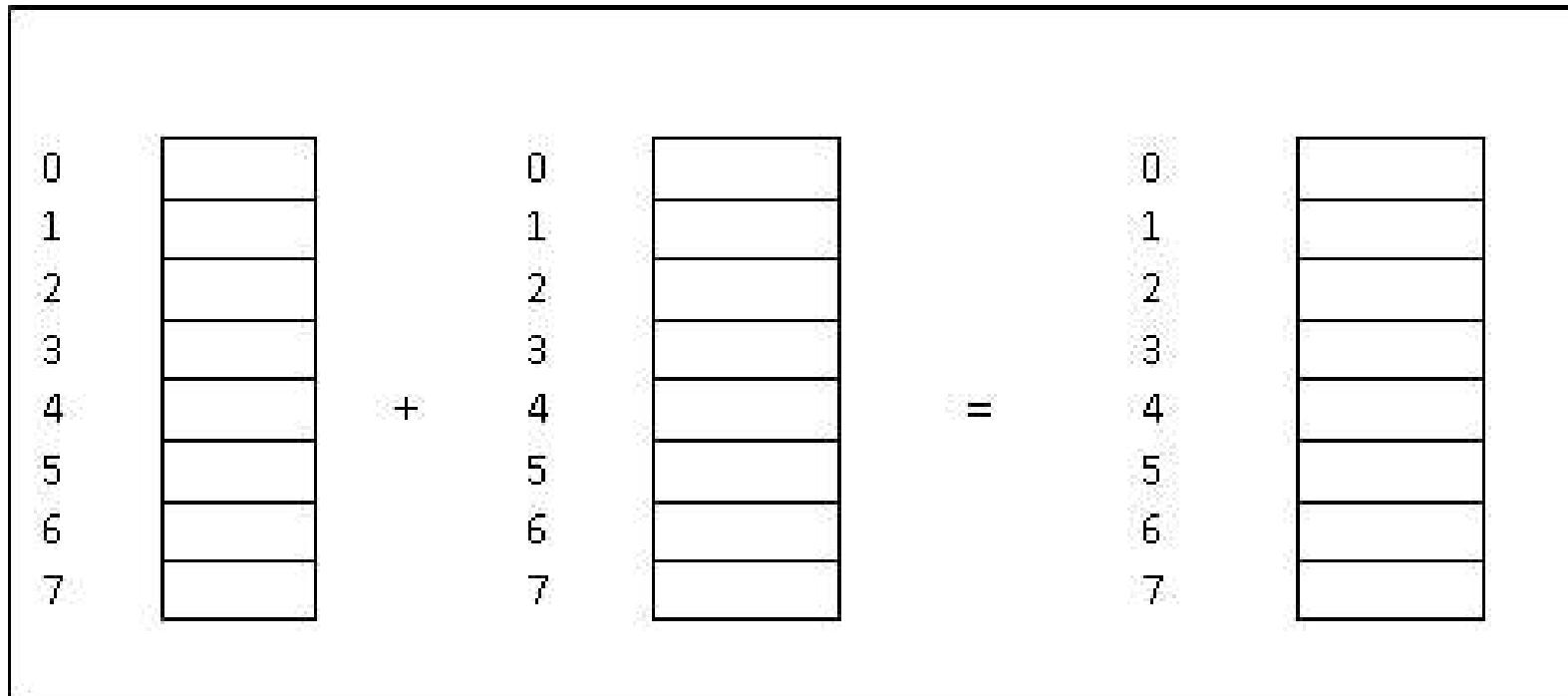
Para uma total utilização do poder processamento oferecido pela tecnologia Multicore, as aplicações devem ser escritas de modo a usar intensivamente o conceito de threads.

Exercícios

Soma de vetores

Soma de matrizes

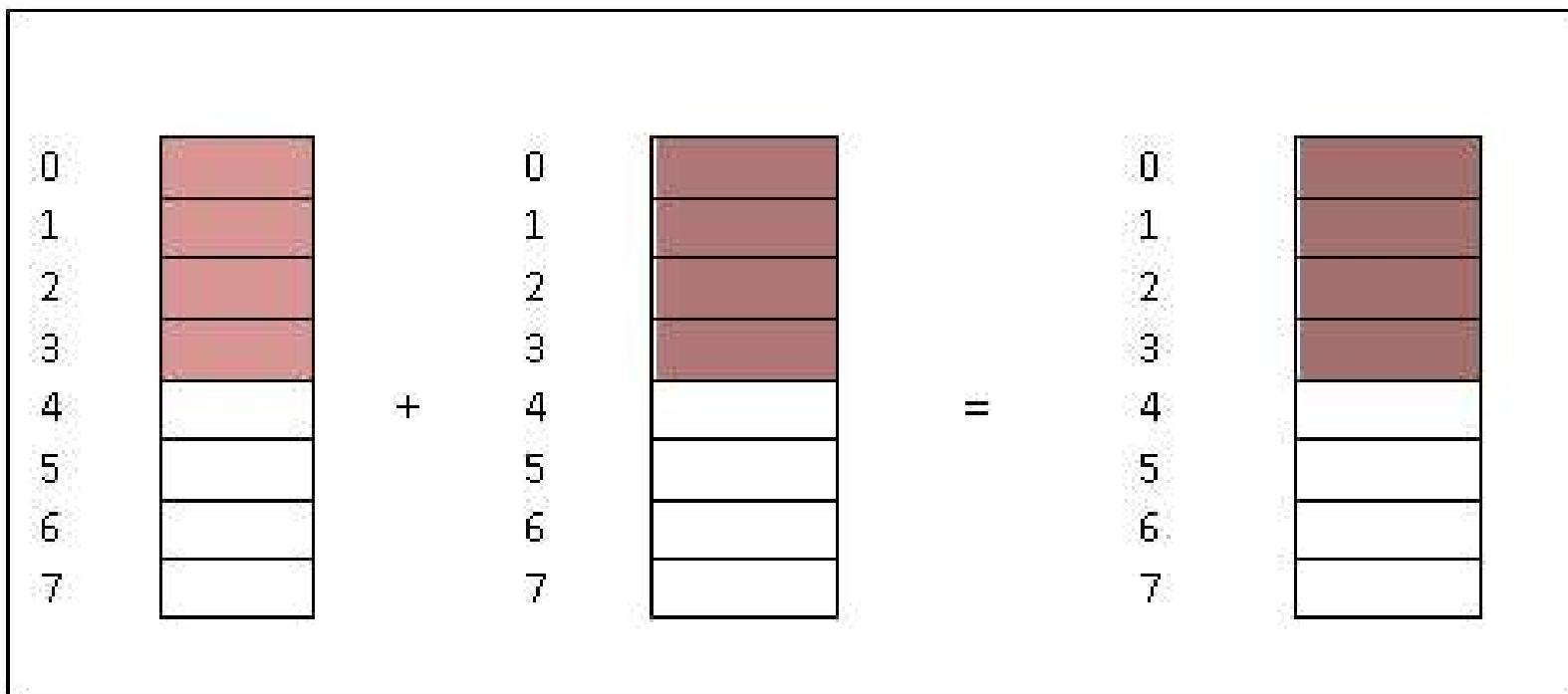
Soma de Matrizes



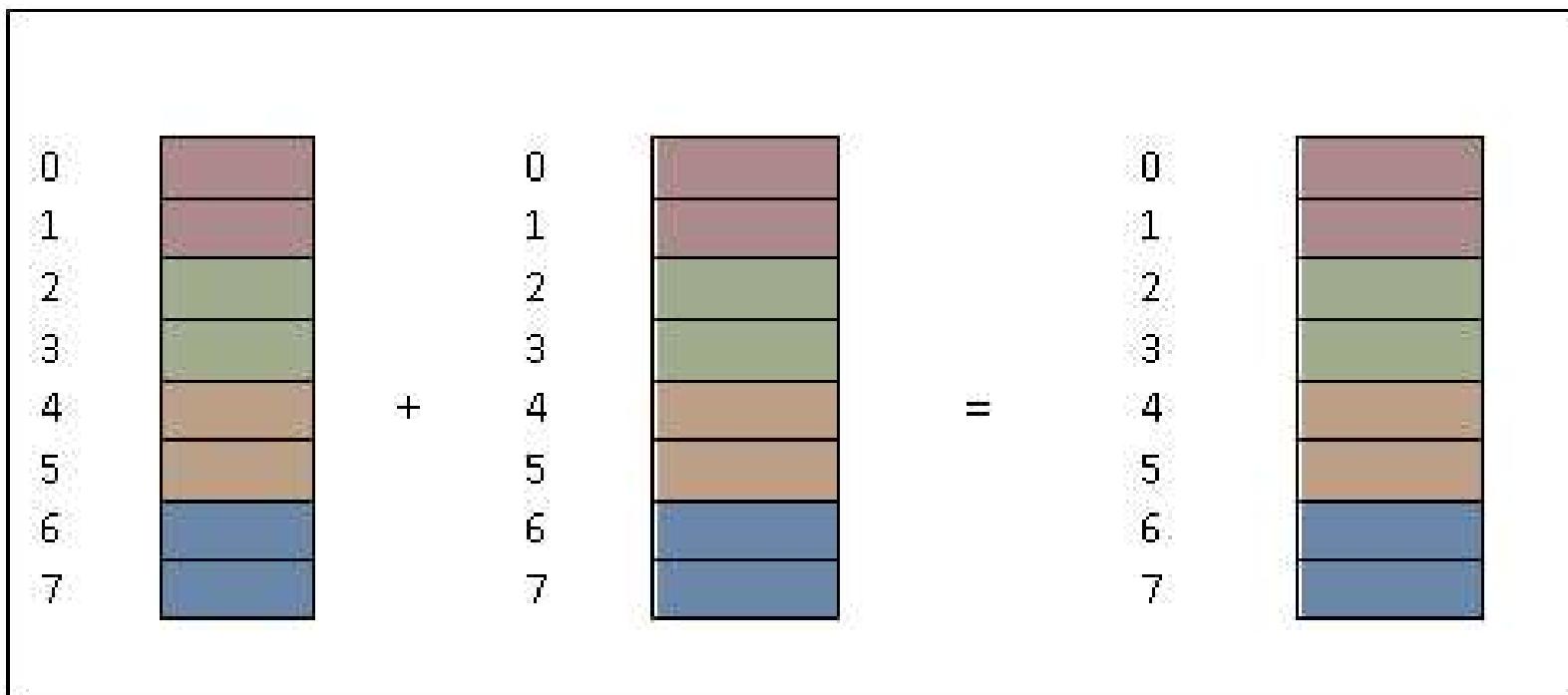
Segundo Código

```
#include <stdio.h>
#include <omp.h>
int main (int argc , char *argv [])
{
    float A[2048];
    float B[2048];
    float C[2048];
    int i;
    for (i=0;i<2048;i++)
    {
        A[ i ]=2;
        B[ i ]=3;
    }
    for ( i=0;i<2048;i++)
    {
        C[ i ] = A[ i]+B[ i ];
    }
    for ( i=0;i<2048;i++)
    {
        printf("C[%d] ==%f\n" ,i , C[ i ]);
    }
    return 0;
}
```

Soma de vetores em OpenMP - 2 threads

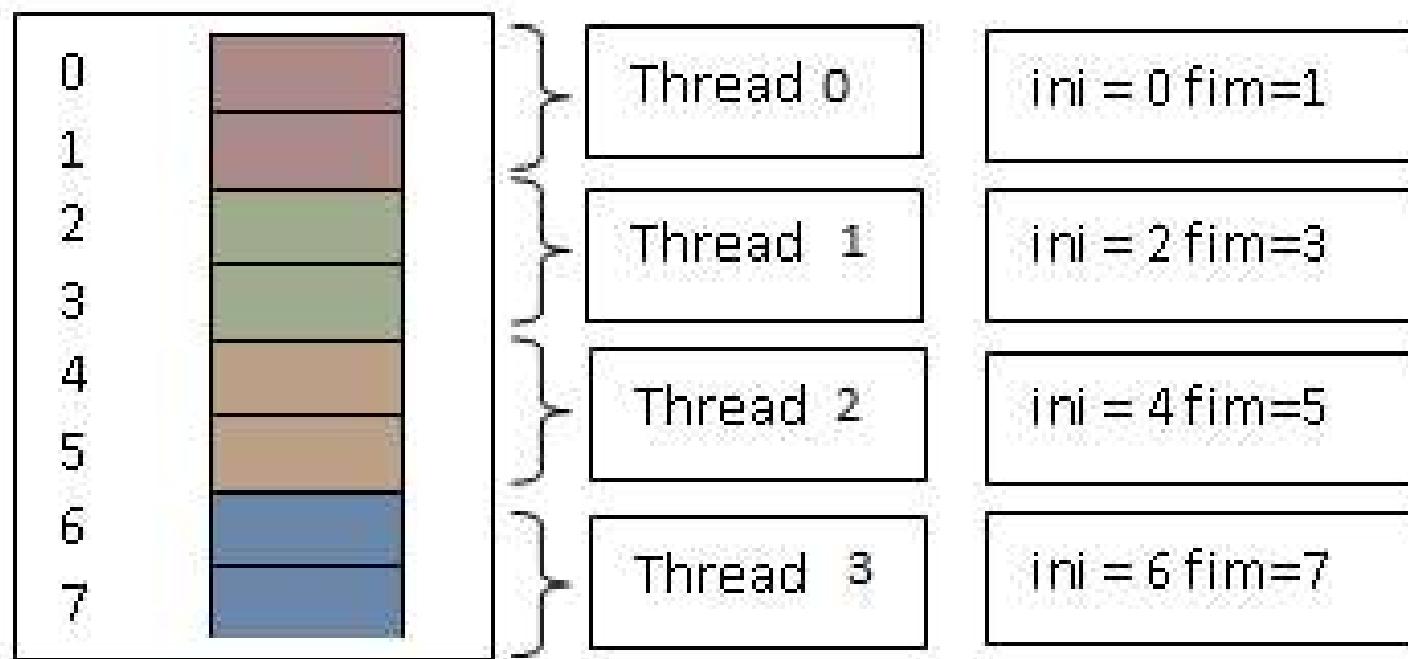


Soma de vetores em OpenMP - 4 threads



Vetor de 8 posições

```
id = omp-get-thread-num();  
nt = omp-get-num-threads();  
size= 8/nt;  
ini = size*id;  
fim= ini + size - 1;
```



Soma de Vetor

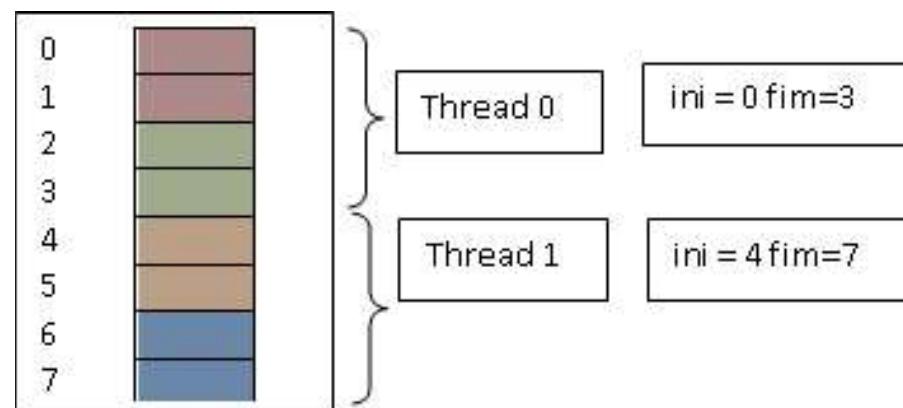
```
#include <stdio.h>
#include <omp.h>
int main ( int argc , char *argv [] )
float A[2048];
float B[2048];
float C[2048];
int i ;
for ( i=0;i<2048;i++)
{
    A[ i ]=2;
    B[ i ]=3;
}
#pragma omp parallel
{
    int id = omp_get_thread_num ();
    int nt = omp_get_num_threads ();
    int size = (int) 2048/nt;
    int ini == id *size;
    int fim = ini + size -1;
    int i ;
    for ( i=ini ;i<=fim ;i++)
    {
        C[ i ] = A[ i ]+B[ i ];
    }
}
for ( i=0;i<2048;i++)
{
    printf("C[%d] =%f\n" ,i , C[ i ]);
}
return 0;
}
```

Vetor de 8 posições

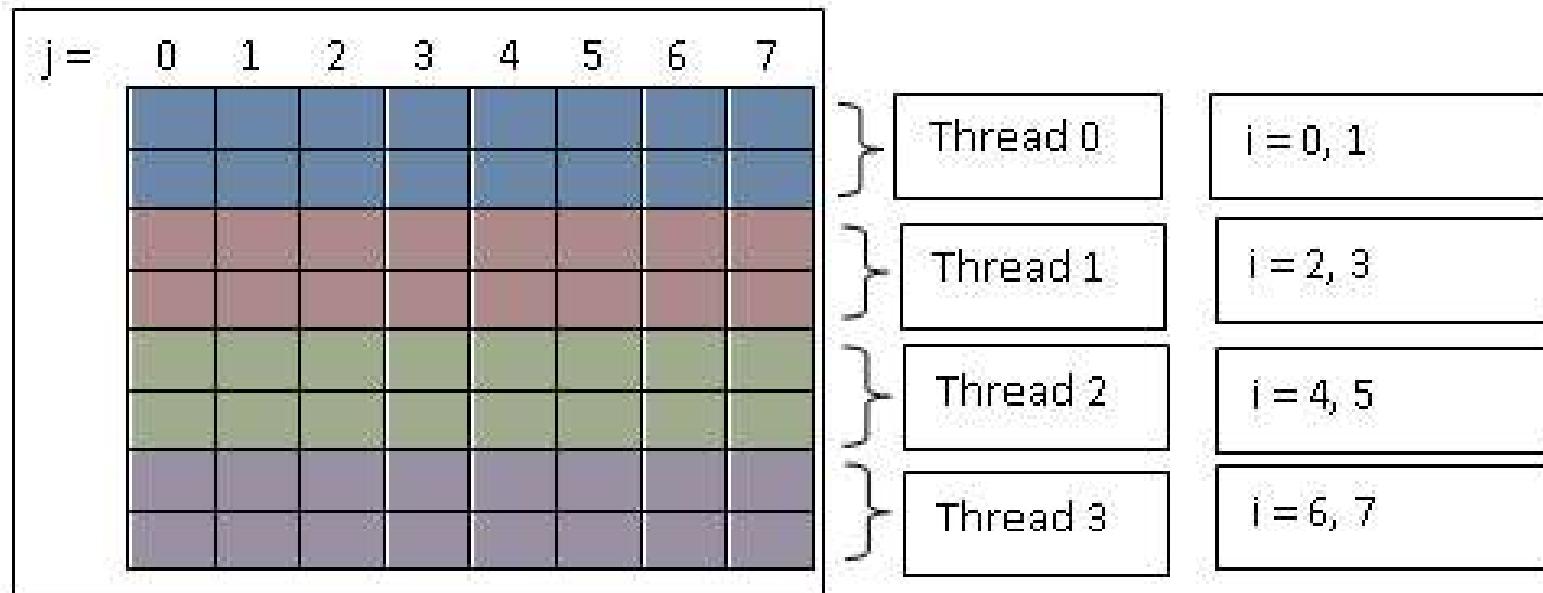
Por que funciona para qualquer número de threads

- Se modificar a variável de ambiente, o comportamento do programa em termos de resultados final não muda

```
id = omp-get-thread-num();
nt = omp-get-num-threads();
size= (int) 8/nt;
ini = size*id;
fim= ini + size - 1;
```



Soma de Matrizes - Particionamento 1D



Soma de Matrizes

```
#include <stdio.h>
#include <omp.h>
int main ( int argc , char *argv [] )
{
    float A[16][16];
    float B[16][16];
    float C[16][16];
    int i , j ;
    for ( i=0;i<16;i++)
    {
        for ( j=0;j<16;j++)
        { A[ i ][ j ]=2;
          B[ i ][ j ]=3;}
    }
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int nt = omp_get_num_threads();
        int size = (int) 16/nt;
        int ini == id *size;
        int fim = ini + size -1;
        int i,j;
        for ( i=ini ;i<=fim ; i++)
        {   for ( j=0;j<16;j++)
            C[ i ][ j ] = A[ i ][ j ]+B[ i ][ j ];
        }
    }
    for ( i=0;i<16;i++)
    {   for ( j=0;j<16;j++)
        printf("C[%d][%d] = %f\n" , i , j , C[ i ][ j ]);
    }
    return 0;
}
```

Usando uma interface mais amigável - Soma de Vetor

```
#include <stdio.h>
#include <omp.h>
int main ( int argc , char *argv [] )
{
    float A[2048];
    float B[2048];
    float C[2048];
    int i ;
    for ( i=0;i<2048;i++)
    {
        A[ i]=2;
        B[ i]=3;
    }
#pragma omp parallel
{
    int id = omp_get_thread_num ();
    int nt = omp_get_num_threads ();
    int i ;
#pragma omp for
    for ( i=0;i<2048;i++)
    {
        printf("A_thread_%d_somou_a_posicao_%d\n" ,id , i );
        C[ i ] = A[ i ]+B[ i ];
    }
    for ( i=0;i<2048;i++)
    {
        printf("C[%d] ==%f\n" ,i , C[ i ]);
    }
    return 0;
}
```

Usando uma interface mais amigável - Soma de Matriz

```
#include <stdio.h>
#include <omp.h>
int main ( int argc , char *argv [] )
{
    float A[16][16];
    float B[16][16];
    float C[16][16];
    int i , j ;
    for ( i=0;i<16;i++)
    {
        for ( j=0;j<16;j++)
        { A[ i ][ j ]=2;
          B[ i ][ j ]=3;}
    }
    #pragma omp parallel
    {
        int i,j;
        #pragma omp for
        for ( i=0;i<16;i++)
        {   for ( j=0;j<16;j++) //Nao ser paralelizado
            C[ i ][ j ] = A[ i ][ j ]+B[ i ][ j ];
        }
    }
    for ( i=0;i<16;i++)
    {   for ( j=0;j<16;j++)
        printf("C[%d][%d] = %f\n" , i , j , C[ i ][ j ]);
    }
    return 0;
}
```

Condições de Corrida

- Variáveis privadas
- Variáveis compartilhadas

Código visto anteriormente

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    printf("\nOla_1__Fora_da_Regiao_Paralela_...\n\n")

#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    printf("Sou_a_thread_%d_de_um_total_%d\n", id, nt);
}
printf("\nOla_2__Fora_da_Regiao_Paralela_...\n\n");

return 0;
}
```

Nova Execução

```
./teste
```

```
Ola 1 - Fora da Regiao Paralela ...
```

```
Sou a thread 1 de um total de 8
```

```
Sou a thread 5 de um total de 8
```

```
Sou a thread 6 de um total de 8
```

```
Sou a thread 0 de um total de 8
```

```
Sou a thread 4 de um total de 8
```

```
Sou a thread 3 de um total de 8
```

```
Sou a thread 2 de um total de 8
```

```
Sou a thread 7 de um total de 8
```

```
Ola 2 - Fora da Regiao Paralela ...
```

E agora, o que irá acontecer?

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    printf("\nOla_1__Fora_da_Regiao_Paralela...\n\n")
    int id, nt;
#pragma omp parallel
{
    id = omp_get_thread_num(); //definicao fora da regiao paralela
    nt = omp_get_num_threads(); //definicao fora da regiao paralela
    printf("Sou_a_thread_%d_de_um_total_%d\n", id, nt);
}
printf("\nOla_2__Fora_da_Regiao_Paralela...\n\n");
return 0;
}
```

Nova Execução

Para visualizar o problema, o programa deve ser executado várias vezes

```
./teste
```

Ola 1 – Fora da Regiao Paralela ...

```
Sou a thread 2 de um total de 8
Sou a thread 2 de um total de 8
Sou a thread 6 de um total de 8
Sou a thread 5 de um total de 8
Sou a thread 0 de um total de 8
Sou a thread 4 de um total de 8
Sou a thread 1 de um total de 8
Sou a thread 7 de um total de 8
```

Ola 2 – Fora da Regiao Paralela ...

Analisando o resultado

- Não faz sentido que duas threads distintas escrevam o mesmo *id*
- Algo estranho, a identificação das threads deve ser única
- O que isto significa?

Região Crítica

Trecho de código que acessa variáveis compatilhadas entre processos ou threads

Condição de Corrida

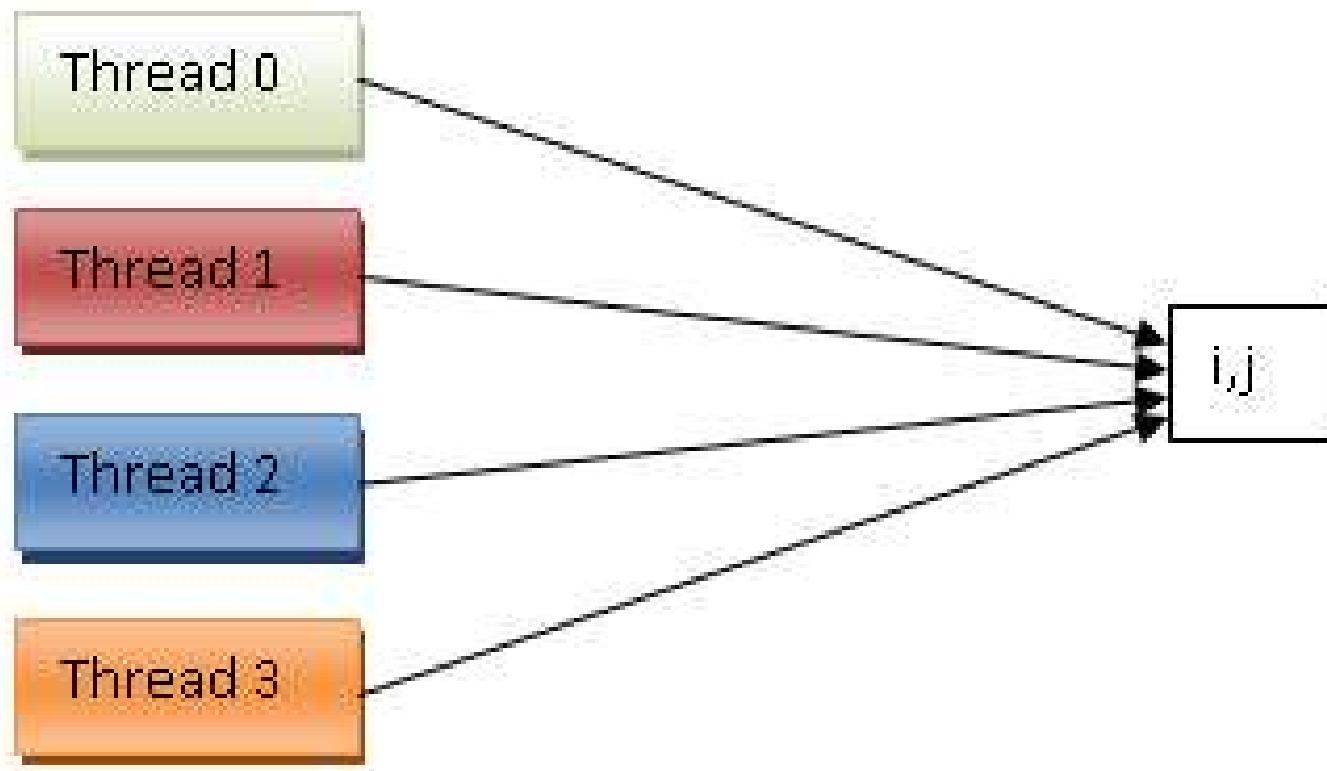
Quando mais de um processo ou thread acessa suas RCs ao mesmo tempo

Exclusão Mútua

Impedir que dois ou mais processos ou threads acessam suas RCs ao mesmo tempo

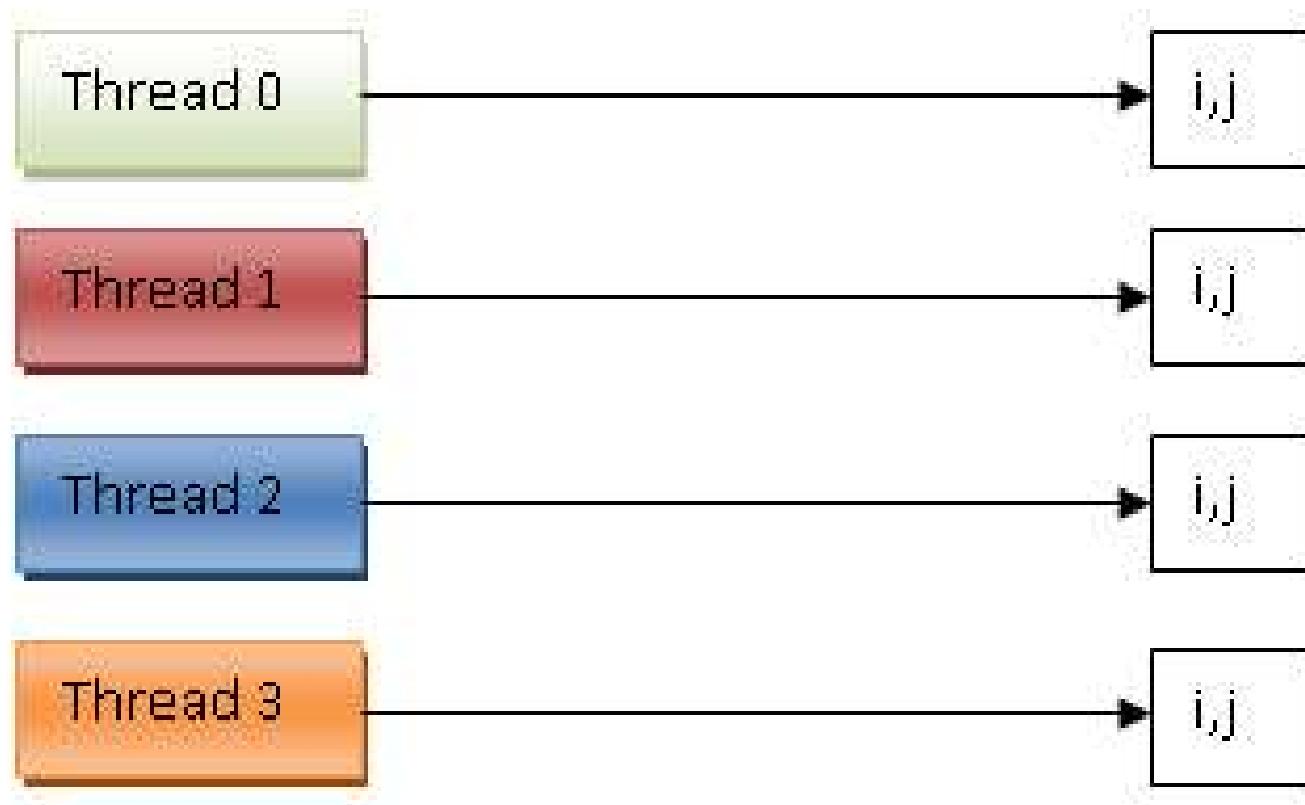
Variáveis i, j no exemplo de matriz

Variáveis Compartilhadas



Variáveis i, j no exemplo de matriz

Variáveis Privadas



Primeira Solução

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    printf("\nOla_1__Fora_da_Regiao_Paralela...\n\n")
    //Declarar as variaveis internamente, tornando-as privadas
#pragma omp parallel
{
    int id = omp_get_thread_num(); //definicao fora da regiao paralela
    int nt = omp_get_num_threads(); //definicao fora da regiao paralela
    printf("Sou_a_thread_%d_de_um_total_%d\n", id, nt);
}
printf("\nOla_2__Fora_da_Regiao_Paralela...\n\n");
return 0;
}
```

Segunda Solução

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    printf("\nOla_1__Fora_da_Regiao_Paralela...\n\n")
    int id, nt; //por default sao variaveis compartilhadas
#pragma omp parallel private (id, nt)
{
    id = omp_get_thread_num(); //definicao fora da regiao paralela
    nt = omp_get_num_threads(); //definicao fora da regiao paralela
    printf("Sou_a_thread_%d_de_um_total_%d\n", id, nt);
}
printf("\nOla_2__Fora_da_Regiao_Paralela...\n\n");
return 0;
}
```

Executando a Segunda Solução

```
./teste
```

```
Ola 1 - Fora da Regiao Paralela ...
```

```
Sou a thread 4 de um total de 8
```

```
Sou a thread 3 de um total de 8
```

```
Sou a thread 2 de um total de 8
```

```
Sou a thread 0 de um total de 8
```

```
Sou a thread 5 de um total de 8
```

```
Sou a thread 6 de um total de 8
```

```
Sou a thread 1 de um total de 8
```

```
Sou a thread 7 de um total de 8
```

```
Ola 2 - Fora da Regiao Paralela ...
```

Soma de matriz - Qual o problema?

```
#include <stdio.h>
#include <omp.h>
int main ( int argc , char *argv [] )
{
    float A[16][16];
    float B[16][16];
    float C[16][16];
    int i , j ;
    for ( i=0;i<16;i++)
    {
        for ( j=0;j<16;j++)
        { A[ i ][ j ]=2;
          B[ i ][ j ]=3;}
    }
    #pragma omp parallel
    {
        #pragma omp for
        for ( i=0;i<16;i++)
        {   for ( j=0;j<16;j++) //Nao ser paralelizado
            C[ i ][ j ] = A[ i ][ j ]+B[ i ][ j ];
        }
    }
    for ( i=0;i<16;i++)
    {   for ( j=0;j<16;j++)
        printf("C[%d][%d] = %f\n" , i , j , C[ i ][ j ]);
    }
    return 0;
}
```

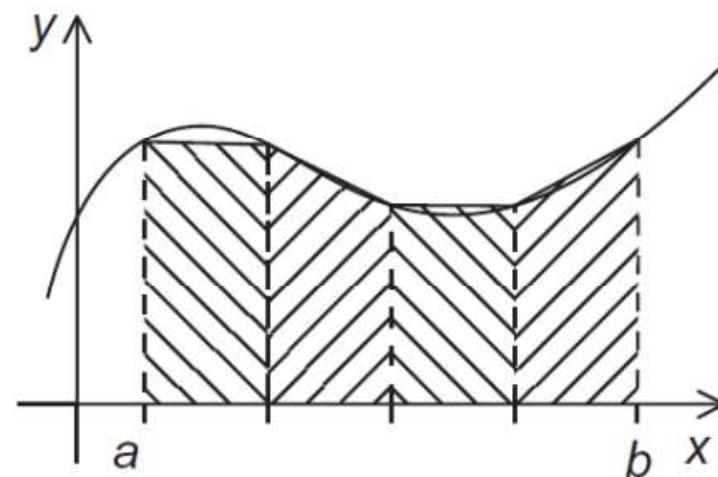
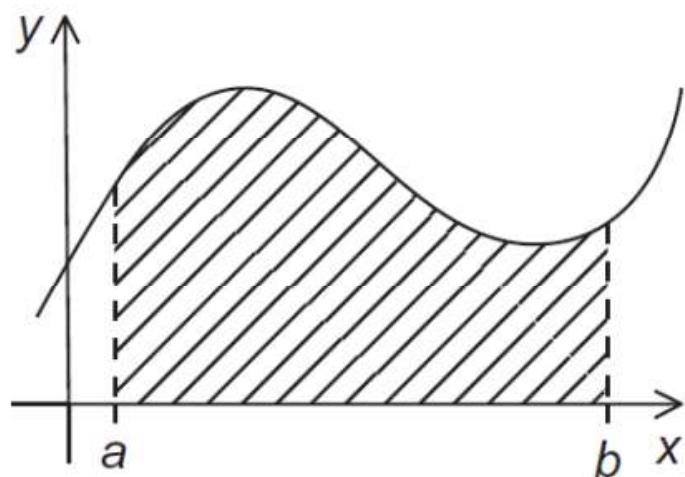
O que aconteceu?

Ao executar, várias vezes pode-se perceber que em algumas execuções o resultado não está correto

Soma de matriz - Solução

```
#include <stdio.h>
#include <omp.h>
int main ( int argc , char *argv [] )
{
    float A[16][16];
    float B[16][16];
    float C[16][16];
    int i , j ;
    for ( i=0;i<16;i++)
    {
        for ( j=0;j<16;j++)
        { A[ i ][ j ]=2;
          B[ i ][ j ]=3;}
    }
    #pragma omp parallel private ( i , j )
    {
        #pragma omp for
        for ( i=0;i<16;i++)
        { for ( j=0;j<16;j++) //Nao ser paralelizado
          C[ i ][ j ] = A[ i ][ j ]+B[ i ][ j ];
        }
    }
    for ( i=0;i<16;i++)
    { for ( j=0;j<16;j++)
      printf("C[%d][%d] = %f\n" , i , j , C[ i ][ j ] );
    }
    return 0;
}
```

A regra do trapézio



Algoritmo Serial

```
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

Uma primeira versão em OpenMP

Identificamos dois tipos de tarefas:

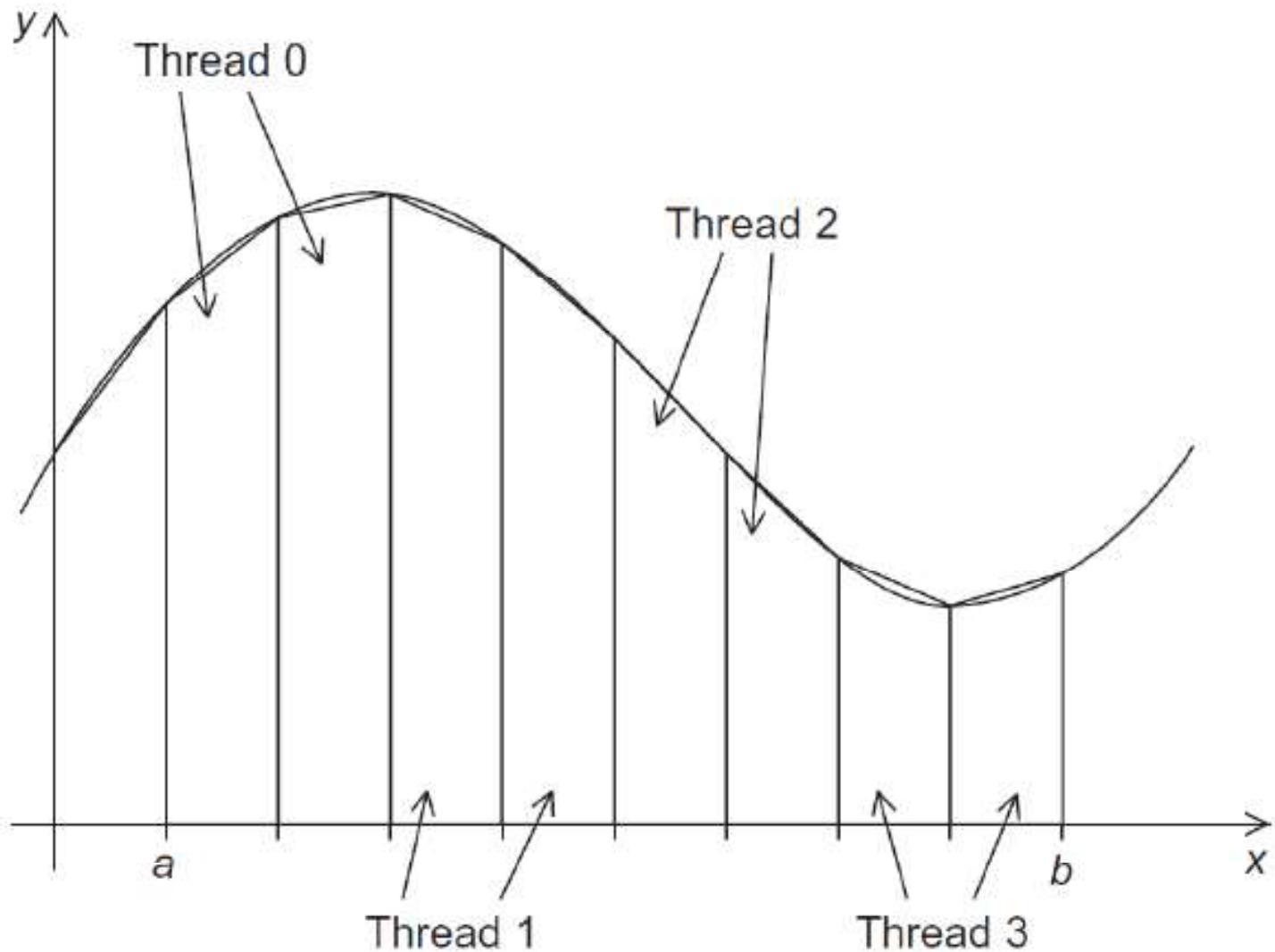
- a) Computação das áreas dos trapézios individuais
- b) Soma das áreas dos trapézios.

Não existe comunicação entre as tarefas do item 1a, mas cada uma delas se comunica com as tarefas do item 1b.

Nós assumimos que existem muito mais trapézios que núcleos.

Agregamos tarefas atribuindo blocos de trapézios consecutivos a cada thread (e uma única thread a cada núcleo).

Atribuindo trapézios à threads



Exclusão mútua

```
# pragma omp critical  
global-result += my-result;
```

pragma omp critical

Somente uma thread pode executar o bloco estruturado por vez

- Mais uma cláusula: **reduction (op:list)**
- usada para operações tipo "all-to-one"
 - exemplo: `op = '+'`
 - cada thread terá cópia da(s) variável(is) definidas em "list" com a devida inicialização;
 - ela efetuará a soma local com sua cópia;
 - ao sair da seção paralela, as somas locais serão automaticamente adicionadas na variável global.

Exemplo - Redução

```
#include <omp.h>
#include <stdio.h>
#define NUM_THREADS 4
int main (){
    int i, tmp, res=0;
    #pragma omp parallel for reduction (+:res) private (tmp)
    for (i=0;i<1000;i++){
        tmp = Calculo();
        res+= tmp;
    }
    printf("O resultado vale %d", res);
}
```

Cálculo de área - Código

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void Trap(double a, double b, int n, double* global_result_p);

int main (int argc, char* argv[]){
double global_result = 0.0;
double a, b;
int n;
int thread_count;

thread_count = strtol(argv[1], NULL, 10);
printf("Entre com a, b, e n\n");
scanf("%f %f %d", &a, &b, &n);

#pragma omp parallel num_threads(thread_count)
Trap(a,b,n,&global_resul);

printf("Com %d trapezios, nossa estimativa é\n", n);
printf("da integral de f para f = %.14e\n", a,b, global_result);
return 0
}
```

Cálculo de área - Código

```
void Trap(double a, double b, int n, double* global_result_p){  
    double h, x, my_result;  
    double local_a, local_b;  
    int i, local_n;  
    int my_rank = omp_get_thread_num(); //numero da thread  
    int thread_count = omp_get_num_threads(); //quantidade de thread  
  
    h = (b-a)/n; //tamanho do passo  
    local_n = n/thread_count; //quantos passos cada thread  
    local_a = a + my_rank*local_n*h; //posicao inicial  
    local_b = local_a + local_n*h; //posicao final  
    my_result = (f(local_a)+f(local_b))/2.0; //media entre os extremos  
    for (i =1; i<=local_n; i++){  
        x = local_a+i*h;  
        my_result += f(x);  
    }  
    my_result = my_result*h;  
  
    #pragma omp critical //acesso a regiao critica  
    *global_result_p += my_result;  
}  
}
```

Dúvida

Precisamos desta versão mais complexa para poder somar o resultado parcial de cada thread em global-result.

```
void Trap(double a, double b, int n, double* global-result-p)
```

O ideal seria usar esta....

```
double Trap(double a, double b, int n);  
global-result = Trap (double a, double b, int n);
```

Se fixarmos desta forma

```
global_result = 0.0;  
# pragma omp parallel num_threads (threads-count)  
{  
# pragma omp critical  
global_result += Trap(double a, double b, int n);  
}
```

Qual o problema?

Solução

```
global_result = 0.0;
# pragma omp parallel num_threads (threads_count)
{
    double my_result = 0.0;
    my_result += Trap(double a, double b, int n);
    # pragma omp critical
    global_result+=my_result;
}
```

Alguém faz melhor?

Solução

```
global_result = 0.0;  
# pragma omp parallel num_threads(threads_count) reduction(:global_result)  
global_result += Trap(double a, double b, int n);  
}
```

Alguém faz melhor?

Objetivo

- Implementar os dois algoritmos do EP1 usando thread baseado em diretivas de compilação (OpenMP)
- Duas versões devem ser implementadas
 - Uma usando uma única thread, programa sequencial
 - Outra usando várias threads
- A entrada de dados deve ser via comando
- Para cada versão, o aluno deverá utilizar os arquivos de dados, os quais contém a sequência a ser ordenada, disponíveis no Tidia e calcular o tempo de ordenação para cada sequência e algoritmo
- Com base nos vários tempos de execução, deve-se plotar um gráfico para cada algoritmo

O relatório deve conter

- Descrição da implementação realizada
- Discussão a respeito do desempenho alcançado pelos algoritmos