

# SuperComputação

Aula 12 – Introdução a paralelismo

2021 – Engenharia

André Filipe M. Batista <[andrefmb@insper.edu.br](mailto:andrefmb@insper.edu.br)>

# Resolução de problemas

- Heurísticas
- Busca local
- Busca exaustiva
  - Branch and Bound (propriedades do problema)

# Solução de alto desempenho

## 1. Algoritmos eficientes

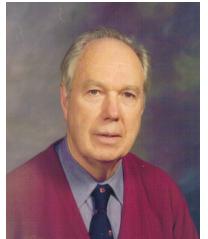
## 2. Implementação eficiente

- Cache, paralelismo de instrução
- Linguagem de programação adequada

## 3. Paralelismo

# Paralelismo

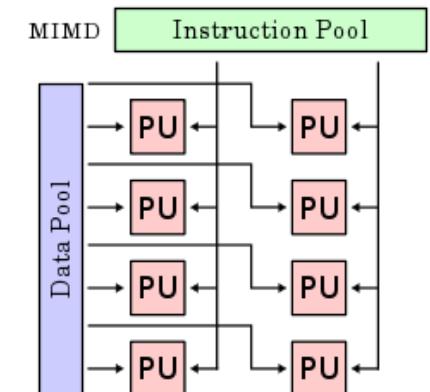
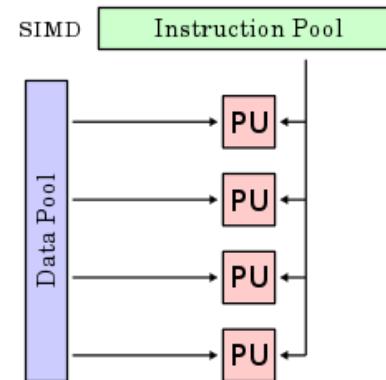
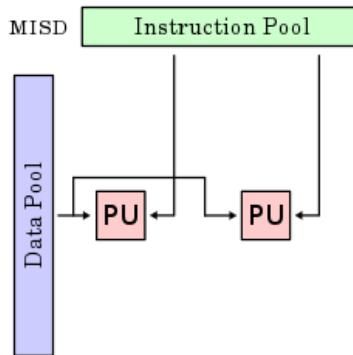
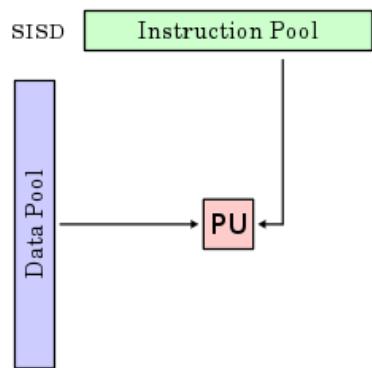
- Consiste no uso de múltiplos processadores, simultaneamente, para resolver um mesmo problema
- Tem por objetivo o aumento do desempenho, isto é, a redução do tempo necessário para resolver um problema
- Usamos paralelismo normalmente por 2 motivos:
  - 1 – Problemas cada vez mais complexos e/ou maiores
  - 2 – Clock dos processadores se aproximamento dos limites ditados pela física



# Taxonomia de Flynn

Michael J. Flynn

- É uma forma de classificar computadores paralelos
- Proposta por Flynn, em 1972
- Baseia-se no fato de um computador executar uma sequência de instruções sobre uma sequência de dados



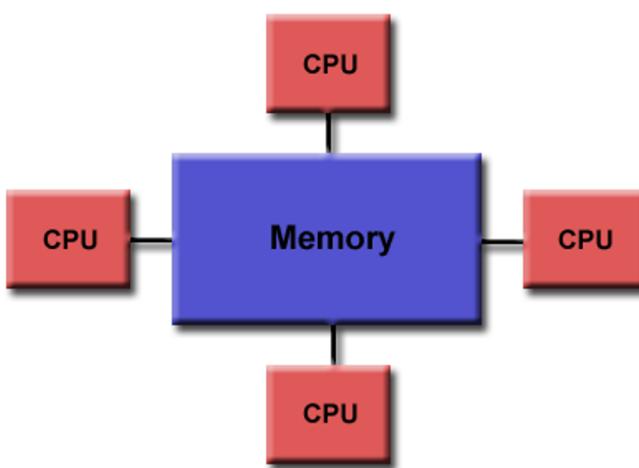
Von Newmann

Pipeline

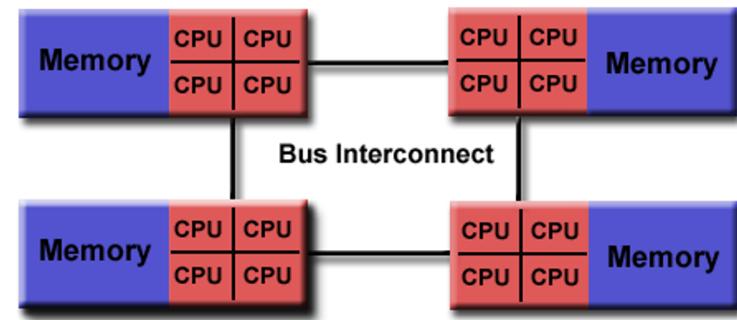
Array

Máquinas Paralelas  
(SMP, clusters e  
NUMA)

# Sistemas Multi-core

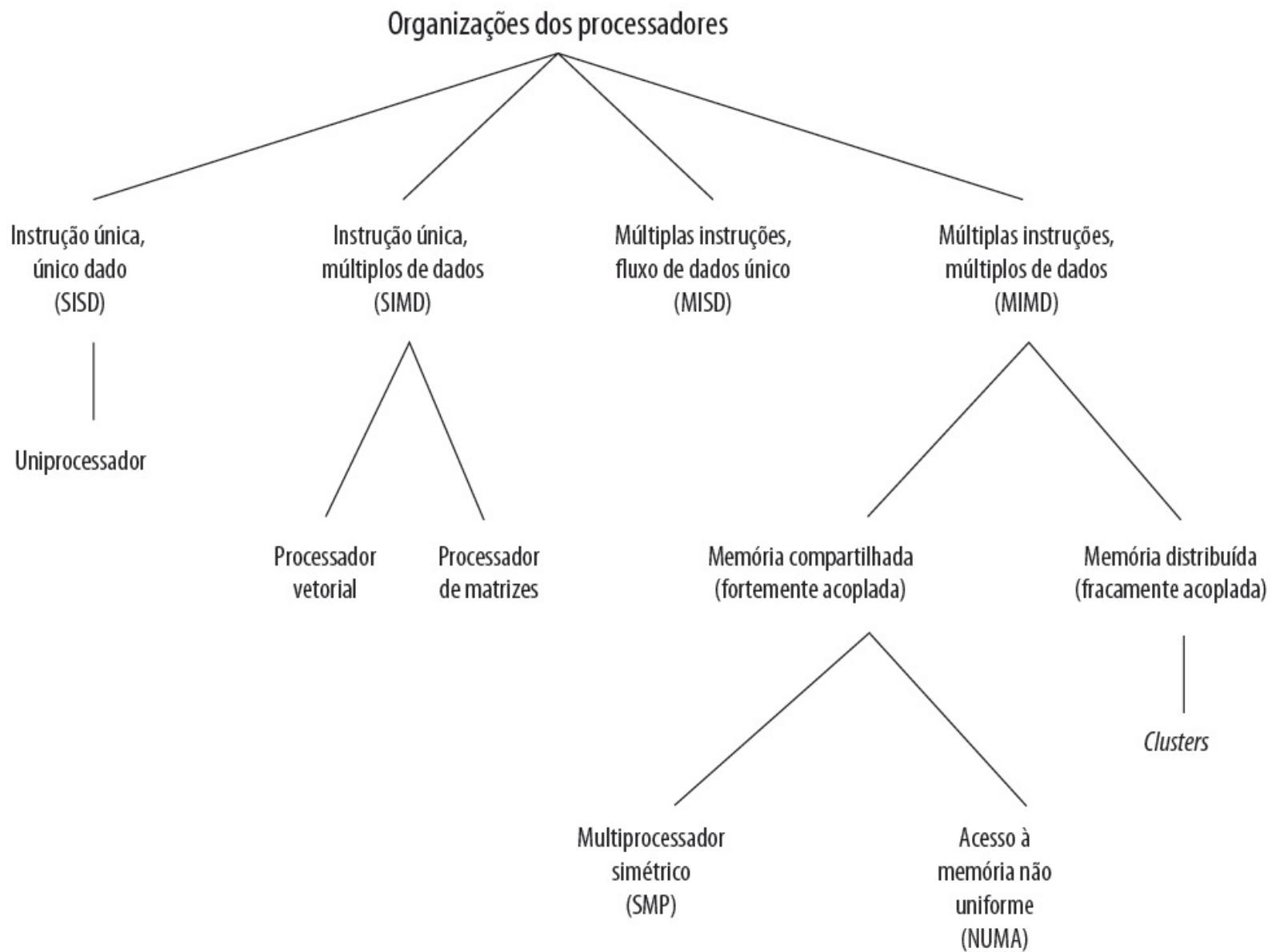


Uniform Memory Access



Non-Uniform Memory Access

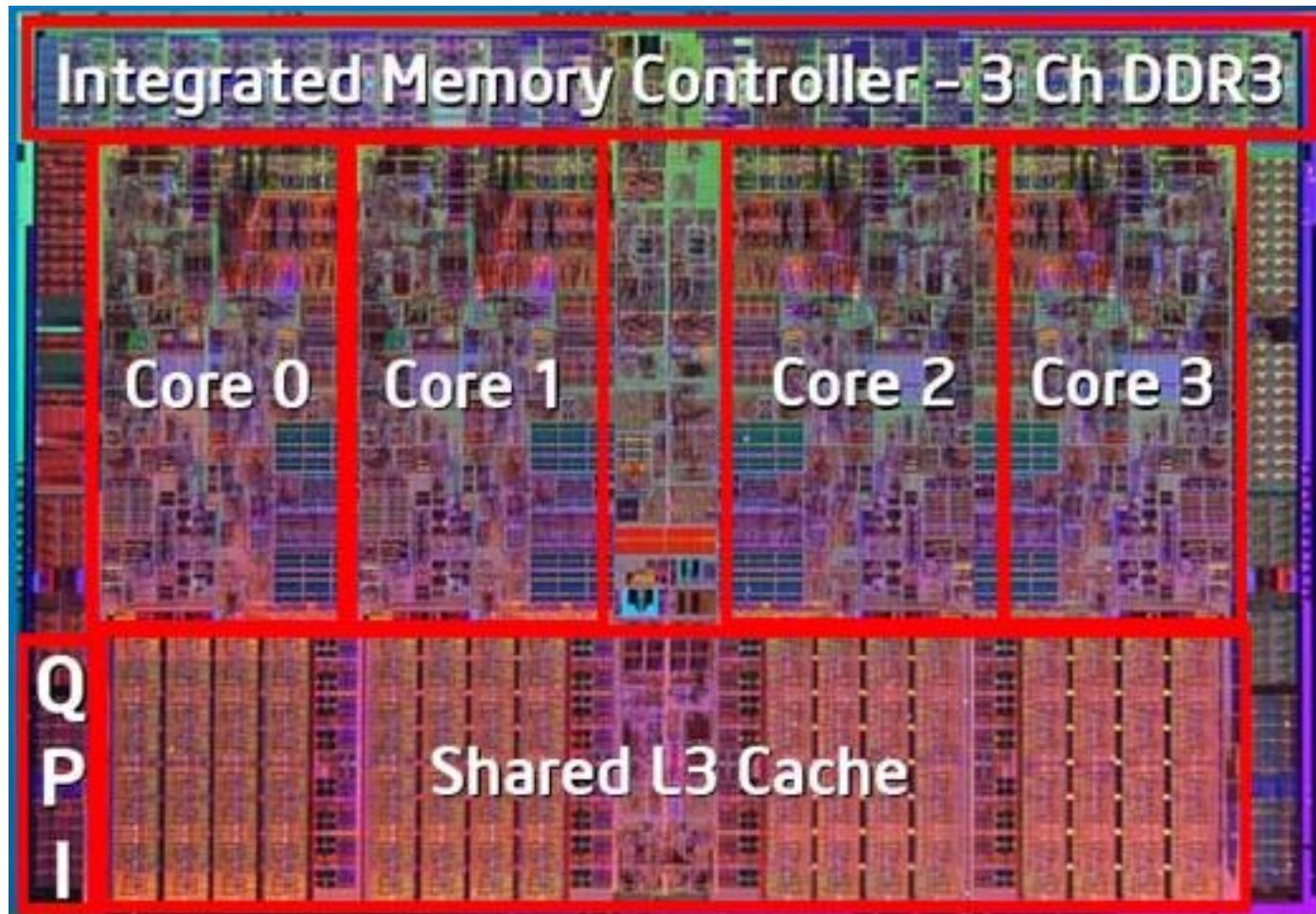
# Organização dos Processadores



# Multiprocessadores Simétricos

- Não faz muito tempo que todos os computadores pessoais continham um único processador de propósito geral
- À medida que a demanda por desempenho aumenta e o custo dos processadores continua a baixar, os fabricantes têm introduzido sistemas com uma organização SMP
- O termo SMP refere-se a uma arquitetura de hardware computacional e também ao comportamento do sistema operacional que reflete essa arquitetura

# Exemplo – Intel i7



# **Discussão I: qual expectativa de melhoria de velocidade?**

# Exemplo 1 (supondo 8 cores)

```
vector<double> dados;
vector<double> resultados;
for (int i = 0; i < dados.size(); i++) {
    resultados[i] = funcao_complexa(dados[i]);
}
```

# Exemplo 1

```
vector<double> dados;
vector<double> resultados;
for (int i = 0; i < dados.size(); i++) {
    resultados[i] = funcao_complexa(dados[i]);
}
```

**Tempo total dividido por 8!**

## Exemplo 2

```
vector<double> dados;
vector<double> resultados;
resultados[0] = 0;
for (int i = 1; i < dados.size(); i++) {
    resultados[i] = funcao_complexa(dados[i], resultados[i-1]);
}
```

## Exemplo 2

```
vector<double> dados;
vector<double> resultados;
resultados[0] = 0;
for (int i = 1; i < dados.size(); i++) {
    resultados[i] = funcao_complexa(dados[i], resultados[i-1]);
}
```

**Nenhum ganho! Depende da iteração anterior :(**

# Conceito 1: Dependência

Um loop tem uma **dependência** de dados sua execução correta depende da ordem de sua execução.

Isto ocorre quando **uma iteração depende de resultados calculados em iterações** anteriores.

Quando não existe nenhuma dependência em um loop ele é dito **ingenuamente paralelizável**.

# Exemplo 3

```
vector<double> dados;
vector<double> resultados1;
vector<double> resultados2;
resultados1[0] = resultados2[0] 0;
for (int i = 1; i < dados.size(); i++) {
    resultados1[i] = funcao_complexa(dados[i], resultados1[i-1]);
    resultados2[i] = funcao_complexa2(dados[i], resultados2[i-1]);
}
```

# Exemplo 3

```
vector<double> dados;
vector<double> resultados1;
vector<double> resultados2;
resultados1[0] = resultados2[0] 0;
for (int i = 1; i < dados.size(); i++) {
    resultados1[i] = funcao_complexa(dados[i], resultados1[i-1]);
    resultados2[i] = funcao_complexa2(dados[i], resultados2[i-1]);
}
```

**Podemos fazer resultados1 e resultados2 em paralelo!**

## Conceito 2: Paralelismo

**Paralelismo de dados:** faço em paralelo a mesma operação (lenta) para todos os elementos em um conjunto de dados (grande).

**Paralelismo de tarefas:** faço em paralelo duas (ou mais) tarefas independentes. Se houver dependências quebro em partes independentes e rodo em ordem.

# Exemplo 4

```
std::vector<double> dados;  
  
le_dados_do_disco(dados); // demora 10 segundos  
// dados.size() == 100  
  
for (int i = 0; i < dados.size(); i++) {  
    operacao_complexa3(dados[i]); // demora 0,1 segundo  
}
```

**Quanto tempo o programa demora?**

**Existem relações de dependência?**

**Qual a expectativa de tempo para um programa paralelo?**

## Conceito 3: Lei de Amdahl

Dada uma tarefa que dura  $X$  horas, sendo que  $Y$  horas correspondem a trabalho que pode ser paralelizado, o número máximo de vezes que podemos acelerá-la é

$$\frac{1}{(1 - p)}$$

onde  $p = \frac{Y}{X}$

# Resumo

1. Paralelizar significa rodar código sem dependências simultaneamente
2. Paralelismo de dados: mesma tarefa, dados diferentes
3. Paralelismo de tarefas: heterogêneo
4. Existem tarefas inherentemente sequenciais
5. Ganhos são limitados a partes do programa

# **OpenMP**

# Paralelismo Multi-core

## Threads:

- Compartilham memória
- Sincronização de acessos

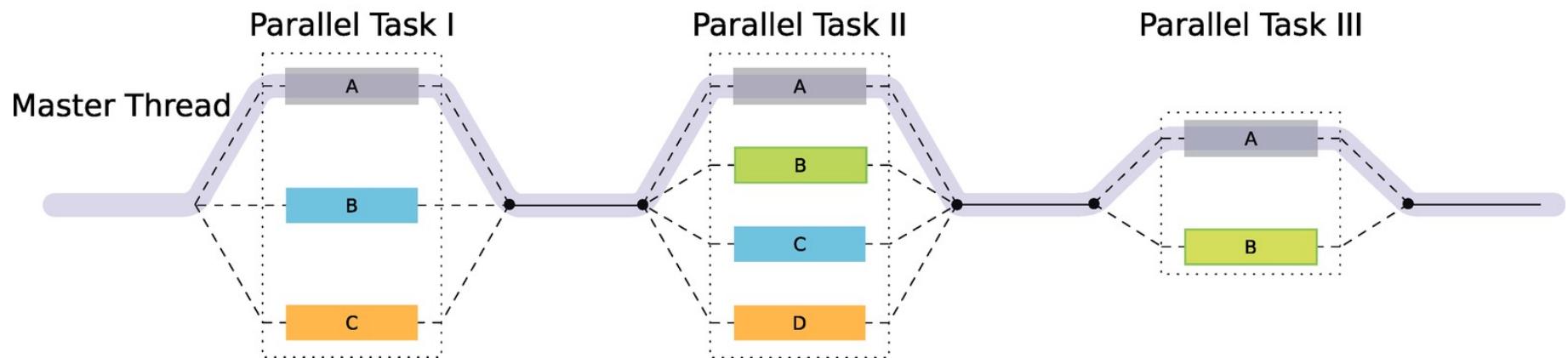
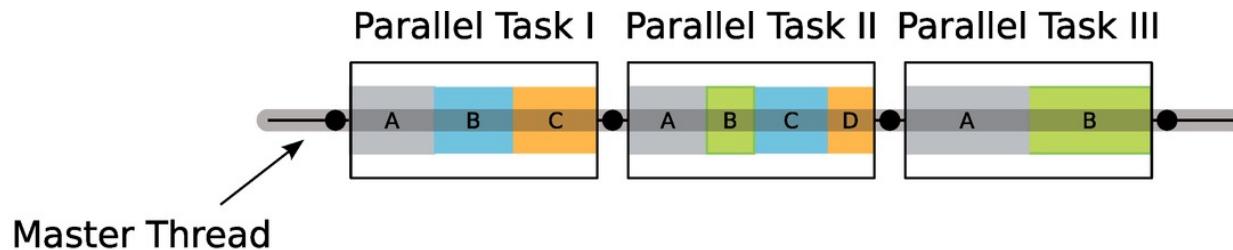
## Processos:

- Troca de mensagens
- Possível distribuir em vários nós

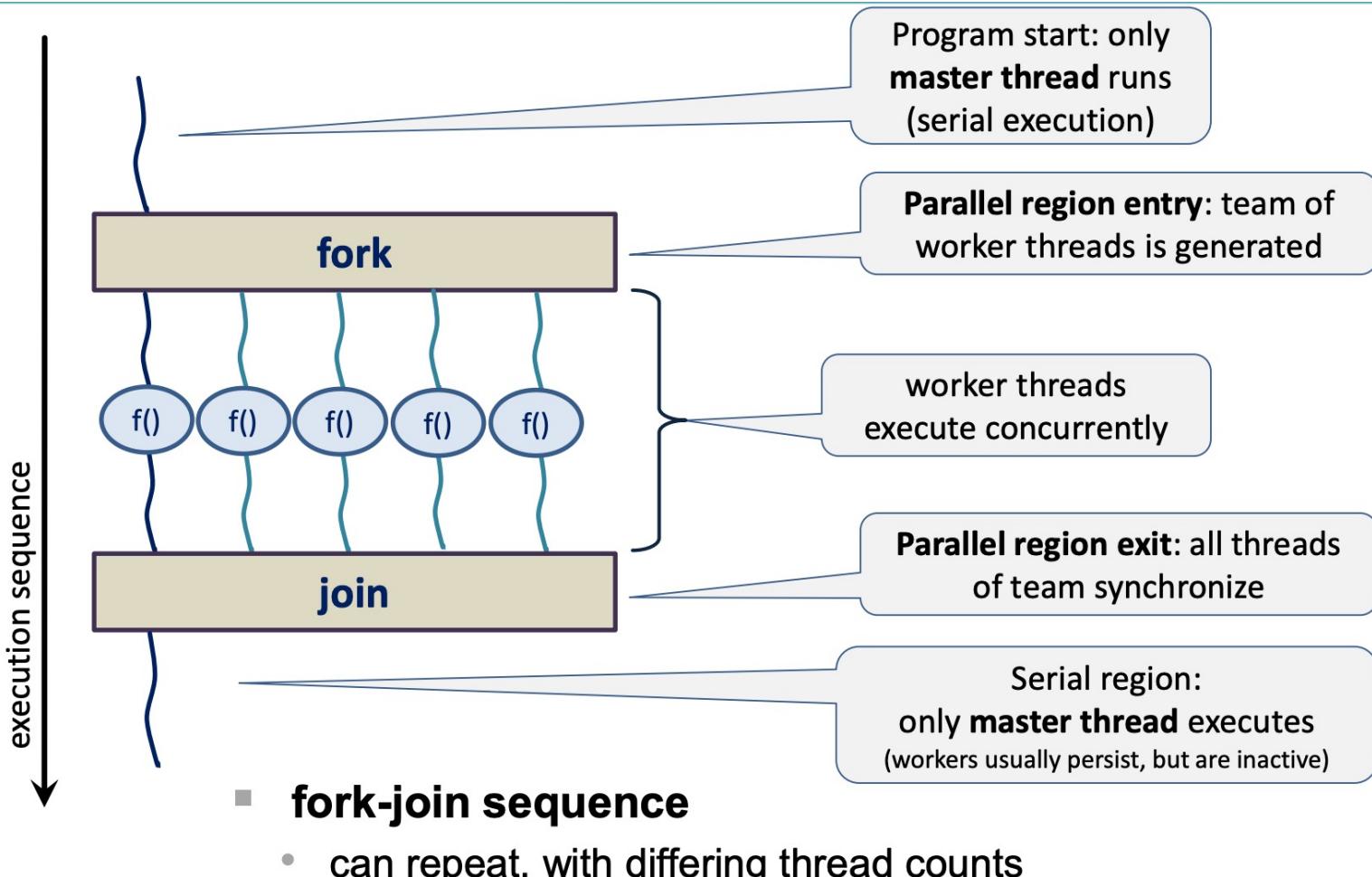
# OpenMP

- Conjunto de extensões para C/C++ e Fortran
- Fornece construções que permitem paralelizar código em ambientes multi-core
- Padroniza práticas SMP + SIMD + Sistemas heterogêneos (GPU/FPGA)
- Idealmente funciona com mínimo de modificações no código sequencial

# OpenMP



# OpenMP



# Fontes importantes

## A brief Introduction to parallel programming

**Tim Mattson**

**Intel Corp.**

[timothy.g.mattson@intel.com](mailto:timothy.g.mattson@intel.com)

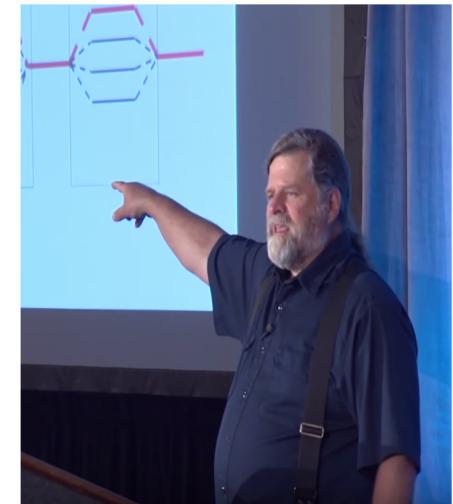
### Vídeos:

<https://www.youtube.com/watch?v=pRtTIW9-Nr0>

<https://www.youtube.com/watch?v=LRsQHDAqPHA>

<https://www.youtube.com/watch?v=dK4PITrQtjY>

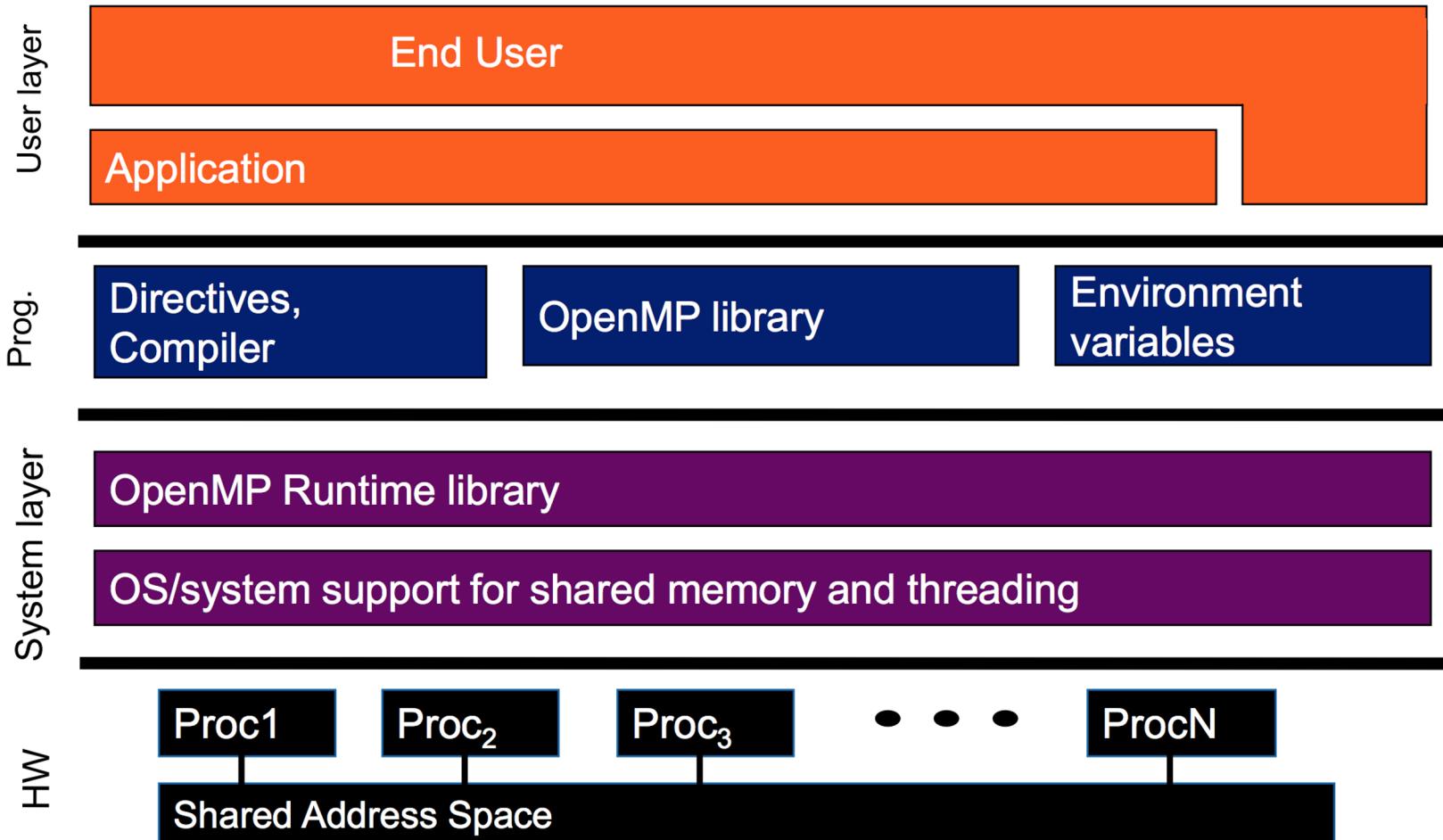
[https://www.youtube.com/watch?v=WvoMpG\\_QvBU](https://www.youtube.com/watch?v=WvoMpG_QvBU)



### Slides:

[http://extremecomputingtraining.anl.gov/files/2016/08/Mattson\\_830aug3\\_HandsOnIntro.pdf](http://extremecomputingtraining.anl.gov/files/2016/08/Mattson_830aug3_HandsOnIntro.pdf)

# OpenMP (host / NUMA)



# OpenMP - sintaxe

## Diretivas de compilação

```
#include <omp.h>
#pragma omp construct [params]
```

## Aplicadas a um bloco de código

limitado diretamente por { }

```
for (...) { }
```

## Com join implícito

# OpenMP - sintaxe

```
// Arquivo interface da biblioteca OpenMP para C/C++
#include <omp.h>

// retorna o identificador da thread.
int omp_get_thread_num();

// indica o número de threads a executar na região paralela.
void omp_set_num_threads(int num_threads);

// retorna o número de threads que estão executando no momento.
int omp_get_num_threads();
```

# OpenMP - sintaxe

most commonly used subset

Name	Result type	Purpose
omp_set_num_threads (int num_threads)	none	number of threads to be created for subsequent parallel region
omp_get_num_threads()	int	number of threads in <b>currently executing</b> region
omp_get_max_threads()	int	maximum number of threads that can be created for a subsequent parallel region
omp_get_thread_num()	int	thread number of calling thread (zero based) in <b>currently executing</b> region
omp_get_num_procs()	int	number of processors available
omp_get_wtime()	double	return wall clock time in seconds since some (fixed) time in the past
omp_get_wtick()	double	resolution of timer in seconds

# OpenMP - sintaxe

```
// Cria a região paralela. Define variáveis privadas e
compartilhadas entre as threads.
#pragma omp parallel private(...) shared(...)
{ // Obrigatoriamente na linha de baixo.

// Apenas a thread mais rápida executa.
#pragma omp single
}

}
```

# OpenMP - sintaxe

Código sequencial

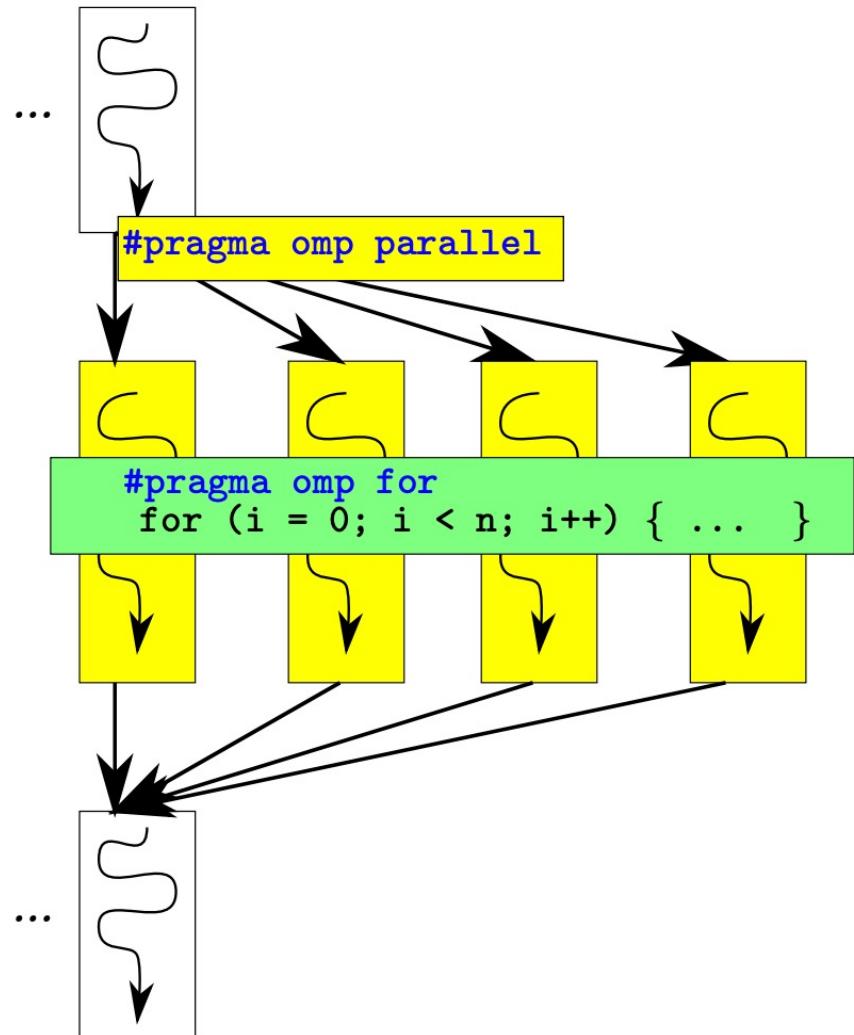
```
for(i = 0; i < N; i++)
    a[i] = a[i] + b[i];
```

Região OpenMP parallel

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if(id == Nthrds-1) iend = N;
    for(i = istart; i < iend; i++)
        a[i] = a[i] + b[i];
}
```

# OpenMP - sintaxe

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if(id == Nthrds-1) iend = N;
    for(i = istart; i < iend; i++)
        a[i] = a[i] + b[i];
}
```



# OpenMP - sintaxe

Código sequencial

```
for(i = 0; i < N; i++)
    a[i] = a[i] + b[i];
```

Região OpenMP parallel

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if(id == Nthrds-1) iend = N;
    for(i = istart; i < iend; i++)
        a[i] = a[i] + b[i];
}
```

Região paralela OpenMP  
com uma construção de  
divisão de laço

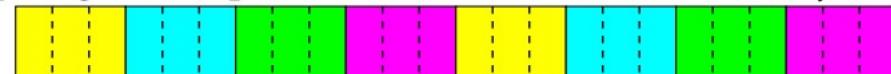
```
#pragma omp parallel
#pragma omp for
for(i = 0; i < N; i++) a[i] = a[i] + b[i];
```

# OpenMP

```
#pragma omp for schedule(static)
```



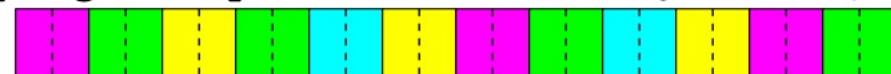
```
#pragma omp for schedule(static,3)
```



```
#pragma omp for schedule(dynamic)
```



```
#pragma omp for schedule(dynamic,2)
```



```
#pragma omp for schedule(guided)
```



```
#pragma omp for schedule(guided,2)
```



# OpenMP - Reduções

```
for(i=1; i<=n; i++){
    sum = sum + a[i];
}
```

```
#pragma omp parallel for reduction(+:sum)
{
    for(i=1; i<=n; i++){
        sum = sum + a[i];
    }
}
```

# OpenMP

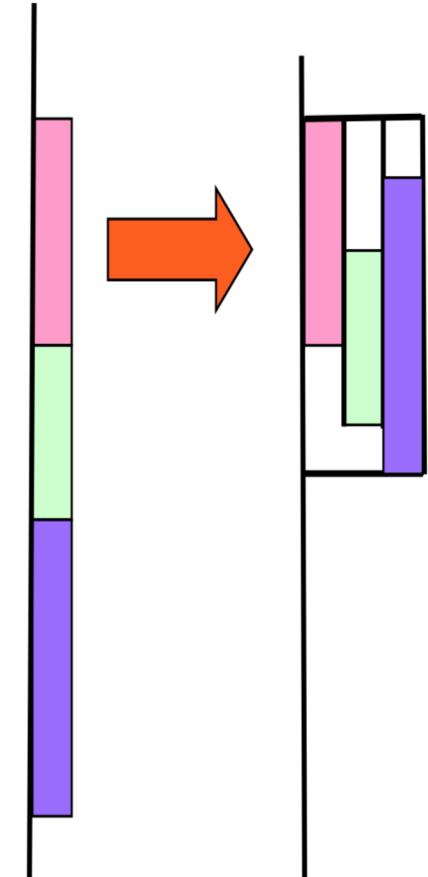
- Demonstração em sala de aula de alguns códigos e suas otimizações com OpenMP

# **Parte II**

## **Tasks**

# O que são tarefas?

- A tarefa é definida em um bloco estruturado de código
- Tarefas podem ser aninhadas: isto é, uma tarefa pode gerar novas tarefas
- Cada thread pode ser alocada para rodar uma tarefa
- Não existe ordenação no início das tarefas
- Tarefa são unidades de trabalho independentes



**Serial**

**Paralela**

# Tarefas em OpenMP

**#pragma omp task[clauses]**

```
#pragma omp parallel  
{
```

```
    #pragma omp master  
    {  
        #pragma omp task  
            func1();  
        #pragma omp task  
            func2();  
        #pragma omp task  
            func3();  
    }
```

```
}
```

Crie um conjunto de threads

Thread 0 organiza as tarefas

Tarefas executadas por alguma thread em alguma ordem

Todas as tarefas devem ser concluídas  
antes que esta barreira seja liberada

# Estrutura Padrão

```
#include <stdio.h>
#include <omp.h>
int main()
{ printf("I think");
#pragma omp parallel
{
#pragma omp single
{
#pragma omp task
printf(" car");
#pragma omp task
printf(" race");
}
}
printf("s");
printf(" are fun!\n");
}
```

# Esperando (taskwait)

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        fred();
        #pragma omp task
        daisy();
        #pragma taskwait
        #pragma omp task
        billy();
    }
}
```

fred() and daisy()  
must complete before  
billy() starts

# Fibonacci

É possível criar tarefas para esse problema?

Altere a função fib para que ela chame uma task antes de calcular x e outra task antes de y.

Lembre-se que quem chama fib (na função main) também precisa de uma task omp do tipo single (a task que dispara as outras tasks).

Um ponto importante, se  $n < 20$ , calcule o Fibonacci sem o OpenMP.

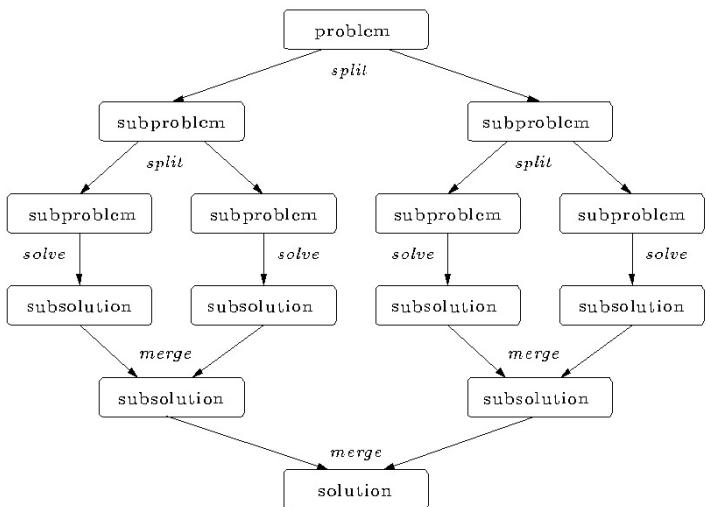
```
#include<iostream>
#include<omp.h>
using namespace std;

int fib (int n) {
    int x, y;
    if(n<2) return n;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
}

int main() {
    int NW = 1000;
    float time = omp_get_wtime();
    fib(NW);
    time = omp_get_wtime() - time;
    cout << "Tempo em segundos : " << time << endl;
}
```

# Fibonacci

## Resolução



```
#include<iostream>
#include<omp.h>
#include <math.h>
using namespace std;

int fib (int n) {
    int x, y;
    if(n<2) return n;
    if (n < 20) {
        return fib(n-1) + fib(n-2);
    } else {
        #pragma omp task shared(x)
        x = fib(n-1);
        #pragma omp task shared(y)
        y = fib(n-2);
        #pragma omp taskwait
        return x+y;
    }
}

int main() {
    int NW = 50;
    float time;
    time = omp_get_wtime();
    #pragma omp parallel
    {
        #pragma omp single
        fib(NW);
    }
    time = omp_get_wtime() - time;
    cout << "Tempo em segundos : " << time << endl;
}
```

# **Atividade prática**

## **Primeiro contato com OpenMP**

1. Executar código paralelo em CPU
2. API do OpenMP para trabalhar com regiões paralelas e tarefas

# Insper

[www.insper.edu.br](http://www.insper.edu.br)