

# Programação Paralela

Prof. Dr. André Mendes Garcia

# Introdução

- O que é Programação Paralela
  - Executar simultaneamente dois ou mais códigos, processos, threads ou programas
  - Possibilidades de Execução Paralela:
    - Em uma máquina com apenas um processador com vários núcleos
    - Em uma máquina com vários processadores
    - Utilização do processador da placa de vídeo (GPU) que possui inúmeros núcleos
    - Em rede, entre várias máquinas

# Introdução

- Por que utilizar processamento paralelo
  - Reduzir tempo de execução de um programa
  - Resolver grandes problemas
  - Muitos problemas de computação científica são muito grandes, e necessitam paralelizar
  - Simulações também são problemas pesados, como por exemplo, determinar a previsão do tempo

# Introdução

- Vantagens na Paralelização em Vários Computadores
  - Aproveitamento de recursos disponíveis na rede (Internet), disponibilizados por grandes centros de processamento de dados (Computação em GRID)
  - Baixos custos de processamento, utilizando estações de trabalho, processadores comuns no mercado, ao invés de se utilizar “supercomputadores”
  - Utilização de recursos de memória de diversos computadores

# Conteúdo Programático – 1º Semestre

- Introdução ao Assunto e breve revisão de conceitos importantes
  - Processos
  - Threads
- Conceitos de Programação Paralela
  - Arquiteturas paralelas
  - Memória compartilhada
  - Troca de mensagens
  - Sincronização condicional

# Conteúdo Programático – 1º Semestre

- **Programação**

- **OpenMP**

- Programação Paralela utilizando memória compartilhada

- **MPI**

- Programação Paralela com troca de mensagens e memória independente

# Revisão

- Processos
- Threads

# Processos

- Um programa em execução necessita de recursos de hardware: Processador, Memória e Dispositivos de I/O (Entrada e Saída)
- O Sistema Operacional é responsável por fornecer esses recursos aos programas
- Como vários programas podem estar ativos ao mesmo tempo, o sistema operacional utiliza a abstração de **Processo** para gerenciar o acesso concorrente aos recursos de hardware
- Assim, cria-se a ilusão de que os programas estejam sendo executados ao mesmo tempo e que cada um possui recursos de hardware exclusivos

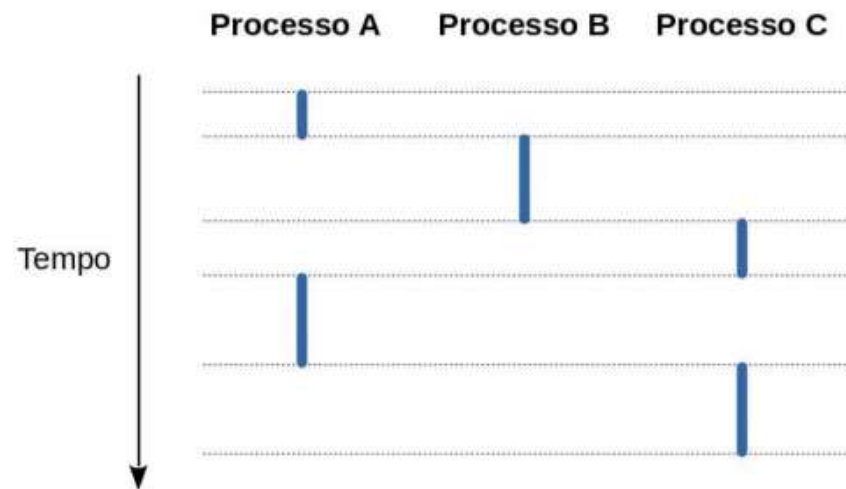


# Processos

- Um **processo** é portanto, um programa em execução
- Quando um programa é executado, é criado um **processo** na memória para este programa, e neste processo contem:
  - Código e dados do programa
  - Pilha de execução (espaço de memória usado para as chamadas de funções)
  - Associação a um conjunto de registradores, e dentre eles:
    - Apontador de instruções (endereço da próxima instrução a ser executada)
    - Apontador do topo da pilha
  - Outras informações e recursos necessários para a execução do programa

# Processos

- Exemplo de execução de três processos em uma máquina com um único processador ao longo do tempo



# Threads

- Nos sistemas operacionais antigos e tradicionais, um **Processo** possuía apenas uma linha de execução, ou também chamado fluxo de controle, ou simplesmente **Thread**
- A tradução de thread é linha, e neste contexto refere-se à linha de execução de um processo
- Nos sistemas operacionais modernos, um processo pode ter diversas linhas de execuções, ou seja, diversas **Threads**, como se fossem “*subprocessos dentro do processo*”
- As Threads dentro de um processo são independentes, exceto pelos recursos de hardware do processo que são compartilhados entre as Threads, incluindo a Memória e os dispositivos de entrada e saída, entre outros...

# Threads

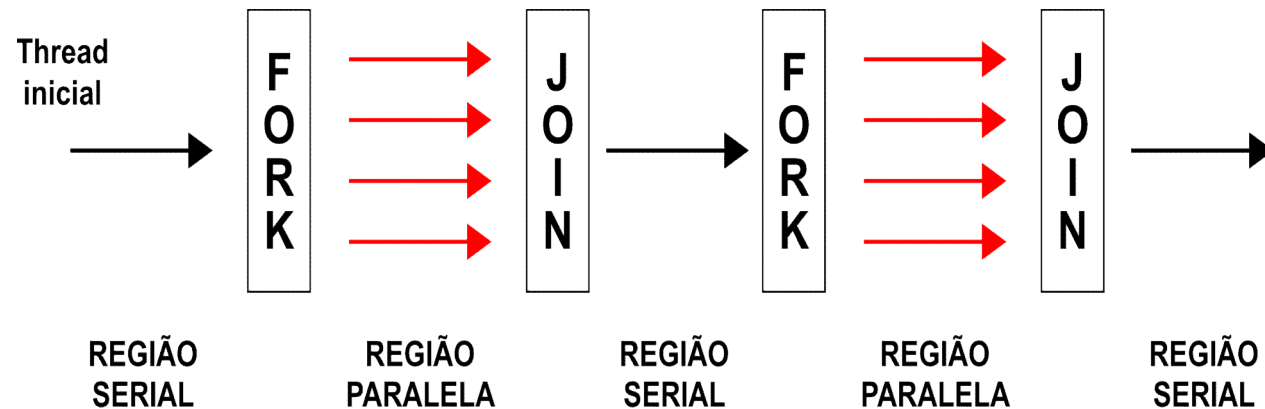
- Com o surgimento de processados com vários núcleos, e também computadores com vários processadores, as Threads podem ser executadas em paralelo em cada núcleo do processador ou em cada processador no caso da máquina tiver vários processadores
- Para realizar a programação paralela explorando o recurso de Threads, são disponibilizados basicamente três API's:
  - OpenMP
  - Win32 Threads
  - POSIX Threads (Pthreads)
- Neste curso será explorada a API OpenMP

# OpenMP

- **OpenMP** é uma API que possui um conjunto de funções e diretivas para o compilador que permite a criação de programas paralelos com compartilhamento de memória, através da criação automática e automatizada de um conjunto de *Threads*
- **API** : Application Programming Interface – Interface de Programação de Aplicativos. Um conjunto de rotinas disponíveis pré-compiladas para prover recursos com alguma finalidade
- **OpenMP** está disponível para as linguagens C/C++ e Fortran

# OpenMP

- O modelo de programação OpenMP baseia-se no modelo do tipo *forkjoin*, ou seja, várias *Threads* são criadas a partir de um ponto do código (*fork*) e em um determinado outro ponto do código as *Threads* adicionais criadas no (*fork*) são eliminadas (*join*) ou colocadas em espera para futuras chamadas

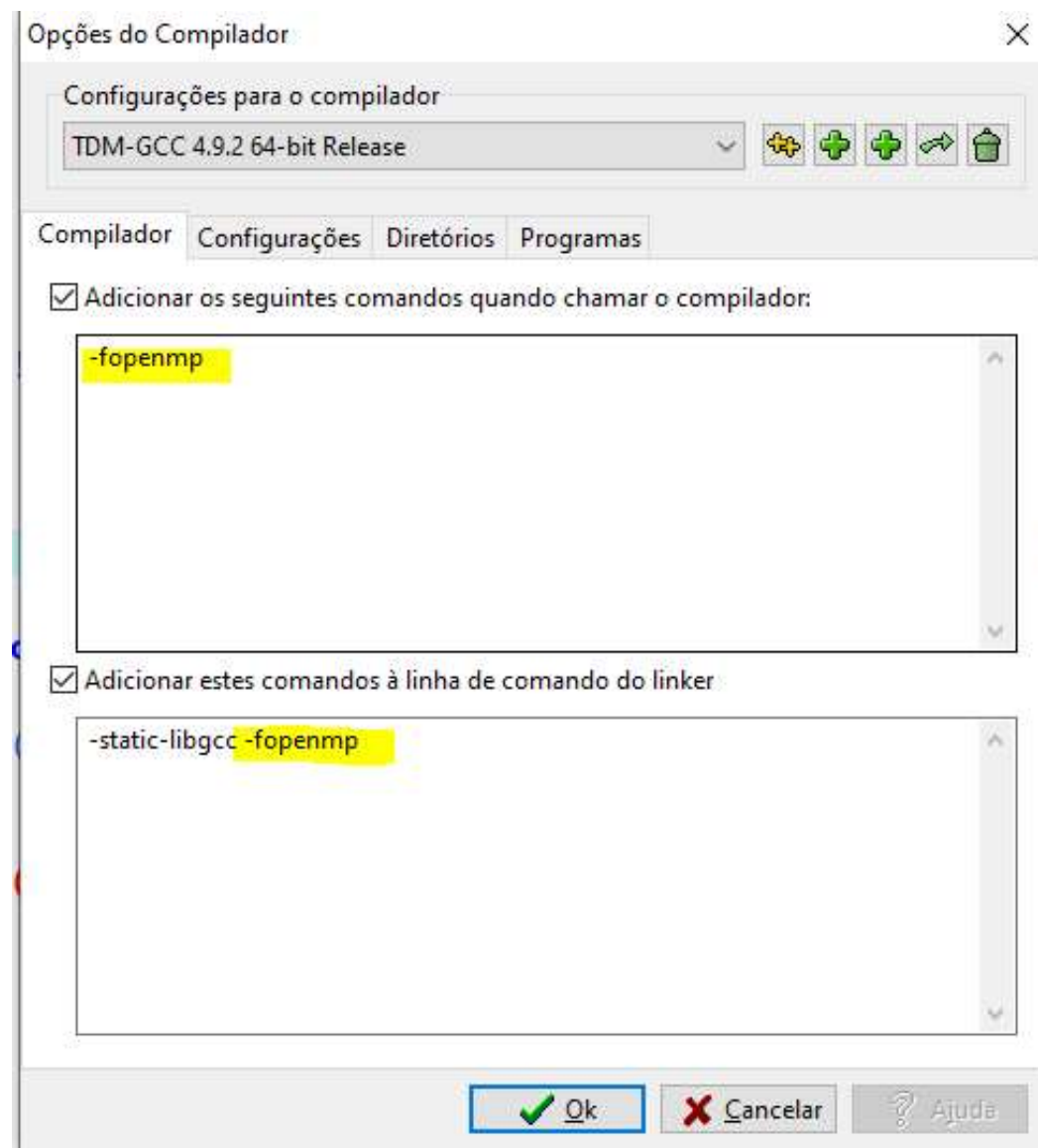


# OpenMP – Hello World em C/C++

- A maioria dos compiladores modernos de C/C++ já possuem recurso para a API OpenMP
- Para utilizar, é necessário:
  1. Adicionar a `#include <omp.h>`
  2. Configurar o compilador para incluir a biblioteca OpenMP, adicionando a seguinte opção nas opções de compilação: `-fopenmp`

# Hello World em C/C++

- Configurando o Compilador no Dev-C++
- Clique em Ferramentas e depois em Opções do Compilador
- Adicione as opções indicadas em amarelo na figura ao lado





# OpenMP – Hello World

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

```
using namespace std;
```

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

```
    printf("\n\nOpenMP : Hello, world!!!\n");
```

```
    int n_threads;
```

```
    n_threads = omp_get_max_threads();
```

```
    printf("\nNumero de Threads (nucleos): %d \n\n ", n_threads);
```

```
    #pragma omp parallel
```

```
{
```

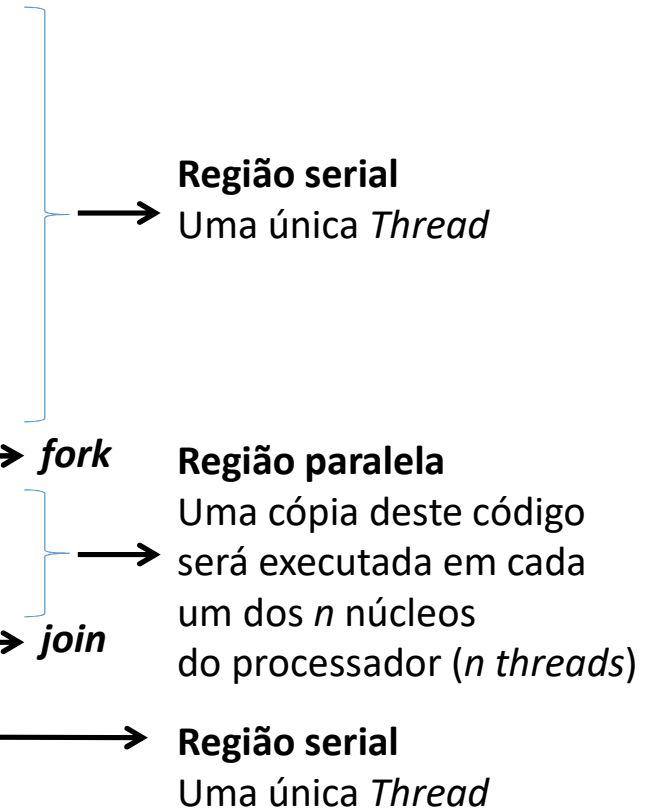
```
    int id_thread = omp_get_thread_num(); // identificação da Thread
```

```
    printf("%d", id_thread);
```

```
}
```

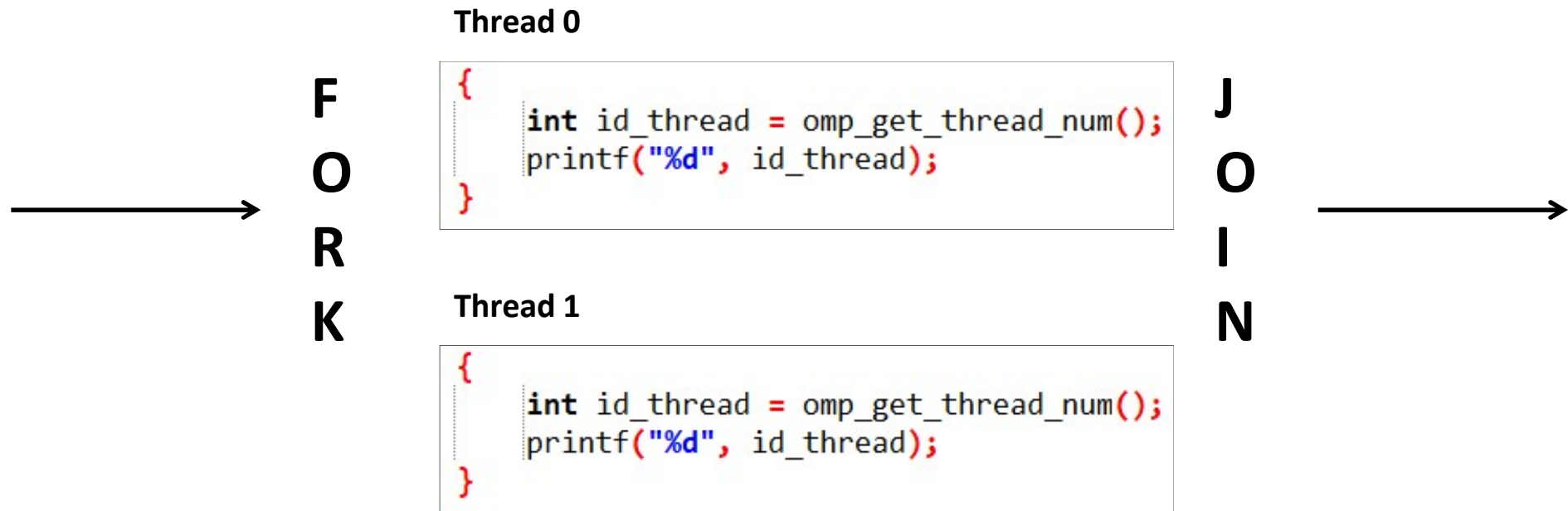
```
    return 0;
```

```
}
```



# OpenMP – Hello World em C/C++

Exemplo de *Forkjoin* em um processador com dois núcleos



# OpenMP – Hello World em C/C++

- Observações sobre o código exemplo:
  - **#pragma omp parallel** : É uma diretiva do compilador que indica que o próximo comando será executado em paralelo entre os núcleos do processador, em *n threads/núcleos* disponíveis
  - **#pragma omp parallel num\_threads(7)** : Indica que serão executadas em paralelo em *7 threads*
  - **omp\_get\_max\_threads()** : obtém o número máximo de núcleos disponíveis na máquina
  - **omp\_get\_num\_threads()** : obtém o número máximo de threads do time atual. O time é o número total de threads em execução, ou seja, se o comando estiver na região serial, o time é composto por apenas 1 thread. Se o comando estiver na região paralela, o time é composto pelo número de threads definidos naquela região
  - **omp\_get\_thread\_num()**: obtém o id, a identificação da thread em execução

# OpenMP – Soma de Vetores

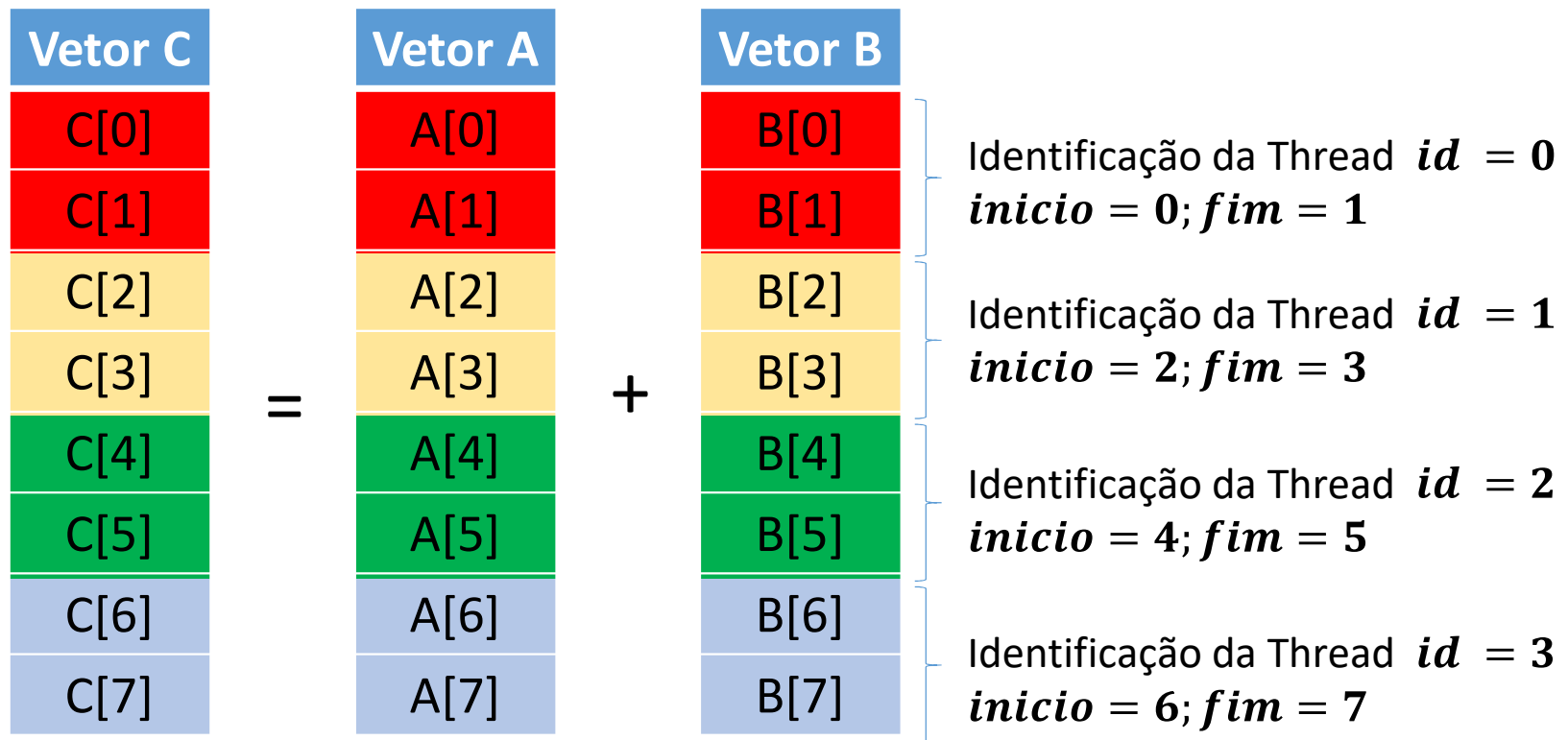
- Somar dois vetores  $A$  e  $B$  consiste em somar suas respectivas posições em um terceiro vetor  $C$

$$C_i = A_i + B_i$$

- A ideia principal é fazer com que cada *thread* some uma parcela dos vetores
- Há duas maneiras de paralelizar a soma de dois vetores utilizando OpenMP
  - Uma das maneiras é manipular os limites da estrutura de repetição, **início** e **fim**, da parcela dos valores dos vetores a serem somados, em função da identificação de cada *thread* e o tamanho de cada parcela
  - A outra maneira é utilizar uma `#pragma` específica da OpenMP para a estrutura `for`, também chamamos este método de maneira implícita

# MPI – Soma de Vetores

- Exemplo: Vetores com 8 posições e processador com 4 núcleos



# OpenMP – Soma de Vetores

- Manipulando os limites da estrutura de repetição

Vetor	
0	Identificação da Thread $id = 0$ $inicio = 0; fim = 1$
1	
2	Identificação da Thread $id = 1$ $inicio = 2; fim = 3$
3	
4	Identificação da Thread $id = 2$ $inicio = 4; fim = 5$
5	
6	Identificação da Thread $id = 3$ $inicio = 6; fim = 7$
7	

Dimensão dos Vetores

$$n = 8$$

Número de *Threads*

$$nt = 4$$

Cada *Thread* deverá processar o número de elementos do vetor dividido pelo número de *Threads*

$$tamanho = \frac{n}{nt} = \frac{8}{4} = 2$$

Os limites podem ser obtidos em função do  $id$  e  $tamanho$

$$inicio = id * tamanho$$

$$fim = inicio + tamanho - 1$$

# MPI – Soma de Vetores

- **Observações:**

- Quando se faz a divisão para obter o tamanho de cada parcela do vetor a ser somada, o número pode ser quebrado, exemplo:

$$\frac{n}{nt} = \frac{100}{16} = 6,25 \cong 6$$

- Neste caso o tamanho do vetor é **100** e o número de núcleos do processador é **16**. O tamanho deve ser sempre um número inteiro porque se trata de posições dos vetores. Neste caso, o número é trucado para **6**
- Multiplicando **6 \* 16 = 94**, entretanto, os vetores possuem tamanho **100**, então as **6** últimas posições do vetor, de **94** a **99**, não serão processadas
- **SOLUÇÃO:** Na thread de id maior, fazer com que o limite **fim** assuma a última posição do vetor, e o **tamanho** será ***tamanho = fim - inicio + 1***

# MPI – Soma de Vetores

- Exemplo da divisão das somas das posições dos vetores de dimensão **100** e com um processador de **16** núcleos

id da Thread	inicio	fim	tamanho
0	0	5	6
1	6	11	6
2	12	17	6
3	18	23	6
4	24	29	6
5	30	35	6
6	36	41	6
7	42	47	6
8	48	53	6
9	54	59	6
10	60	65	6
11	66	71	6
12	72	77	6
13	78	83	6
14	84	89	6
15	90	99	10



# MPI – Soma de Vetores

- **Observações:**

- Uma outra observação a ser feita é que, o número de threads pode ser maior que o número de posições dos vetores, por exemplo:

$$n = 8$$

$$nt = 16$$

- Neste caso, deve-se atribuir uma posição para cada thread, e as demais threads com *id* superior ao número de posições dos vetores, não devem realizar processamento, como ilustra a tabela ao lado

id da thread	Soma
0	$C[0] = A[0] + B[0]$
1	$C[1] = A[1] + B[1]$
2	$C[2] = A[2] + B[2]$
3	$C[3] = A[3] + B[3]$
4	$C[4] = A[4] + B[4]$
5	$C[5] = A[5] + B[5]$
6	$C[6] = A[6] + B[6]$
7	$C[7] = A[7] + B[7]$
8	-
9	-
10	-
11	-
12	-
13	-
14	-
15	-

# OpenMP – Soma de Vetores

## Exemplo Sem Paralelismo

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

    int const n = 10;
    float A[n], B[n], C[n];

    // alimentar os vetores A e B com valores quaisquer
    for(int i=0; i<n; i++)
    {
        A[i] = i * sin(i);
        B[i] = A[i] - cos(i*i);
    }

    // realizando a soma dos vetores
    for(int i=0; i<n; i++)
    {
        C[i] = A[i] + B[i];
    }

    // imprimindo o resultado
    for(int i=0; i<n; i++)
    {
        printf("\nA[%d] + B[%d] = %10.3f", i, i, C[i]);
    }

    return 0;
}
```



```
#pragma omp parallel // fork
{
    int tamanho, inicio, fim;
    int id = omp_get_thread_num(); // identificação da Thread
    int nt = omp_get_num_threads(); // número total de threads disponíveis no time

    if( nt > n ) // No. de threads maior que o de posições dos vetores
    {
        if( id <= n-1 )
        {
            tamanho = 1;
            inicio = id;
            fim = id;
        }
        else
        {
            inicio = 0;
            fim = -1;
            tamanho = 0;
        }
    }

    // nt > n
    else
    {
        // No. de threads <= n
        tamanho = n / nt;
        inicio = id * tamanho;
        fim = inicio + tamanho - 1;

        if( id == nt-1 )
        {
            fim = n-1;
            tamanho = fim - inicio + 1;
        }
    }

    // nt <= n

    for(int i = inicio; i <= fim; i++)
    {
        C[i] = A[i] + B[i];
    }
}

} // join
```

# OpenMP – Soma de Vetores

- Algumas observações sobre o código:
  - Note que as variáveis *id*, *i*, *inicio*, e *fim* devem ser declaradas dentro do escopo da *#pragma omp parallel*
  - Desta forma elas serão variáveis privadas em cada *thread*, ou seja, duplicadas e exclusivas em cada *thread*
  - Se elas fossem declaradas antes do *fork* (*#pragma omp parallel*), elas seriam compartilhadas entre as *n threads*, e como as *threads* são executadas em paralelo, cada *thread* pode interferir nos valores destas variáveis na execução das outras *threads*

# OpenMP – Soma de Vetores

- Utilizando o construtor de trabalho **#pragma omp for** da OpenMP para paralelizar o processamento da estrutura **for**
- A **#pragma omp for** faz com que o processamento da estrutura **for** logo abaixo dela, seja paralelizado de forma igual entre as *threads* automaticamente

```
#pragma omp parallel // fork
{
    int i;

    #pragma omp for
    for(i = 0; i < n; i++)
    {
        C[i] = A[i] + B[i];
    }
} // join
```

# OpenMP – Soma de Vetores

- Qual maneira é melhor ?
  - As duas maneiras resolvem o problema da mesma forma sem diferença de desempenho ou resultado final
  - A maneira fazendo a manipulação dos limites da estrutura for, é interessante porque o programador tem maior controle da situação, podendo resolver outro problema em que o **#pragma omp for** não poderia resolver

# OpenMP – Soma de Matrizes

- Para realizar a soma de matrizes utilizando programação paralela com OpenMP, podemos utilizar uma analogia ao algoritmo de soma de vetores
- Há duas possibilidades:
  - 1 : Paralelizar apenas o for das linhas ou o for das colunas. Neste caso temos uma abordagem de uma dimensão
  - 2 : Paralelizar em duas dimensões, ou seja, dividir o código de forma paralela tanto para as linhas quanto para as colunas
- Entretanto, das duas maneiras, o resultado será o mesmo.

# OpenMP – Soma de Matrizes

- A soma de matrizes exige que as dimensões das matrizes envolvidas sejam iguais, ou seja, a quantidade de linhas e colunas das matrizes devem ser iguais
- A matriz resultante terá a mesma dimensão das matrizes somadas. A expressão abaixo ilustra a soma de duas matrizes A e B

$$C_{ij} = A_{ij} + B_{ij}$$

# OpenMP – Soma de Matrizes

- Exemplo de paralelização da soma de matrizes unidimensional, paralelizando apenas o processamento das linhas

Soma de matrizes sem paralelização

```
for(int i=0; i<nl; i++)  
{  
    for(int j=0; j<nc; j++)  
    {  
        C[i][j] = A[i][j] + B[i][j];  
    }  
}
```

Soma de matrizes COM paralelização

```
#pragma omp parallel // fork  
{  
    #pragma omp for  
    for(int i=0; i<nl; i++)  
    {  
        for(int j=0; j<nc; j++)  
        {  
            C[i][j] = A[i][j] + B[i][j];  
        }  
    }  
} // join
```



# OpenMP – Cláusula IF da *#pragma omp parallel*

- **Cláusula if** : É utilizada partindo-se da premissa que só compensa paralelizar um código quando a carga de trabalho é muito grande

```
#pragma omp parallel if (n > 1000)
{
    #pragma omp for
    for(int i=0; i<n; i++)
    {
        C[i] = A[i] + B[i];
    }
}
```

- Neste exemplo, a região paralela só será dividida em threads e executada em paralelo, se n for maior que 1000

# OpenMP – Escopo de Variáveis

- As variáveis declaradas fora do bloco de código do `#pragma omp parallel`, são globais, ou seja, são vistas e são as mesmas dentro de todas as threads
- As variáveis declaradas dentro do bloco de código do `#pragma omp parallel`, são privadas/locais dentro de cada thread

# OpenMP – Escopo de Variáveis

- Exemplo:

```
int i, j;
```



Variáveis **globais** que serão vistas dentro de todas as threads do time da região paralela

```
#pragma omp parallel
```

```
{
```

```
    int x, y;
```



```
    printf("%d", omp_get_thread_num() );
```

```
}
```

Variáveis **locais** independentes em cada thread. Ou seja, em cada thread do time irá existir as variáveis x e y **privadas**

# OpenMP – Escopo de Variáveis

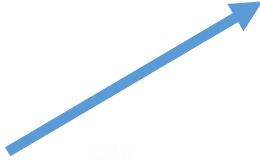
- Eventualmente, dependendo do problema a ser resolvido, devem existir variáveis globais e que dentro das threads na região paralela, essas variáveis devem ter comportamento de variáveis privadas
- Para isso, deve-se utilizar a cláusula **private** da `#pragma omp parallel`

# OpenMP – Escopo de Variáveis

- Exemplo:

```
int i, j, x, y;  
  
#pragma omp parallel private(x, y)  
{  
    x = omp_get_thread_num();  
    y = x + 2;  
  
    printf("%d", x + y );  
}
```

A cláusula private, neste caso, faz com que as variáveis globais x e y, sejam variáveis privadas em cada thread do time na região paralela



# OpenMP – Construtores de Trabalho

- São responsáveis pela distribuição de trabalho entre as Threads e determinam como o trabalho será dividido
  - **#pragma omp for**
  - **#pragma omp single**
  - **#pragma omp master**
  - **#pragma omp sections**

# OpenMP – Construtores de Trabalho


- **#pragma omp for**
  - Divide a tarefa da estrutura for de forma igual para todas as threads do time da região paralela

# OpenMP – Construtores de Trabalho

- **#pragma omp for**

```
#pragma omp parallel // fork
{
    int i;

    #pragma omp for
    for(i = 0; i < n; i++)
    {
        C[i] = A[i] + B[i];
    }
} // join
```



Divide de forma igual a tarefa de somar o vetor entre todas as threads.

Se a quantidade de threads for 4 por exemplo, cada thread irá executar  $\frac{n}{4}$  somas

Além disso, este construtor faz com que a variável de controle do for, seja privada em todas as threads



# OpenMP – Construtores de Trabalho

- **#pragma omp single**
  - Faz com que o bloco de código seja executado em apenas uma thread

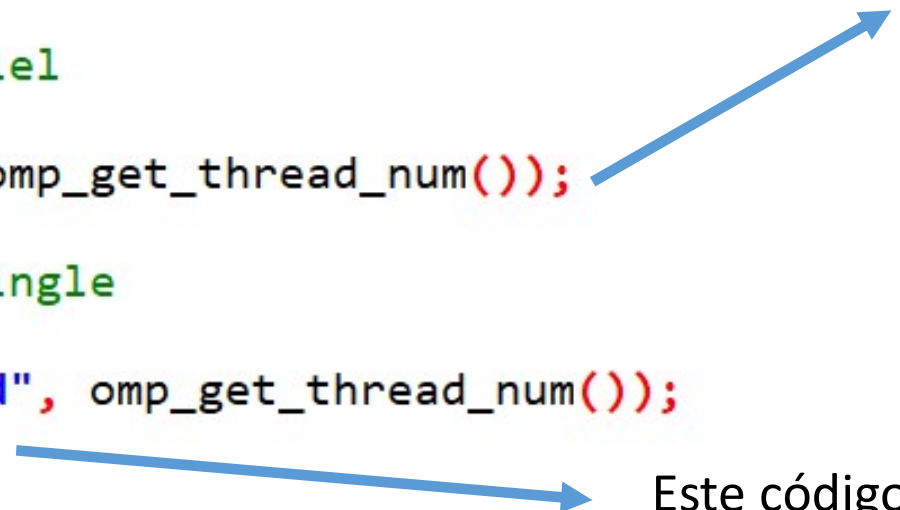
# OpenMP – Construtores de Trabalho

- **#pragma omp single**

Este código será executado em paralelo nas  $n$  threads do time

```
#pragma omp parallel
{
    printf("%d", omp_get_thread_num());

    #pragma omp single
    {
        printf("%d", omp_get_thread_num());
    }
} // join
```



Este código será executado em paralelo em apenas 1 thread do time

# OpenMP – Construtores de Trabalho

- **#pragma omp master**
  - Semelhante ao construtor de trabalho **single**, faz com que o bloco de código seja executado em apenas uma thread, com a diferença de que esta thread será a thread principal chamada **thread master**

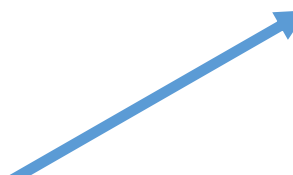
# OpenMP – Construtores de Trabalho

- **#pragma omp master**


```
#pragma omp parallel
{
    printf("%d", omp_get_thread_num());

    #pragma omp master
    {
        printf("%d", omp_get_thread_num());
    }
} // join
```

Este código será executado em paralelo nas ***n*** threads do time



Este código será executado em paralelo apenas uma vez na thread **master**



# OpenMP – Construtores de Trabalho

- **#pragma omp sections**

- Nem sempre o problema que está se resolvendo necessita trabalhar em paralelo com todas as threads/núcleos disponíveis no sistema
- Por exemplo, supomos que o computador de trabalho possua 10 núcleos e o problema a ser resolvido necessita de processar apenas 2 blocos de códigos em paralelo, assim, necessita-se de apenas 2 threads/núcleos
- Neste caso o ideal é executar os dois blocos de código, um em cada thread, e as demais 8 threads deveriam estar ociosas

# OpenMP – Construtores de Trabalho

- **#pragma omp sections**
  - Para isto existe o construtor de trabalho **sections**
  - O construtor **sections** faz com que blocos de códigos individuais sejam executados em threads individuais

# OpenMP – Construtores de Trabalho

- **Exemplo:** Atribuir valores para dois vetores A e B, sendo que a atribuição dos valores de A deve ser numa thread e atribuição dos valores de B em outra thread

# OpenMP – Construtores de Trabalho

```
#pragma omp parallel
```

→ São criadas **n threads**

```
{  
  #pragma omp sections
```

→ O bloco de código abaixo desta diretiva será executado em apenas uma thread

```
{  
  #pragma omp section
```

```
{  
    for(int i=0; i<n; i++)  
    {  
        A[i] = i + 10;  
    }  
} // section1
```

```
#pragma omp section
```

→ O bloco de código abaixo desta diretiva será executado em apenas uma **OUTRA** thread

```
{  
    for(int i=0; i<n; i++)  
    {  
        B[i] = i * 8;  
    }  
} // section2
```

```
} // sections
```

```
} // join
```

**Observação:** Supondo que o processador possua 10 núcleos/threads neste exemplo apenas duas threads serão executadas em paralelo, as demais 8 threads estarão ociosas



# OpenMP – Construtores de Trabalho

- **BARREIRAS**

- Todo construtor de trabalho cria uma barreira e não deixa nada ser executado após o seu bloco de código, até que todas as tarefas do seu bloco sejam executadas por completo

# OpenMP – Construtores de Trabalho

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            for(int i=0; i<n; i++) A[i] = i + 10;
        }

        #pragma omp section
        {
            for(int i=0; i<n; i++) B[i] = i * 8;
        }
    } // sections
    //////////////////////////////////////
    /// BARREIRA
    //////////////////////////////////////
    #pragma omp single
    {
        for(int i=0; i<n; i++) C[i] = i + 158;
    }
} // join
```

Neste ponto do código é criada uma **barreira**

Desta forma, o código do construtor **single** não será executado e nem alocado em uma thread até que as duas sections terminem seus respectivos processamentos

# OpenMP – Construtores de Trabalho

- **Liberando as BARREIRAS com nowait**
  - **nowait** é uma cláusula que é utilizada com os construtores de trabalho para que não sejam criadas barreiras

# OpenMP – Construtores de Trabalho

```
#pragma omp parallel
{
    #pragma omp sections nowait
    {
        #pragma omp section
        {
            for(int i=0; i<n; i++) A[i] = i + 10;
        }

        #pragma omp section
        {
            for(int i=0; i<n; i++) B[i] = i * 8;
        }
    } // sections

    #pragma omp single
    {
        for(int i=0; i<n; i++) C[i] = i + 158;
    }
} // join
```

A cláusula **nowait** no construtor sections faz com que não seja criada uma barreira

Desta forma, o código correspondente ao construtor **single** será alocado em um thread ociosa e será executado em paralelo

# OpenMP – Região Crítica

- **O que é Região Crítica ?**

- É um determinado trecho de código onde este código não pode ser executado de forma paralela
- Geralmente nestes trechos de códigos, região crítica, há uma ou mais variáveis **PÚBLICAS** a todas as **Threads** que são atualizadas
- Pelo fato destas variáveis ser públicas, não se pode atualizá-las de forma paralela pois senão a lógica do algoritmo seria “quebrada”

# OpenMP – Região Crítica

- **Exemplo: Produto escalar entre dois vetores**

- Considere dois vetores  $A$  e  $B$  de dimensão  $n$ , o produto escalar  $P$  entre esses dois vetores é dado pela seguinte expressão:

$$P = A[1] * B[1] + A[2] * B[2] + \dots + A[n] * B[n]$$

# OpenMP – Região Crítica

- **Produto escalar entre dois vetores : Versão Single**

```
P = 0;
```

```
// produto escalar entre os vetores A e B
```

```
for(int i=0; i<n; i++)
```

```
{
```

```
    P = P + A[i] * B[i];
```

```
}
```

# OpenMP – Região Crítica

- Produto escalar entre dois vetores :  
Versão Parallel **com Problemas**

```
P = 0;
```

```
// produto escalar entre os vetores A e B
```

```
#pragma omp parallel
```

```
{
```

```
    #pragma omp for
```

```
    for(int i=0; i<n; i++)
```

```
    {
```

```
        P = P + A[i] * B[i]; // Região Crítica
```

```
    }
```

```
} // join
```

A variável **P** deve ser global/pública, ou seja, deve ser vista fora e dentro da região paralela. Todas as threads devem ver ela.

O problema deste algoritmo está na região crítica.

Esta região é crítica porque a variável **P** é pública a todas as threads, e é atualizada em paralelo por todas elas.

Neste caso, o valor de **P** poderá ser calculado de forma errada.



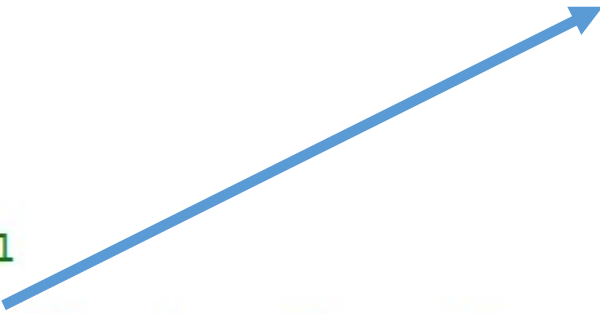
# OpenMP – Região Crítica

- **Como resolver o problema da Região Crítica**
  - ***#pragma omp critical***
  - O código abaixo da ***#pragma omp critical***, será executado apenas por uma thread
  - Então utiliza-se esta ***#pragma*** para evitar que atualizações de variáveis públicas, por exemplo, sejam feitas em paralelo

# OpenMP – Região Crítica

- Produto escalar entre dois vetores :  
Versão Parallel **Corrigido**, mas com problema de Desempenho

```
#pragma omp parallel
{
    #pragma omp for
    for(int i=0; i<n; i++)
    {
        #pragma omp critical
        {
            P = P + A[i] * B[i]; // Região Crítica
        }
    }
} // join
```



Como esta região crítica está dentro da `#pragma omp critical`, não teremos problemas de atualizações em paralelo da variável **P**, que é pública.

Porque esta pragma garante que esta atualização seja feita por apenas uma thread de cada vez

## Por que o desempenho não é bom ?

Como apenas uma thread será capaz de executar este código, então o desempenho será semelhante ao algoritmo não paralelizado. Na realidade, o desempenho será pior que o algoritmo single, porque custa tempo para criar estas regiões críticas

# OpenMP – Região Crítica

- Produto escalar entre dois vetores :  
Versão Parallel **Corrigido e com bom desempenho**

```
#pragma omp parallel
{
    // variável privada (local) de cada thread
    int PAUX = 0;

    #pragma omp for
    for(int i=0; i<n; i++)
    {
        PAUX = PAUX + A[i] * B[i];
    } // barreira

    #pragma omp critical
    {
        P = P + PAUX; // Região Crítica
    }

} // join
```

Cada thread terá sua própria variável (privada) PAUX  
Então não teremos problema de atualizações paralelas.

Neste exemplo, o número de vezes que a região crítica  
será executada é muito menor que o exemplo anterior.


No exemplo anterior, a quantidade de vezes que a  
região crítica é executada, é a dimensão do vetor, no  
caso  $n$ .

Aqui, a quantidade de vezes que a região crítica será  
executada, é o número de threads apenas.

# OpenMP – Região Crítica

- Produto escalar entre dois vetores :  
Versão Parallel **Utilizando a Cláusula REDUCTION**

```
#pragma omp parallel
{
    #pragma omp for reduction (+:P)
    for(int i=0; i<n; i++)
    {
        P = P + A[i] * B[i];
    }
} // join
```



A cláusula **reduction**, neste exemplo, solicita ao compilador que crie  $X$  variáveis  $P$  privadas, uma em cada thread, onde  $X$  é o número de threads do time.

Após terminar o for, o valor de cada  $P$  privado é somado à variável  $P$  global.

Como pode ser visto a cláusula **reduction** é de altíssimo nível de programação.

Entretanto ela é limitada aos operadores da linguagem. Caso necessite de proteger uma região crítica com um cálculo mais complexo, por exemplo, o **reduction** não resolve.

# OpenMP – Multiplicação de Matrizes

- A multiplicação entre duas matrizes exige que o número de colunas da primeira matriz seja igual ao número de linhas da segunda matriz exemplo:

$$C_{ac} = A_{ab} * B_{bc}$$

$$C_{24} = A_{23} * B_{34}$$

- Os elementos da matriz  $C$  são obtidos com a seguinte expressão:

$$C_{ij} = \sum_{k=0}^{ncA-1 \text{ ou } nlB-1} A_{ik} * B_{kj}$$

# OpenMP – Multiplicação de Matrizes

- Para paralelizar a multiplicação de matrizes devemos primeiramente analisar o algoritmo sem paralelização, como segue:

```
for(int i=0; i<nlC; i++)
{
    for(int j=0; j<ncC; j++)
    {
        C[i][j] = 0;
        for(int k=0; k<ncA; k++)
        {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        } // k
    } // j
} // i
```

# OpenMP – Multiplicação de Matrizes

- É importante analisarmos todos os detalhes do algoritmo, como por exemplo, a declaração das variáveis *i*, *j* e *k* dentro das estruturas for.
- Como o número de linhas e colunas pode ser bastante elevado, há um custo computacional considerável em criar essas variáveis estáticas
- Sendo assim, o ideal é declarar estas variáveis antes
- Além disso, poderíamos inicializar todos os elementos da matriz *C* antes do algoritmo de multiplicação

# OpenMP – Multiplicação de Matrizes

- Algoritmo de Multiplicação de Matrizes não paralelo e otimizado

```
for(i=0; i<nlC; i++)
{
    for(j=0; j<ncC; j++)
    {
        C[i][j] = 0;
    }
}

for(i=0; i<nlC; i++)
{
    for(j=0; j<ncC; j++)
    {
        for(k=0; k<ncA; k++)
        {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        } // k
    } // j
} // i
```



# OpenMP – Multiplicação de Matrizes

- Algoritmo de Multiplicação de Matrizes **PARALELIZADO**

```
#pragma omp parallel private(i, j, k)
{
    #pragma omp for
    for(i=0; i<nlC; i++)
    {
        for(j=0; j<ncC; j++)
        {
            float s=0;

            for(k=0; k<ncA; k++)
            {
                s = s + A[i][k] * B[k][j];
            } // k

            #pragma omp critical
            {
                C[i][j] = C[i][j] + s;
            }

        } // j
    } // i
}
```

Inicializa a região paralela – *fork*

Note que, as variáveis *i*, *j* e *k* precisam ser privadas em cada thread

Como elas já foram declaradas antes, foi necessário utilizar a cláusula *private*

O construtor de trabalho `#pragma omp for` dividiu o trabalho do `for` que percorre as linhas da matriz *C*, de forma igual entre todas as threads

Como a matriz *C* é pública, e há uma atualização de valores nela, foi necessário proteger esta atualização em uma região crítica, fazendo com que apenas uma thread possa executar seu bloco de código por vez

# OpenMP – Multiplicação de Matrizes

- Testes : Os algoritmos de multiplicação de matrizes, single e parallel, foram executados em dois microcomputadores distintos
- As dimensões das matrizes utilizadas foram:  $A_{2000 \times 3000} * B_{3000 \times 4000}$

## Resultados dos Testes

Equipamento	Algoritmo Single Tempo em segundos	Algoritmo Parallel Tempo em segundos
Processador: Intel Core 2Duo 2.2 GHz Núcleos: 2 Memória RAM: 32 GB	894	642
Processador: Intel i9 3.6 GHz Núcleos: 12 Memória RAM: 32 GB	223	21

# Fatoração LU

- Consiste em transformar uma matriz quadrada  $A$  em duas matrizes  $L$  e  $U$ , onde:

$$A = L * U$$

- Sendo que  $L$  é uma matriz **Triangular Inferior**, onde todos seus membros **acima** da diagonal principal são **0** (zero)
- E  $U$  uma matriz **Triangular Superior**, onde todos seus elementos **abaixo** da diagonal principal são **0** (zero)

# Fatoração LU

- Com a fatoração LU, é possível resolver sistemas lineares e calcular a matriz inversa de outra matriz
- Exemplo: Considere o seguinte sistema linear:  $\mathbf{Ax} = \mathbf{b}$

$$\begin{aligned}2x_1 + 3x_2 - x_3 &= 5 \\4x_1 + 4x_2 - 3x_3 &= 3 \\2x_1 - 3x_2 + x_3 &= -1\end{aligned}$$



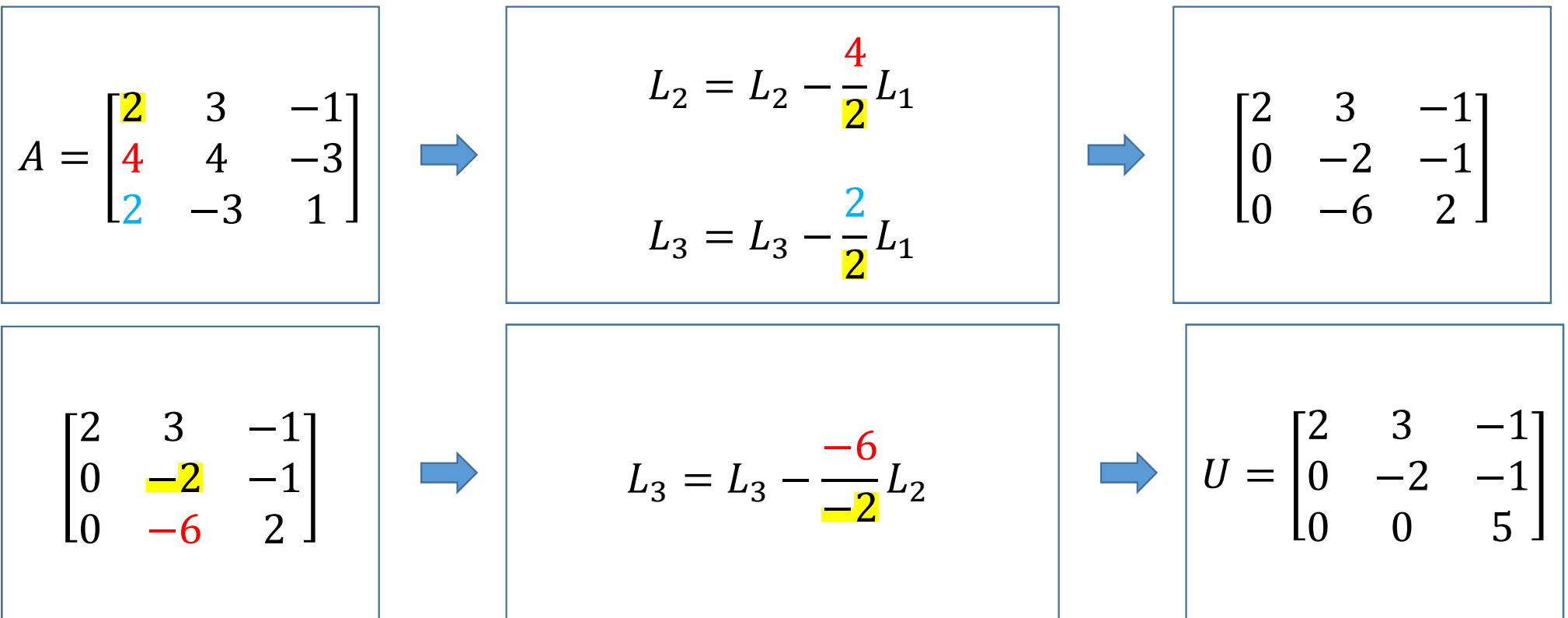
$$\begin{bmatrix} 2 & 3 & -1 \\ 4 & 4 & -3 \\ 2 & -3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \\ -1 \end{bmatrix}$$

$A \qquad x \qquad b$

- Encontrando a matriz  $\mathbf{L}$  e  $\mathbf{U}$  a partir da matriz  $\mathbf{A}$ , é possível resolver o sistema linear  $\mathbf{Ax} = \mathbf{b}$  de forma direta realizando substituições

# Fatoração LU

- Calculando a matriz triangular superior  $U$



# Fatoração LU

- Calculando a matriz triangular superior  $\mathbf{L}$
- Todos elementos da diagonal principal são 1. Os elementos acima da diagonal primal são 0 (zero)

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ ? & 1 & 0 \\ ? & ? & 1 \end{bmatrix}$$

- Os elementos ? de  $\mathbf{L}$  são os termos multiplicativos utilizados para zerar os elementos abaixo da diagonal principal de  $\mathbf{U}$ , ou seja:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 2 & -6 & 1 \\ 2 & -2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 3 & 1 \end{bmatrix}$$

# Fatoração LU

- Algoritmo para calcular as matrizes triangulares  $U$  e  $L$

```
//--- inicializando as matrizes L e U
for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
    {
        U[i][j] = A[i][j];

        if( i == j ) // diagonal principal
            L[i][j] = 1;
        else
            if( j > i ) // acima da diagonal principal
                L[i][j] = 0;
            else
                // i < j : abaixo da diagonal principal os
                // valores serão calculados posteriormente
                L[i][j] = 0;
    } // for j
} // for i
```

```
//--- calculando as matrizes L e U
for(k=0; k<n-1; k++)
{
    for(i=k+1; i<n; i++)
    {
        numerador = U[i][k];
        denominador = U[k][k];

        L[i][k] = numerador/denominador;

        for(j=k+1; j<n; j++)
        {
            U[i][j] = U[i][j] - L[i][k] * U[k][j];
        }
    } // for i
} // for k
```

# Fatoração LU

- **Resolvendo o Sistema Linear**

- Como  **$b$**  é conhecido, acha-se os valores de  **$y$**  através da seguinte expressão:

$$Ly = b$$

- A partir dos valores de  **$y$** , resolve-se o sistema linear achando os  **$x$**  através da seguinte expressão:

$$Ux = y$$



Fatoração LU : Calculando os valores de  $\mathbf{y}$

$$L\mathbf{y} = \mathbf{b}$$

*forward  
substitution*

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 3 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \\ -1 \end{bmatrix} \quad \begin{array}{l} 1y_1 + 0y_2 + 0y_3 = 5 \\ 2y_1 + 1y_2 + 0y_3 = 3 \\ 1y_1 + 3y_2 + 1y_3 = -1 \end{array} \quad \begin{array}{l} 1y_1 = 5 \\ 2y_1 + 1y_2 = 3 \\ 1y_1 + 3y_2 + 1y_3 = -1 \end{array}$$

$$y_1 = 5$$

$$y_2 = 3 - 2y_1$$

$$y_3 = -1 - (1y_1 + 3y_2)$$



$$y_i = b_i - \sum_{j=1}^{i-1} L_{ij} y_j \quad \text{para } i = 1, 2, \dots, n$$

$$y_1 = \frac{5}{1} \Rightarrow y_1 = 5$$

$$2 * 5 + 1y_2 = 10 + 1y_2 = 3 \Rightarrow y_2 = \frac{3 - 10}{1} \Rightarrow y_2 = -7$$

$$1 * 5 + 3 * -7 + 1y_3 = 5 - 21 + 1y_3 = -1 \Rightarrow y_3 = \frac{-1 + 16}{1} \Rightarrow y_3 = 15$$

$$\mathbf{y} = \begin{bmatrix} 5 \\ -7 \\ 15 \end{bmatrix}$$

## Fatoração LU : Calculando os valores de $\mathbf{y}$

$$L\mathbf{y} = \mathbf{b}$$

$$y_i = b_i - \sum_{j=1}^{i-1} L_{ij} y_j \quad \text{para } i = 1, 2, \dots, n$$



```
for(i=0; i<n; i++)  
{  
    y[i] = b[i];  
    for(j=0; j<=i-1; j++)  
    {  
        y[i] = y[i] - L[i][j] * y[j];  
    }  
}
```

*forward  
substitution*



# Fatoração LU : Calculando os valores de $\mathbf{x}$


$$U\mathbf{x} = \mathbf{y}$$

$$\begin{bmatrix} 2 & 3 & -1 \\ 0 & -2 & -1 \\ 0 & 0 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ -7 \\ 15 \end{bmatrix}$$

$$\begin{aligned} 2x_1 + 3x_2 - 1x_3 &= 5 \\ 0x_1 - 2x_2 - 1x_3 &= -7 \\ 0x_1 + 0x_2 + 5x_3 &= 15 \end{aligned}$$

$$\begin{aligned} 2x_1 + 3x_2 - 1x_3 &= 5 \\ -2x_2 - 1x_3 &= -7 \\ 5x_3 &= 15 \end{aligned}$$

backward  
substitution



$$\begin{aligned} x_3 &= \frac{15}{5} \\ x_2 &= \frac{-7 - (-1x_3)}{-2} \\ x_1 &= \frac{5 - (3x_2 - 1x_3)}{2} \end{aligned}$$



$$x_i = \frac{y_i - \sum_{j=i+1}^n U_{ij}x_j}{U_{ii}}$$

para  $i = n, n-1, n-2, \dots, 0$

$$x_3 = \frac{15}{5} \Rightarrow x_3 = 3$$

$$-2x_2 - 1 * 3 = -2x_2 - 3 = -7 \Rightarrow -x_2 = \frac{-7+3}{2} \Rightarrow x_2 = 2$$

$$2x_1 + 3 * 2 - 1 * 3 = 2x_1 + 6 - 3 = 5 \Rightarrow x_1 = \frac{5-3}{2} \Rightarrow x_1 = 1$$

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

## Fatoração LU : Calculando os valores de $\mathbf{x}$

$$U\mathbf{x} = \mathbf{y}$$

$$x_i = \frac{y_i - \sum_{j=i+1}^n U_{ij}x_j}{U_{ii}} \text{ para } i = n, n-1, n-2, \dots, 0$$



```
for(i=n-1; i>=0; i--)  
{  
    x[i] = y[i];  
  
    for(j=n-1; j>=i+1; j--)  
    {  
        x[i] = x[i] - U[i][j] * x[j];  
    }  
  
    x[i] = x[i] / U[i][i];  
}
```

*backward  
substitution*



# Fatoração LU – Paralelizando o Algoritmo

- Como pode ser observado nos slides anteriores, para resolver o sistema linear utilizando a fatoração LU, tem-se basicamente quatro algoritmos
  - **Inicialização das matrizes  $L$  e  $U$ :**  
Não há a necessidade de paralelizar porque neste algoritmo há apenas atribuições de valores escalares
  - **Cálculo das matrizes  $L$  e  $U$ :**  
Neste caso é necessário paralelizar porque há cálculos aritméticos, o que aumenta o tempo computacional
  - **Cálculo do vetor  $y$ :**  
Não é possível a paralelização porque o cálculo de cada  $y_i$  depende do  $y_{i-1}$
  - **E finalmente, cálculo do vetor  $x$ :**  
Não é possível a paralelização porque o cálculo de cada  $x_i$  depende do  $x_{i-1}$

# Fatoração LU – Paralelizando o Algoritmo

- Algoritmo para Cálculo das Matrizes  $L$  e  $U$

```
//--- calculando as matrizes L e U
for(k=0; k<n-1; k++)
{
    for(i=k+1; i<n; i++)
    {
        numerador = U[i][k];
        denominador = U[k][k];

        L[i][k] = numerador/denominador;

        for(j=k; j<n; j++)
        {
            U[i][j] = U[i][j] - L[i][k] * U[k][j];
        }
    }
} // for i
} // for k
```

→ Não é possível paralelizar a partir deste for, porque o objetivo dele é zerar os elementos abaixo da diagonal principal coluna por coluna

E sendo assim, para zerar os elementos abaixo da diagonal principal da coluna  $k_i$ , é necessário primeiramente zerar todos os elementos abaixo da diagonal principal da coluna  $k_{i-1}$

# Fatoração LU – Paralelizando o Algoritmo

- Algoritmo para Cálculo das Matrizes  $L$  e  $U$  – **PARALELIZADO**

```
for(k=0; k<n-1; k++)  
{  
    #pragma omp parallel for private (i, j, numerador, denominador)  
    for(i=k+1; i<n; i++)  
    {  
        numerador = U[i][k];  
        denominador = U[k][k];  
  
        L[i][k] = numerador/denominador;  
  
        for(j=k; j<n; j++)  
        {  
            U[i][j] = U[i][j] - L[i][k] * U[k][j];  
        }  
    }  
} // for i  
  
} // for k
```

A paralelização do algoritmo é feita a partir deste *for*, porque o cálculo dos elementos  $L_{ik}$  e  $U_{ij}$  pode ser feito de forma totalmente paralela e independente.

Observa-se também que em uma mesma *#pragma* iniciou-se a região paralela e aplicou-se o construtor de trabalho *for*.

Isso é possível fazer com qualquer construtor de trabalho, desde que nesta região paralela não se utilize outro construtor de trabalho.

Nota-se que foi necessário privatizar as variáveis  $i$ ,  $j$ ,  $numerador$  e  $denominador$  em cada *thread*.