Insper

# Supercomputação

# Aula - 11

- Paralelismo

# Recaptulando...

- Soluções de alto desempenho
- (1) Algoritmos eficientes
- (2) Implementação eficiente (cache, por exemplo)
- (3) Paralelismo

# Paralelismo

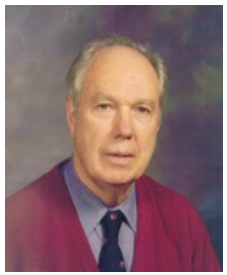
- Consiste no uso de múltiplos processadores, simultaneamente, para resolver um problema

- Tem por objetivo o aumento do desempenho, i.e., a redução do tempo necessário para resolver um problema

# Paralelismo

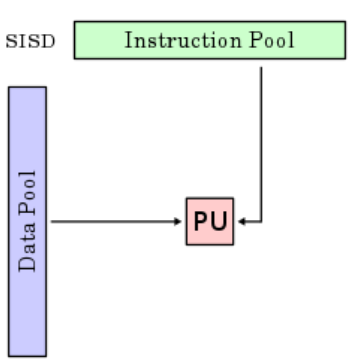
- Usamos paralelismo normalmente por 2 motivos
- (1) Problemas cada vez mais complexos e/ou maiores
- (2) Clock dos processadores se aproximando dos limites ditados pela física

# Taxonomia de Flynn

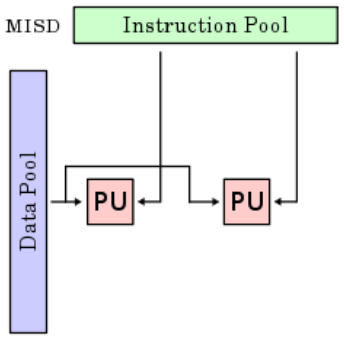


Michael J. Flynn

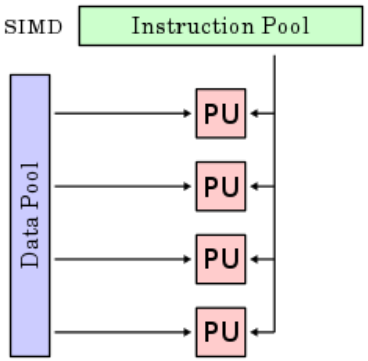
- É uma forma de classificar computadores paralelos
- Proposta por Flynn, em 1972
- Baseia-se no fato de um computador executar uma sequência de instruções sobre uma sequência de dados



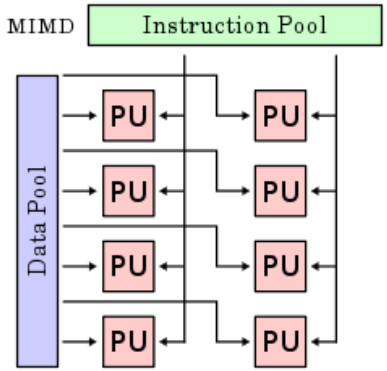

Von Newmann




Pipeline

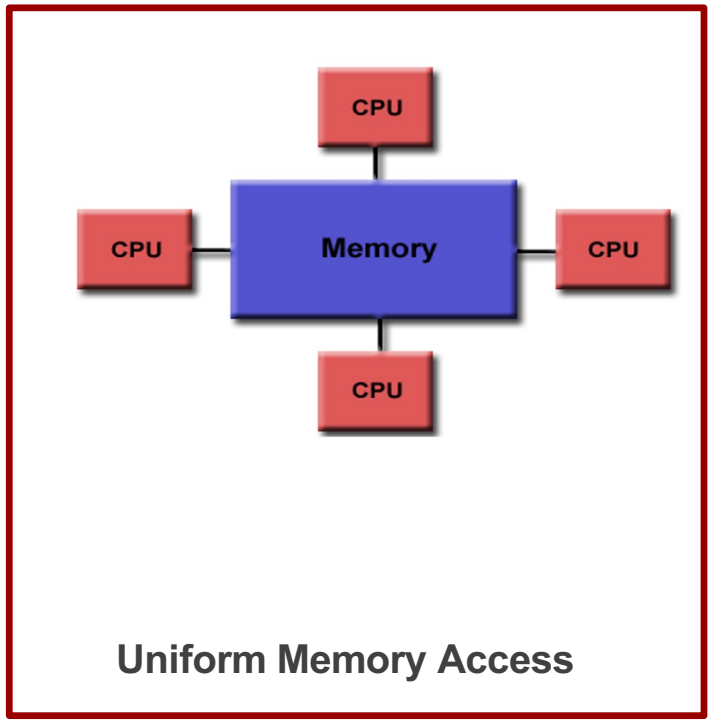



Array

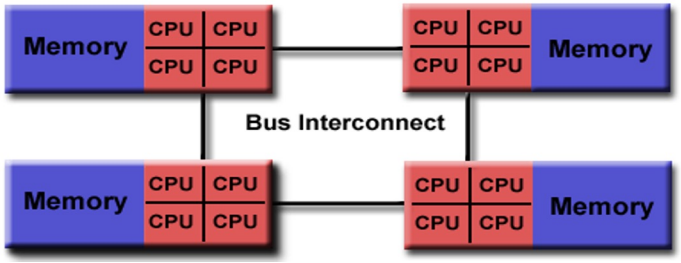



Máquinas Paralelas (SMP, clusters e NUMA)

# Sistemas Multi-core

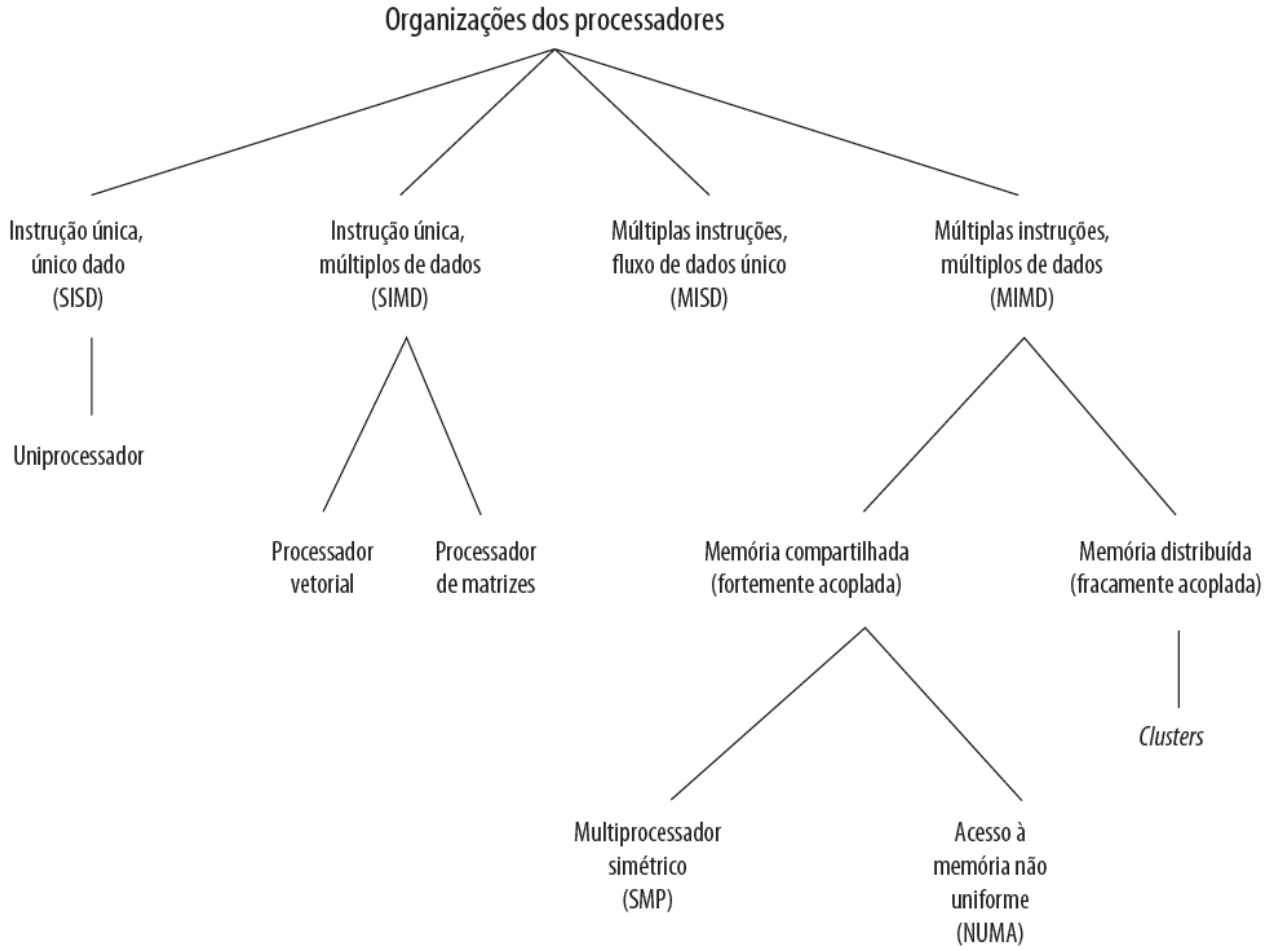



Uniform Memory Access




Non-Uniform Memory Access

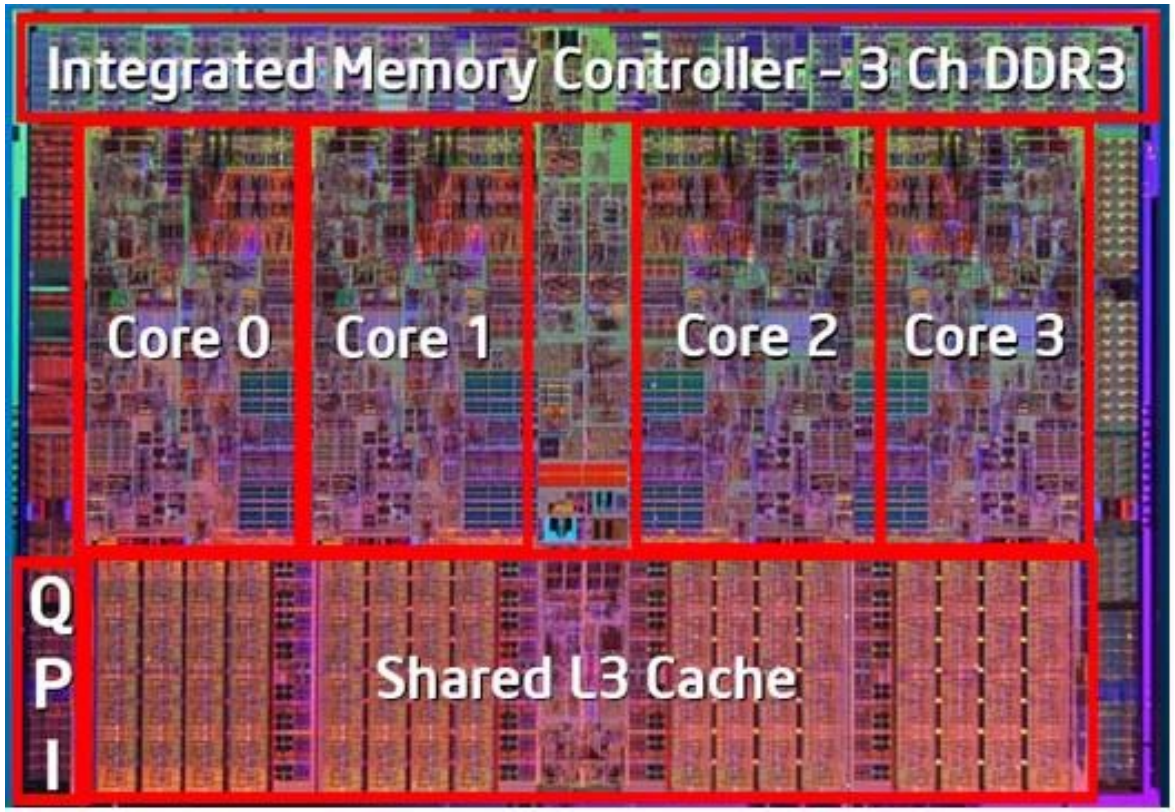
# Organização dos processadores



# Multiprocessadores simétricos

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

# Exemplo – Intel i7

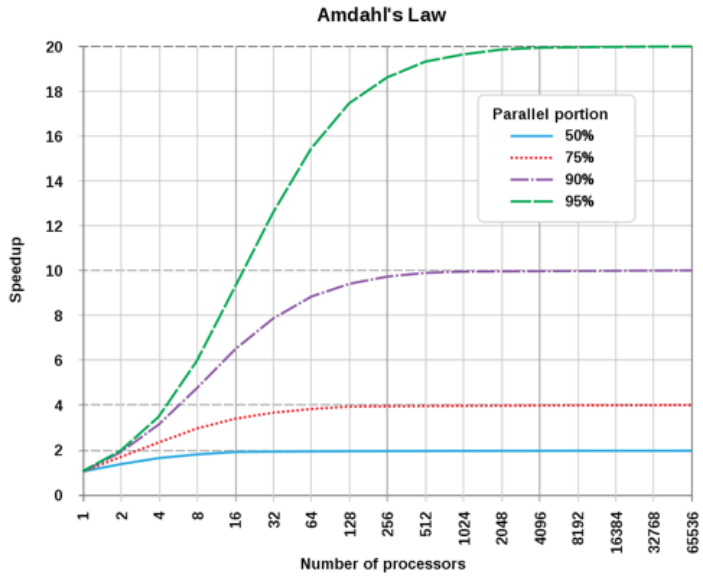


# Discussão

- **Discussão 1: qual a expectativa da melhoria de velocidade com paralelismo?**

# Discussão

- **Discussão 1: qual a expectativa da melhoria de velocidade com paralelismo?**



# Discussão

- **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]);
}
```

# Discussão

- **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]);
}
```

**– Tempo total / 8**

# Discussão

- **Exemplo 2 – Supondo 8 cores**

```
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]);
}
```

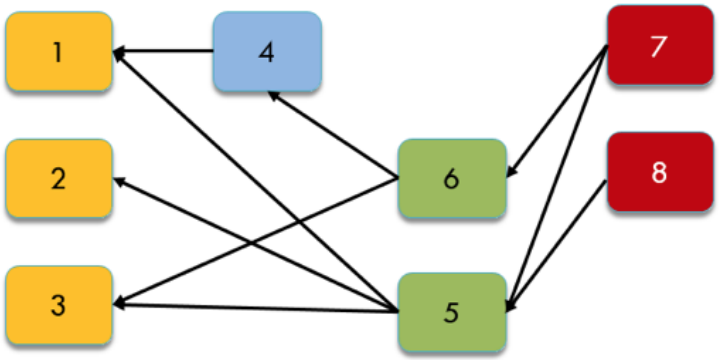
# Discussão

- **Exemplo 2 – Supondo 8 cores**

```
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 :(**

# Dependência




- É quando uma iteração depende de resultados calculados em iterações anteriores
- Quando não existe nenhuma dependência em um loop, por exemplo, dizemos que ele é ingenuamente paralelizável

# Discussão

- **Exemplo 3 – Supondo 8 cores**

```
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]);
}
```

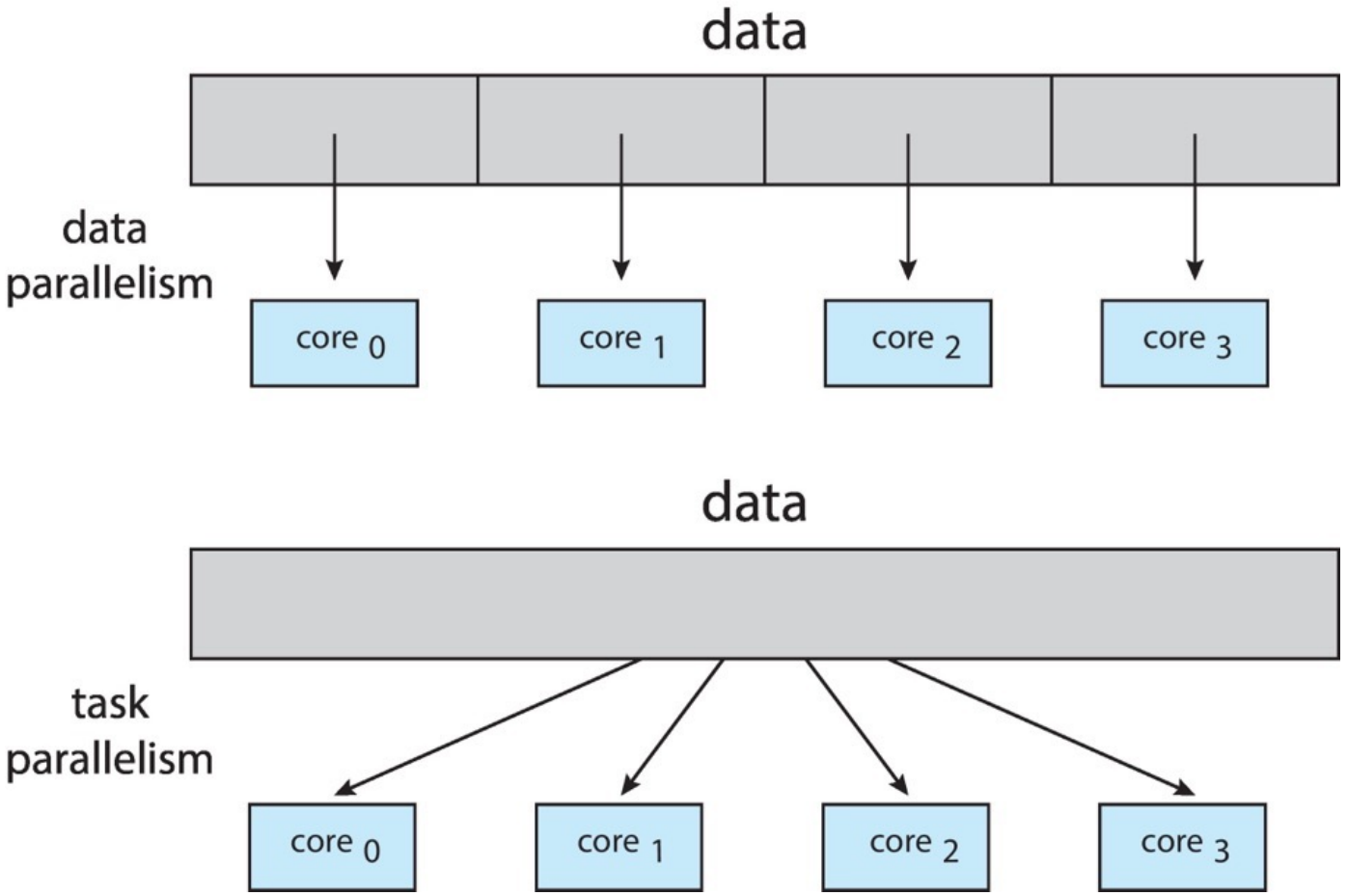
**Podemos calcular resultados1 e resultados2 paralelamente**

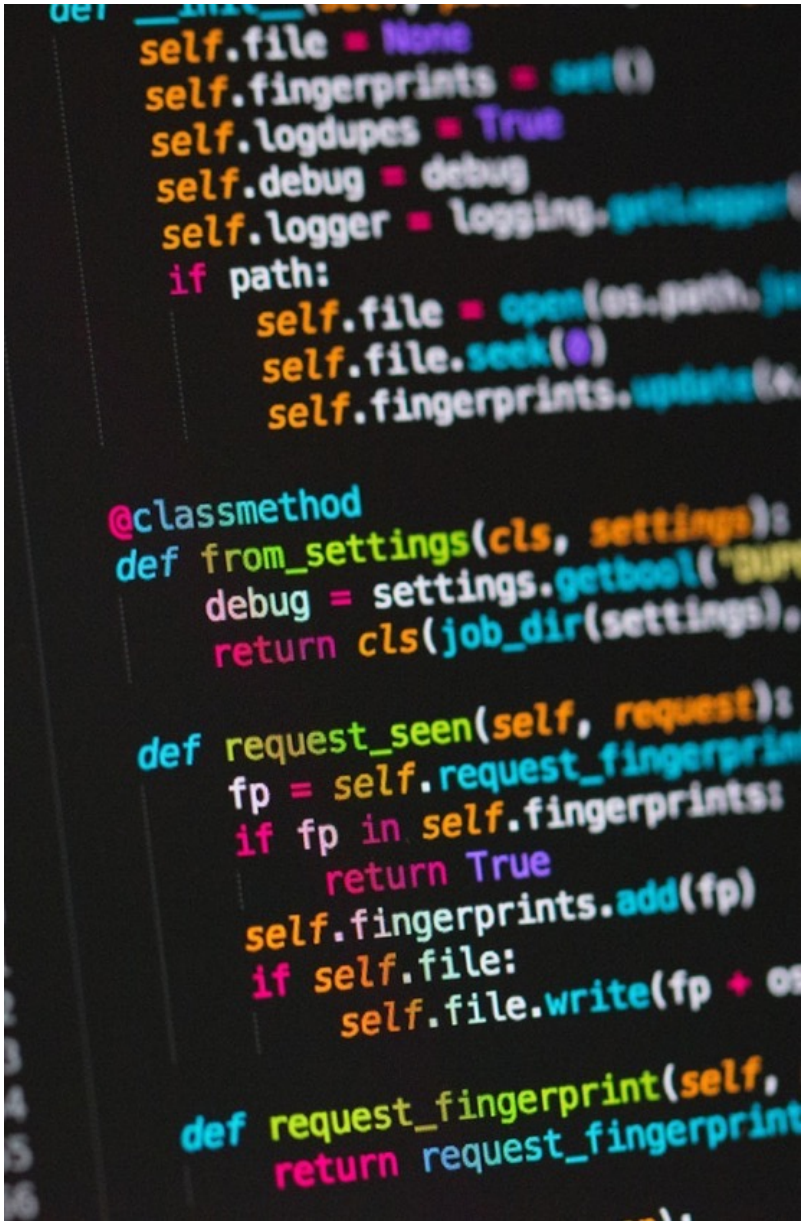
# Paralelismo

- Paralelismo de dados: a mesma operação (lenta) é executada para todos os elementos de um conjunto de dados (grande)

- Paralelismo de tarefas: duas ou mais tarefas independentes são executadas em paralelo. Se houver dependências, quebramos o problema em partes independents e rodamos na ordem adequada

# Paralelismo






# Paralelismo

- Vamos entender essa ideia de dependência executando códigos em Python por meio da biblioteca **Dask**, neste link

# Paralelismo - Resumo

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

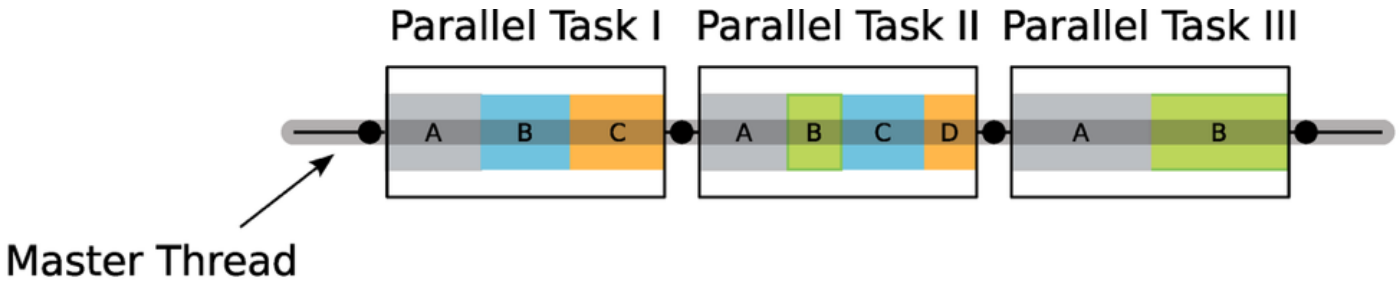
# OpenMP

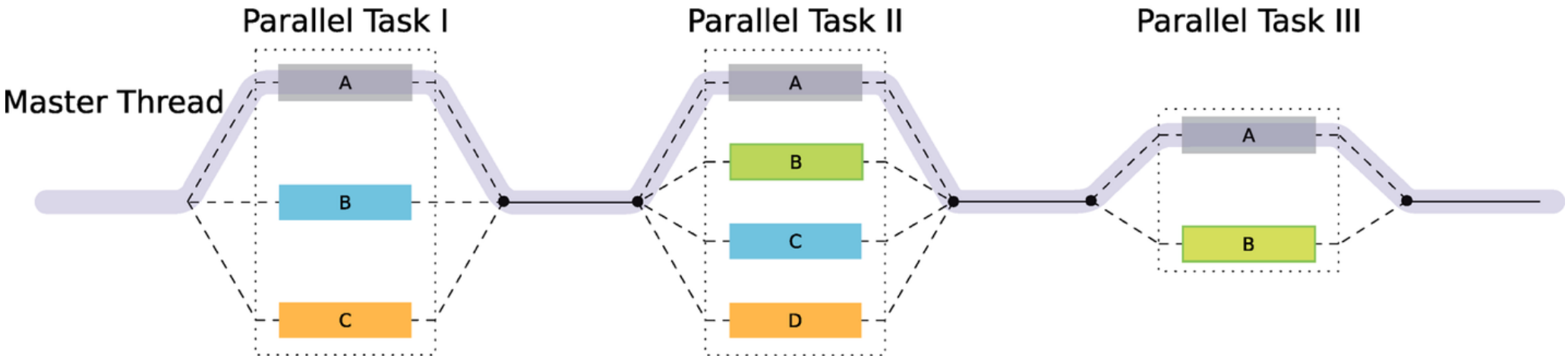
Sintaxe, Principais Conceitos

# OpenMP

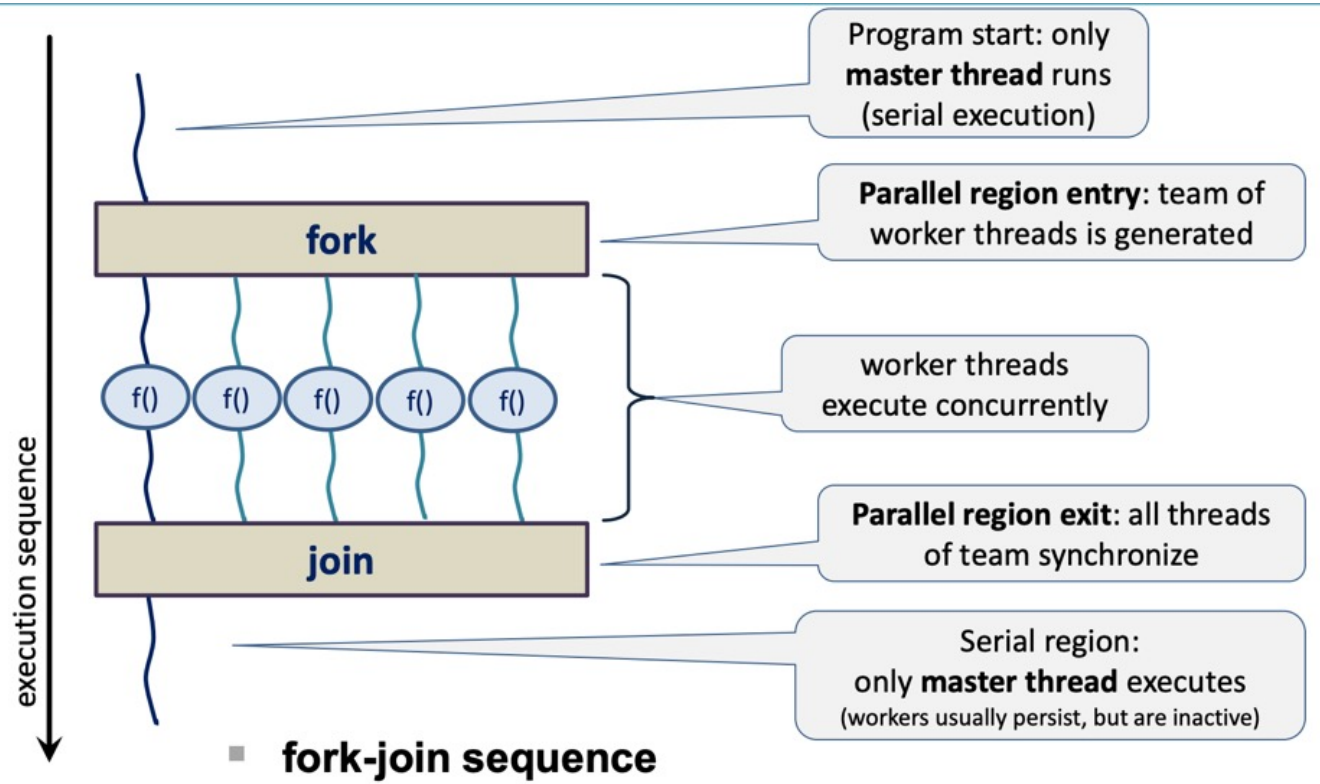
- Conjunto de extensões para C/C++ e Fortran
- Fornece construções que permitem paralelizar código em ambientes multi-core
- Padroniza práticas SMP + SIMD + sistemas heterogêneos (GPU/FPGA)
- Idealmente funciona com o mínimo de modificações no código sequêncial

# OpenMP


Parallel Task I
Parallel Task II
Parallel Task III
A
B
C
A
B
C
D
A
B
Master Thread


Parallel Task I
Parallel Task II
Parallel Task III
Master Thread
A
B
C
A
B
C
D
A
B

# OpenMP – Fork/Join

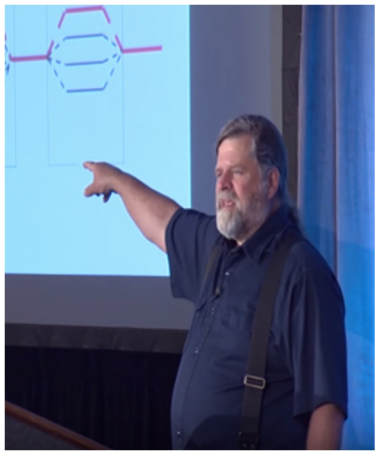



- **fork-join sequence**
  - can repeat, with differing thread counts

# OpenMP – onde aprender mais

## A brief Introduction to parallel programming

**Tim Mattson**
**Intel Corp.**
**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

Slides:

http://extremecomputingtraining.anl.gov/files/2016/08/Mattson_830aug3_HandsOnIntro.pdf

# OpenMP - compilação

To compile this with `gcc` we need to include the `-fopenmp` option:[1]

```
$ gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

# 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 **currently executing** 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 **currently executing** 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

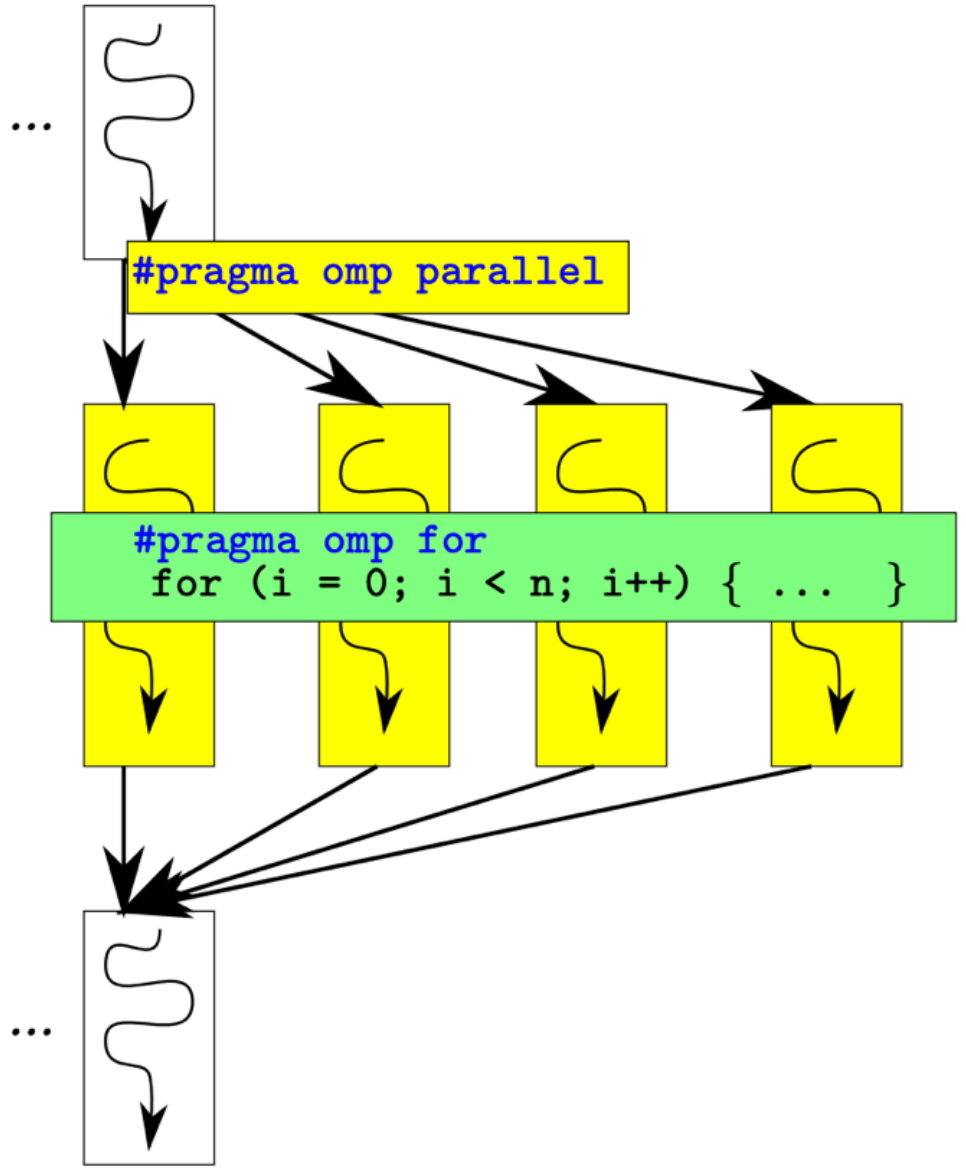
```
for(i = 0; i < N; i++)
  a[i] = a[i] + b[i];
```

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 - variáveis

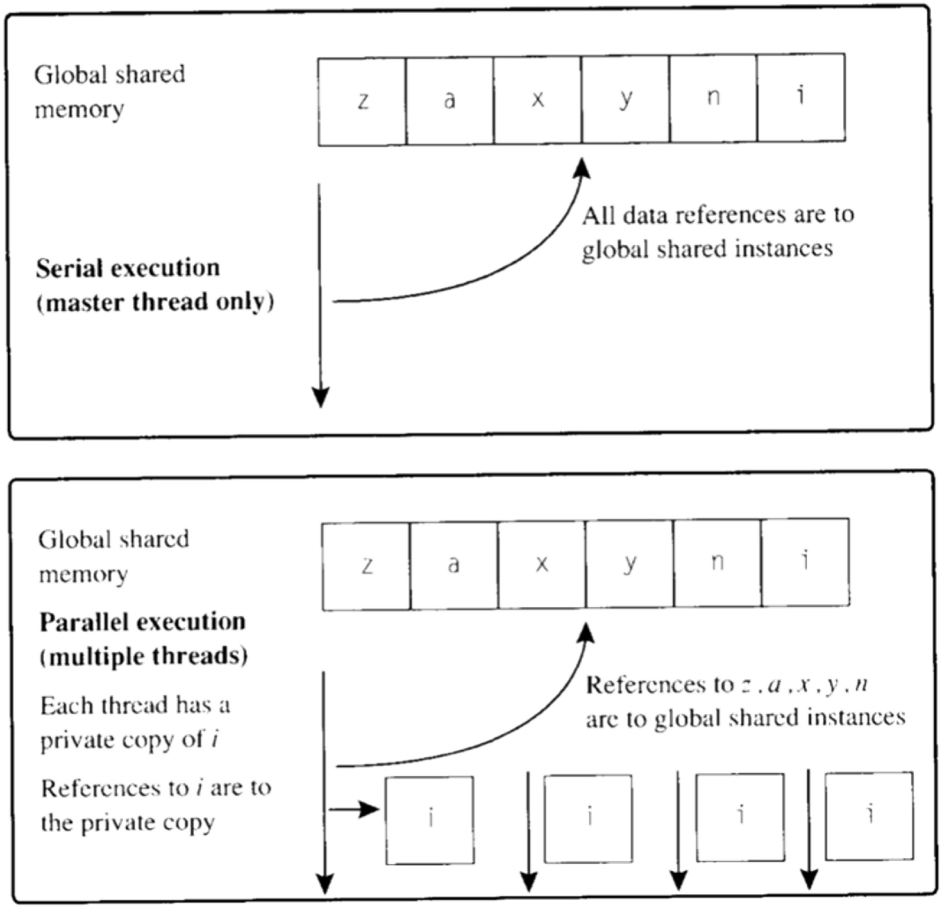
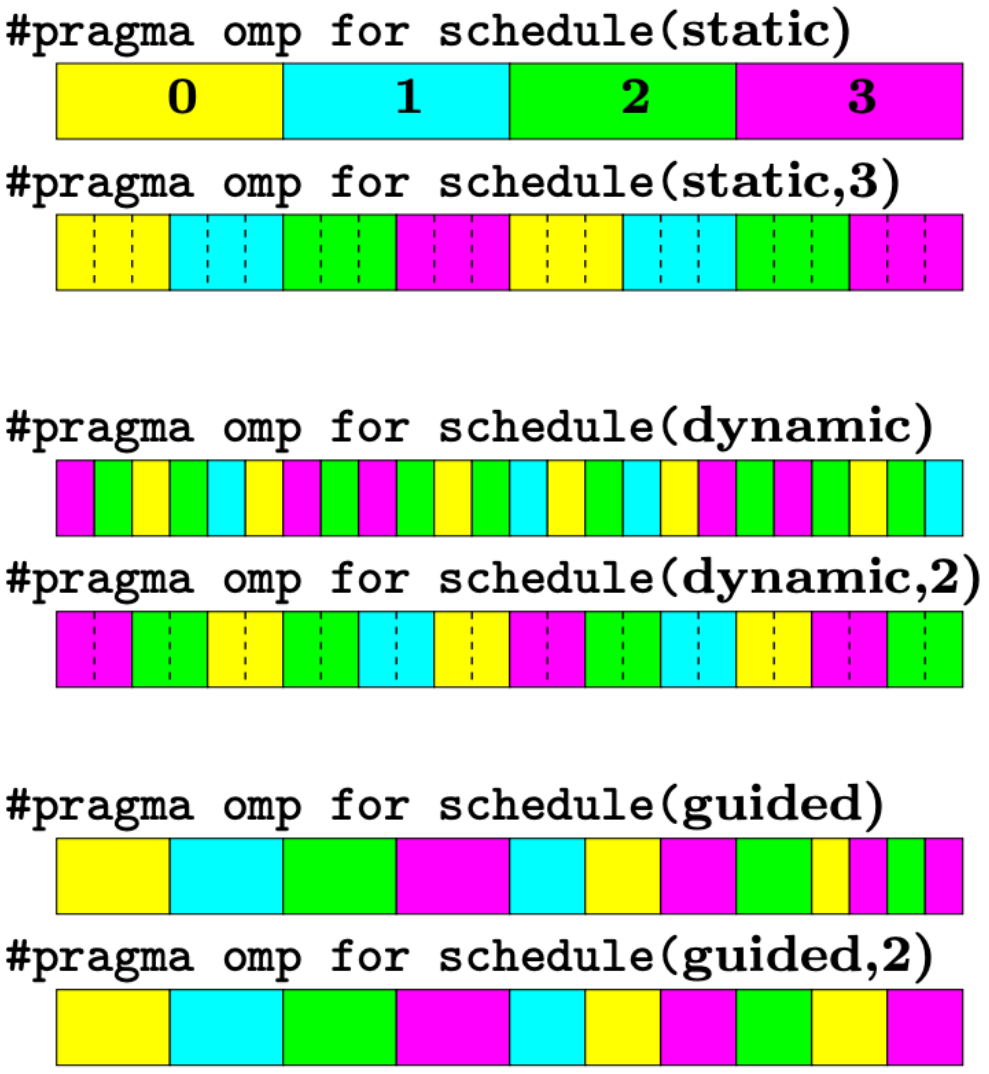




Figure 2.3 The behavior of private variables in an OpenMP program.

# OpenMP - Scheduling



# OpenMP – o conceito de redução

```
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 - Demonstrações

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