

Escola Supercomputador



INTRODUÇÃO À PROGRAMAÇÃO PARALELA E VETORIAL

Matheus S. Serpa

msserpa@inf.ufrgs.br

MC-SD02-II

GRUPO DE PROCESSAMENTO PARALELO E DISTRIBUÍDO

Philippe O. A. Navaux (**Coordenador**)

Big Data

Computer Architecture

Fog and Edge Computing

Cloud Computing

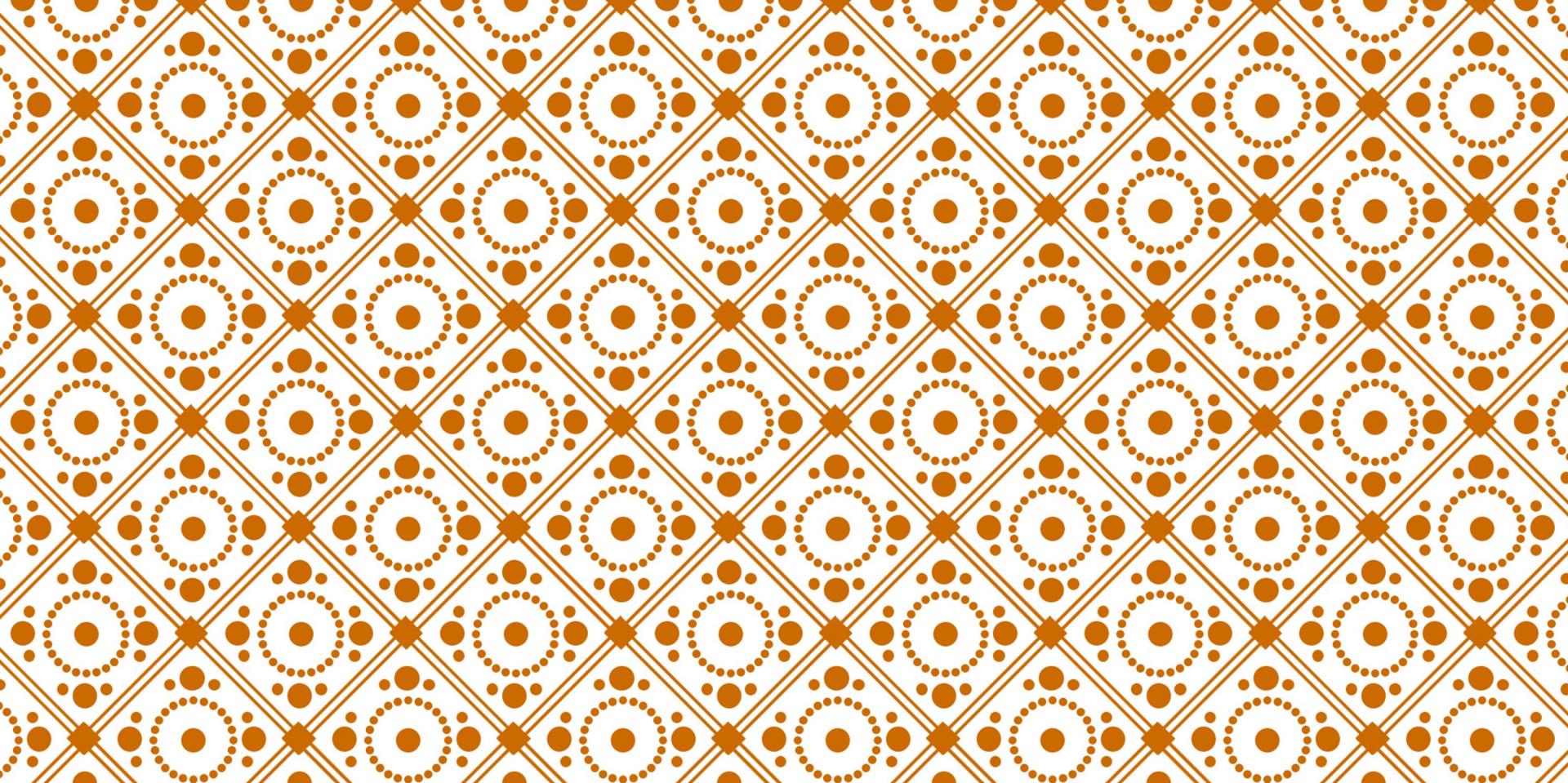
High Performance Computing

Oil and Gas



ACESSO AO MATERIAL

<https://www.inf.ufrgs.br/~msserpa/MC-SD02-II.zip>



APRESENTAÇÃO DA ÁREA

POR QUE ESTUDAR PROGRAMAÇÃO PARALELA?

Os programas já não são rápidos o suficiente?

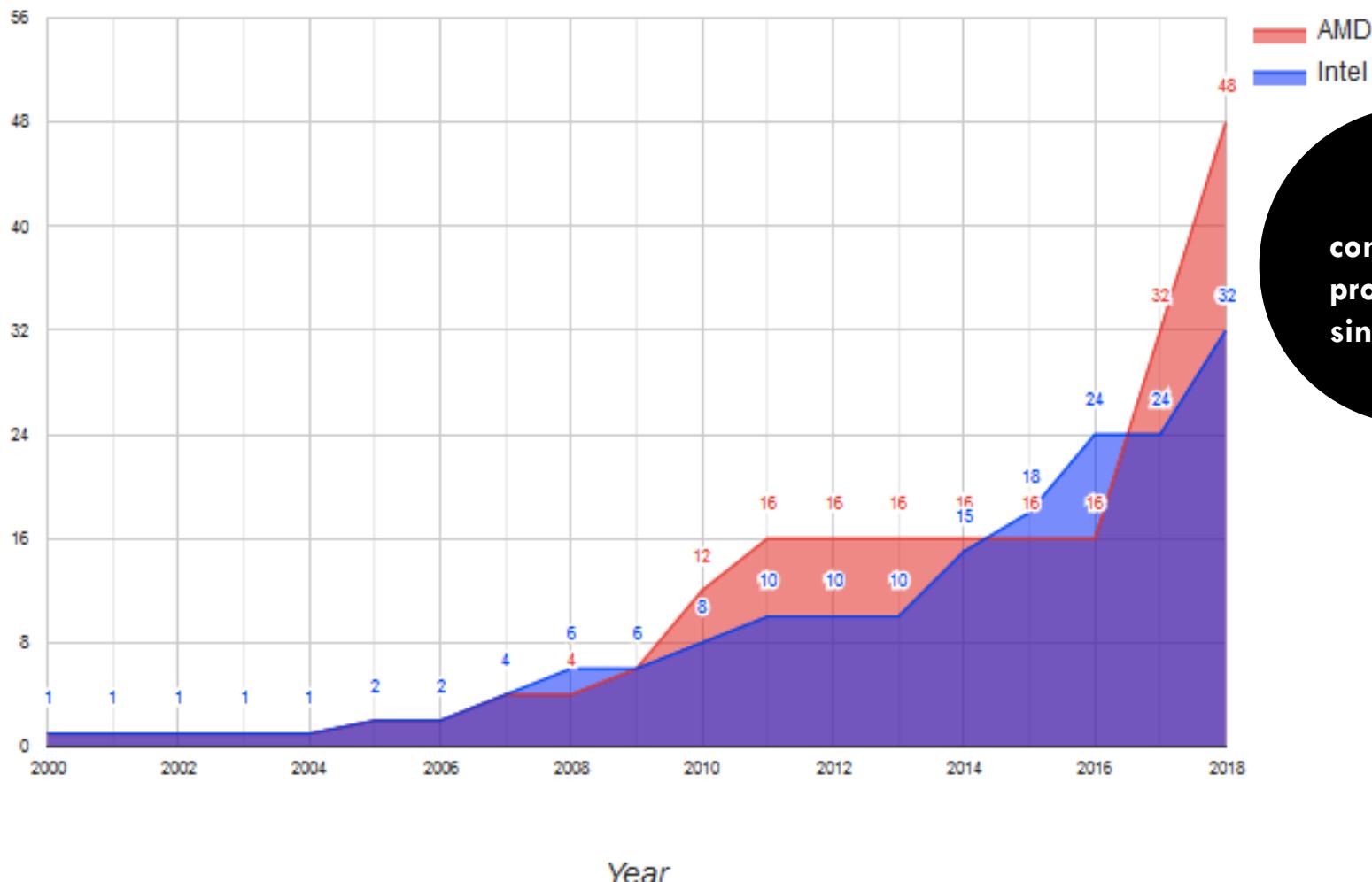
As máquinas já não são rápidas o suficiente?

REQUISITOS SEMPRE MUDANDO



EVOLUÇÃO DA INTEL E AMD

Highest amount of cores per CPU (AMD vs Intel year by year)



Onde
comprar um
processador
single-core?

POR QUE PROGRAMAÇÃO PARALELA?

Dois dos principais motivos para utilizar programação paralela são:

- **Reducir o tempo** necessário para solucionar um problema.
- **Resolver problemas mais complexos** e de maior dimensão.

POR QUE PROGRAMAÇÃO PARALELA?

Dois dos principais motivos para utilizar programação paralela são:

- **Reducir o tempo** necessário para solucionar um problema.
- **Resolver problemas mais complexos** e de maior dimensão.

Outros motivos são:

- Utilizar recursos computacionais subaproveitados.
- Ultrapassar limitações de memória quando a memória disponível num único computador é insuficiente para a resolução do problema.
- Ultrapassar os limites físicos que atualmente começam a restringir a possibilidade de construção de computadores sequenciais cada vez mais rápidos.

OPÇÕES PARA CIENTISTAS DA COMPUTAÇÃO

1. Crie uma **nova linguagem** para programas paralelos
2. Crie um **hardware** para extrair paralelismo
3. Deixe o **compilador** fazer o trabalho sujo
 - Paralelização automática
 - Ou **crie anotações no código sequencial**
4. Use os recursos do **sistema operacional**
 - Com memória compartilhada – threads
 - Com memória distribuída – SPMD
5. Use a **estrutura dos dados** para definir o paralelismo
6. Crie uma **abstração de alto nível** – Objetos, funções aplicáveis, etc.

PRINCIPAIS MODELOS DE PROGRAMAÇÃO PARALELA

Programação em Memória Compartilhada (OpenMP, Cilk, CUDA)

- Programação usando processos ou threads.
- Decomposição do domínio ou funcional com granularidade fina, média ou grossa.
- Comunicação através de **memória compartilhada**.
- Sincronização através de mecanismos de exclusão mútua.

Programação em Memória Distribuída (MPI)

- Programação usando processos distribuídos
- Decomposição do domínio com granularidade grossa.
- Comunicação e sincronização por **troca de mensagens**.

FATORES DE LIMITAÇÃO DO DESEMPENHO

Código Sequencial: existem partes do código que são inherentemente sequenciais (e.g. iniciar/terminar a computação).

Concorrência/Paralelismo: o número de tarefas pode ser escasso e/ou de difícil definição.

Comunicação: existe sempre um custo associado à troca de informação e enquanto as tarefas processam essa informação não contribuem para a computação.

Sincronização: a partilha de dados entre as várias tarefas pode levar a problemas de contenção no acesso à memória e enquanto as tarefas ficam à espera de sincronizar não contribuem para a computação.

Granularidade: o número e o tamanho das tarefas é importante porque o tempo que demoram a ser executadas tem de compensar os custos da execução em paralelo (e.g. custos de criação, comunicação e sincronização).

Balanceamento de Carga: ter os processadores maioritariamente ocupados durante toda a execução é decisivo para o desempenho global do sistema.

COMO IREMOS PARALELIZAR? PENSANDO!

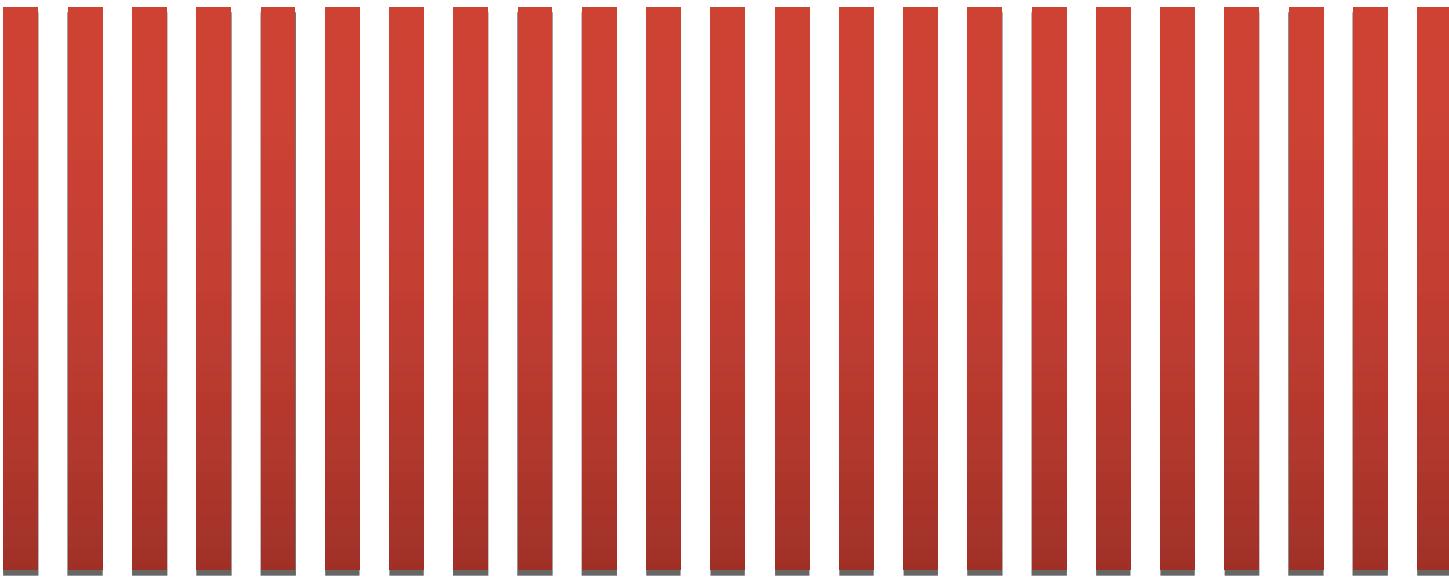


Trabalho



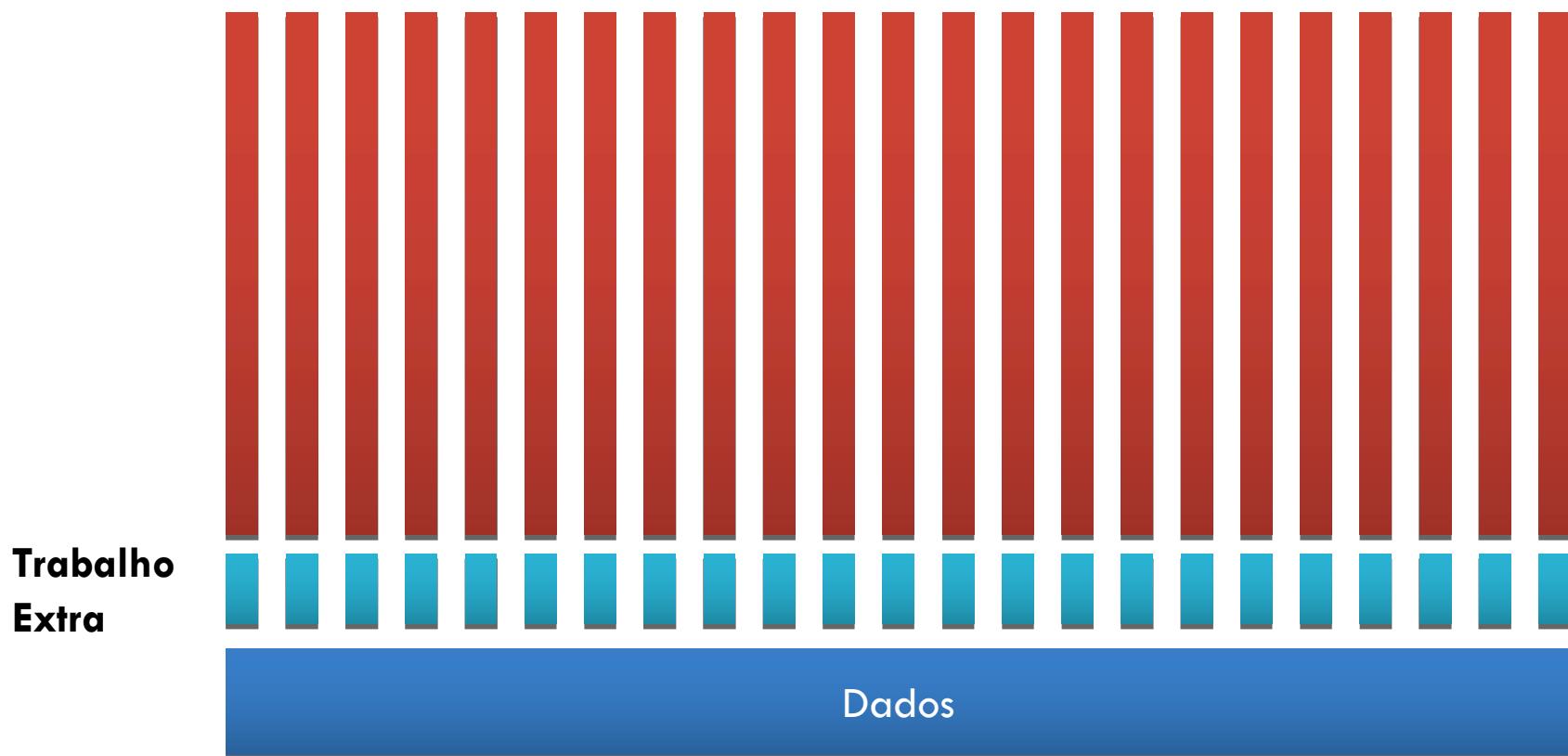
Dados

COMO IREMOS PARALELIZAR? PENSANDO!

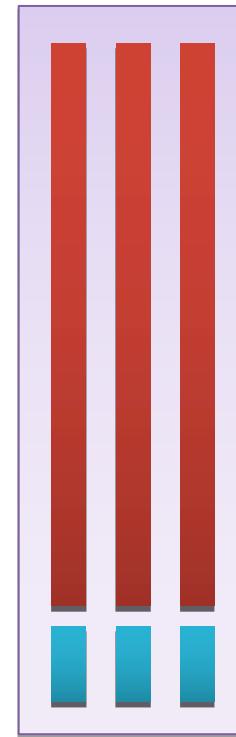
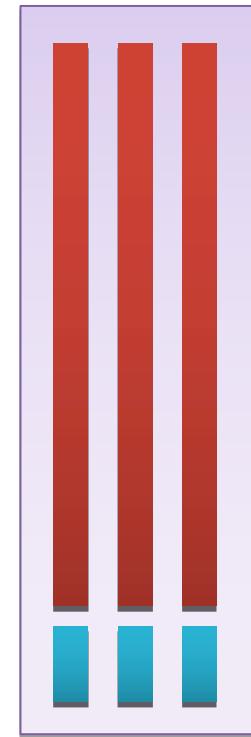
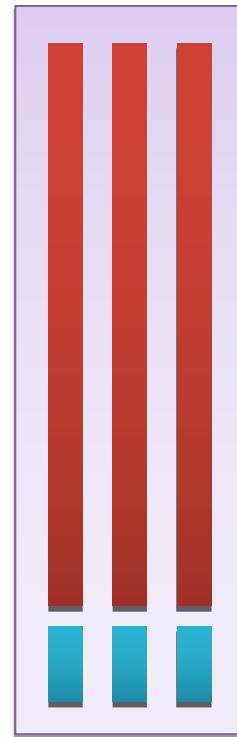
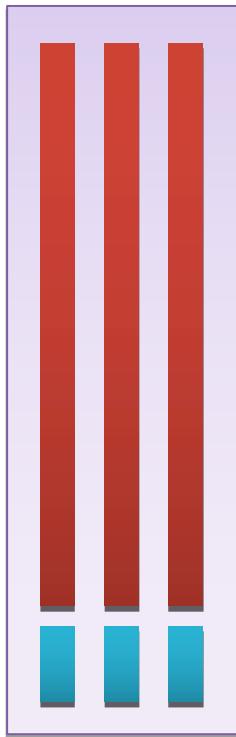
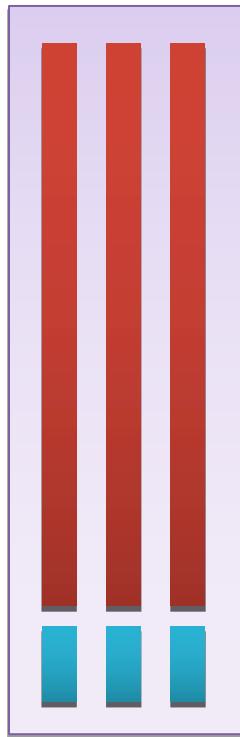


Dados

COMO IREMOS PARALELIZAR? PENSANDO!



COMO IREMOS PARALELIZAR? PENSANDO!



Divisão e Organização lógica do nosso algoritmo paralelo

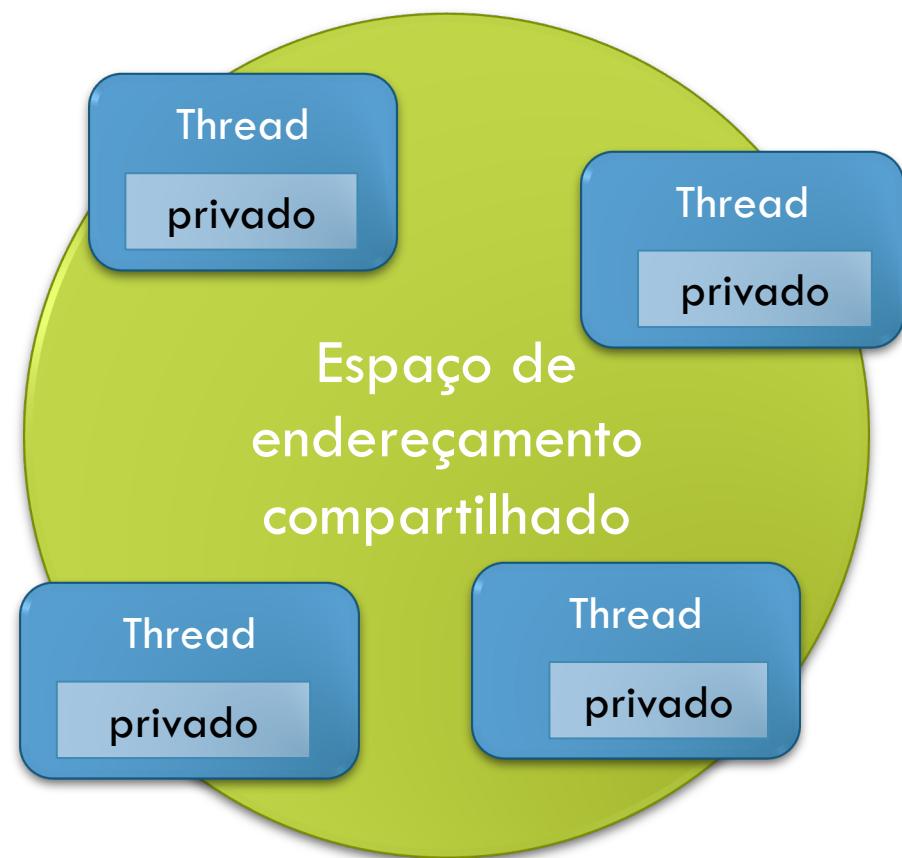
UM PROGRAMA DE MEMÓRIA COMPARTILHADA

Uma instância do programa:

Um processo e muitas threads.

Threads interagem através de leituras/escrita com o espaço de endereçamento compartilhado.

Sincronização garante a ordem correta dos resultados.



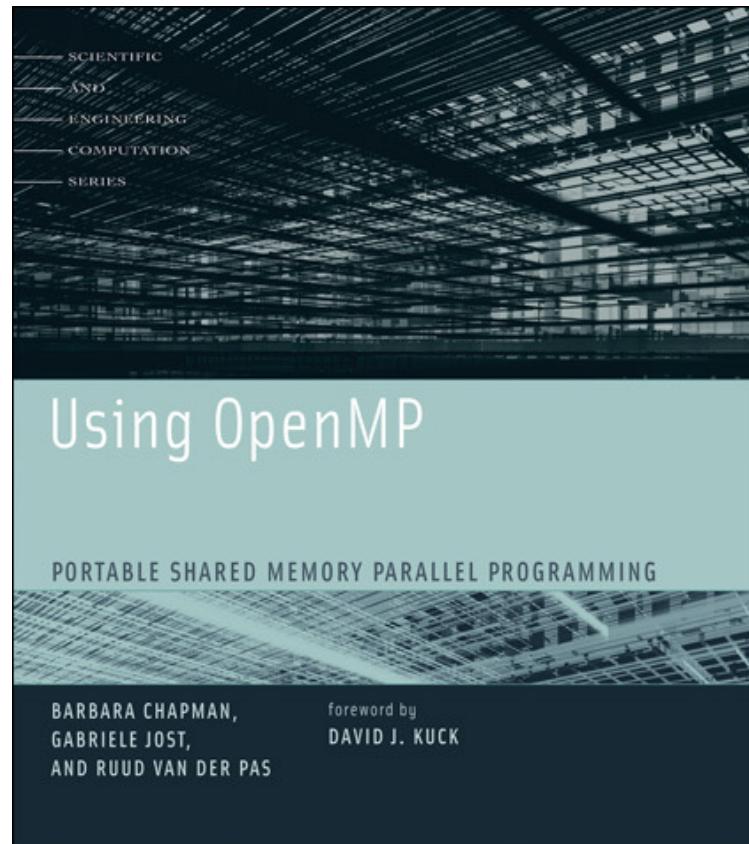
BIBLIOGRAFIA BÁSICA

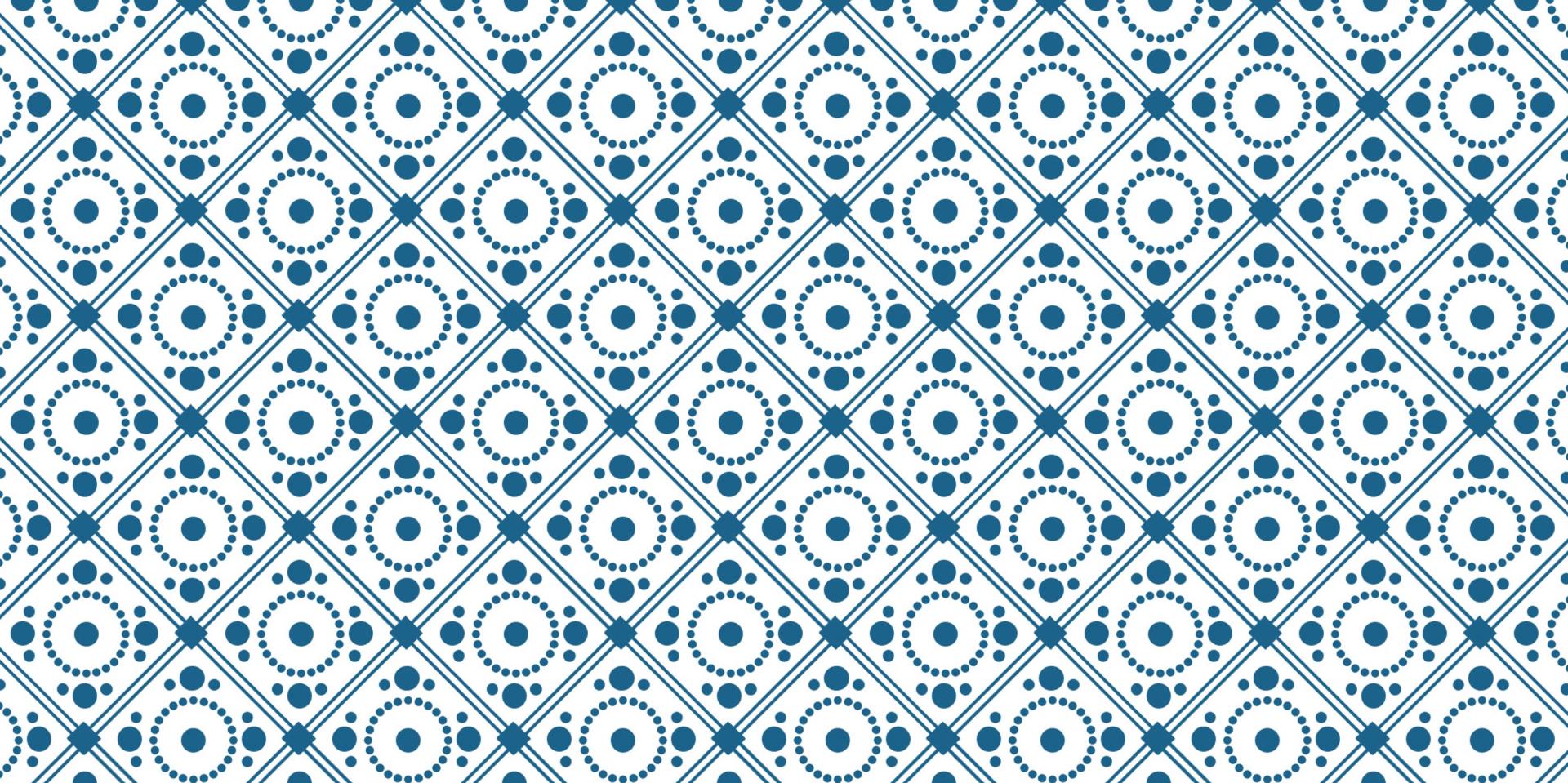
Using OpenMP - Portable Shared Memory Parallel Programming

Autores: Barbara Chapman,
Gabriele Jost and Ruud van der
Pas

Editora: MIT Press

Ano: 2007





FUNDAMENTOS E TERMINOLOGIAS

PORQUÊ PROGRAMAÇÃO PARALELA?

Se um único computador (processador) consegue resolver um problema em N segundos, podem N computadores (processadores) resolver o mesmo problema em 1 segundo?

PORQUÊ PROGRAMAÇÃO PARALELA?

Dois dos principais motivos para utilizar programação paralela são:

- **Reducir o tempo** necessário para solucionar um problema.
- **Resolver problemas mais complexos** e de maior dimensão.

Outros motivos são: ???

PORQUÊ PROGRAMAÇÃO PARALELA?

Dois dos principais motivos para utilizar programação paralela são:

- **Reducir o tempo** necessário para solucionar um problema.
- **Resolver problemas mais complexos** e de maior dimensão.

Outros motivos são:

- Tirar partido de recursos computacionais não disponíveis localmente ou subaproveitados.
- Ultrapassar limitações de memória quando a memória disponível num único computador é insuficiente para a resolução do problema.
- Ultrapassar os limites físicos de velocidade e de miniaturização que atualmente começam a restringir a possibilidade de construção de computadores sequenciais cada vez mais rápidos.

PORQUÊ PROGRAMAÇÃO PARALELA?

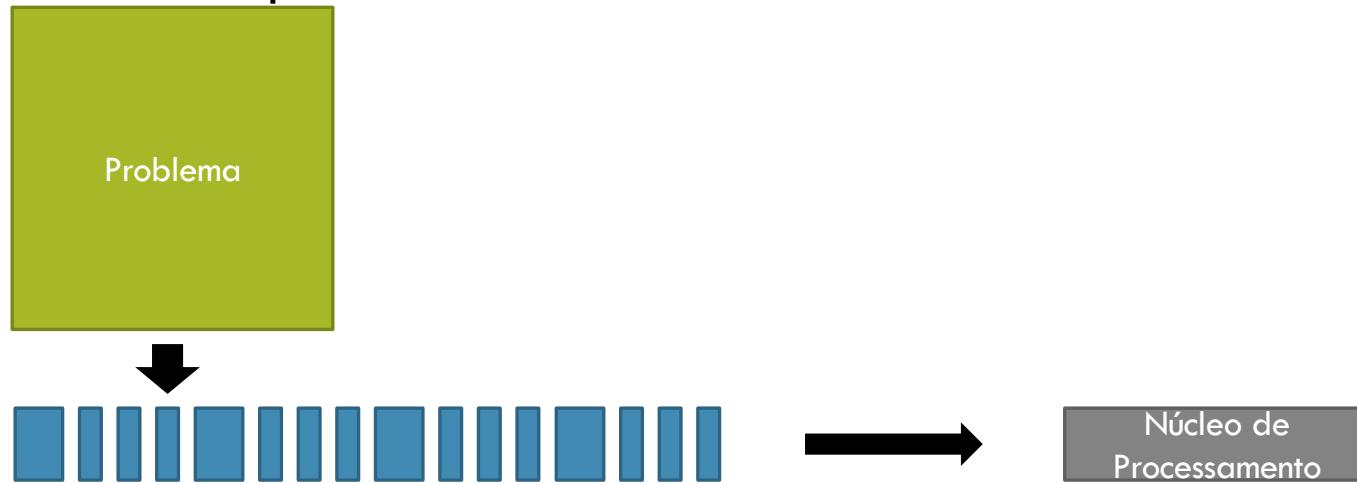
Atualmente, as aplicações que exigem o desenvolvimento de computadores cada vez mais rápidos estão por todo o lado.

Estas aplicações ou requerem um **grande poder de computação** ou requerem o processamento de **grandes quantidades de informação**. Alguns exemplos são:

- Bases de dados paralelas
- Mineração de dados (data mining)
- Serviços de procura baseados na web
- Serviços associados a tecnologias multimídia e telecomunicações
- Computação gráfica e realidade virtual
- Diagnóstico médico assistido por computador
- Gestão de grandes indústrias/corporações

PROGRAMAÇÃO SEQUENCIAL

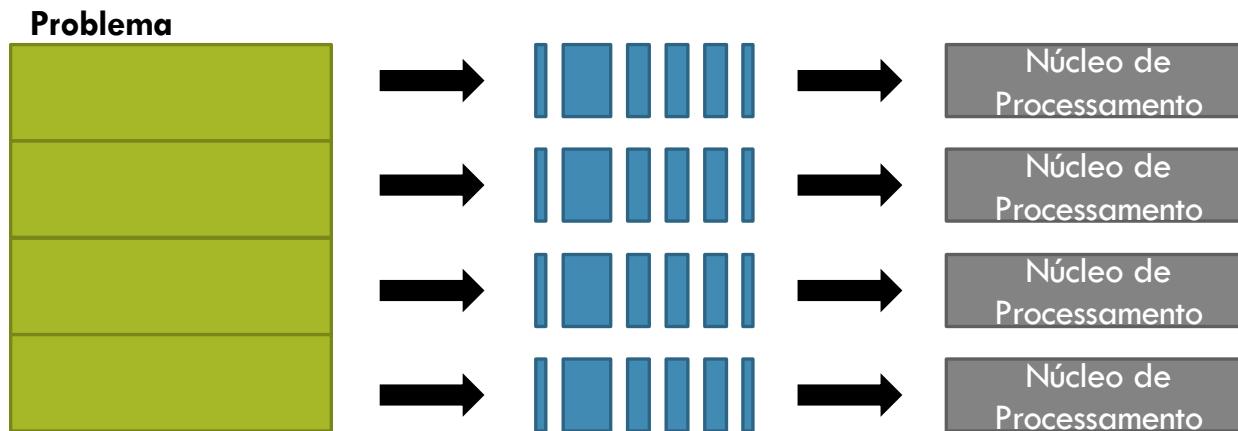
Considerado **programação sequencial** quando este é visto como uma série de instruções sequenciais que devem ser executadas num único processador.



PROGRAMAÇÃO PARALELA

Considerado **programação paralela** quando este é visto como um conjunto de partes que podem ser resolvidas concorrentemente.

Cada parte é igualmente constituída por uma série de instruções sequenciais, mas que no seu conjunto podem ser executadas simultaneamente.

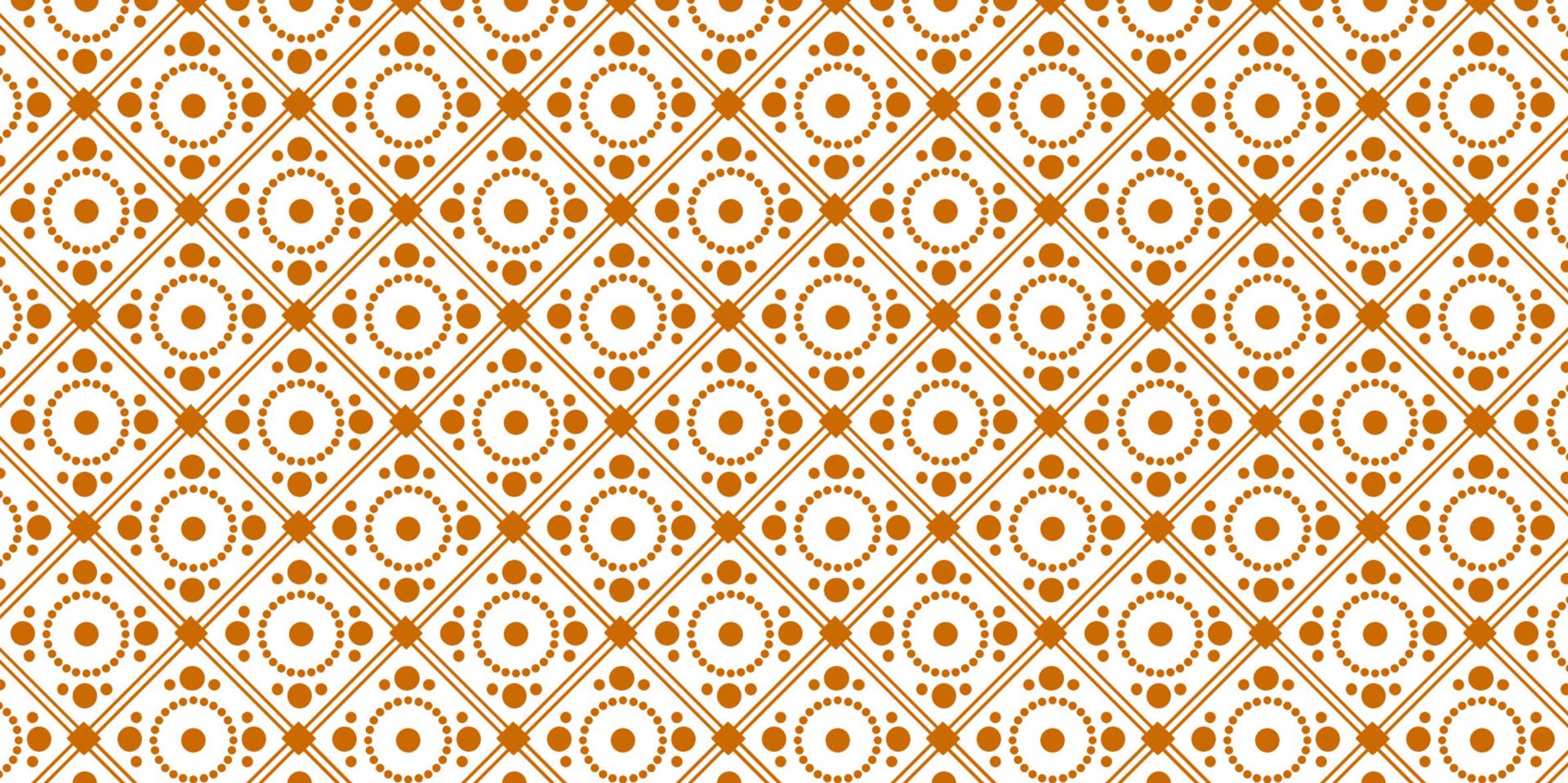


COMPUTAÇÃO PARALELA

De uma forma simples, a computação paralela pode ser definida como o uso simultâneo de vários recursos computacionais de forma a reduzir o tempo necessário para resolver um determinado problema.

Esses recursos computacionais podem incluir:

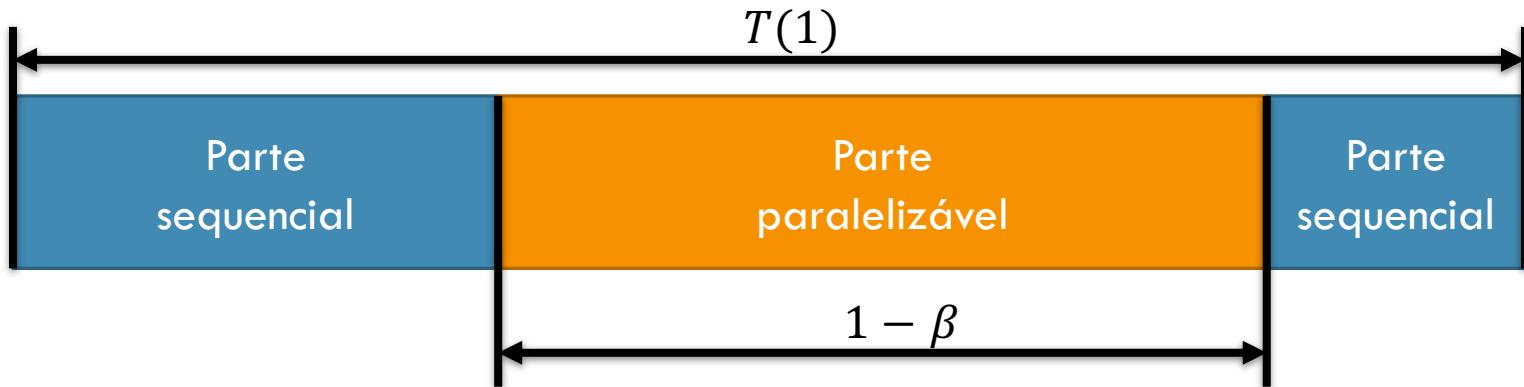
- Um único computador com múltiplos processadores.
- Um número arbitrário de computadores ligados por rede.
- A combinação de ambos.



LEI DE AMDAHL (1967)

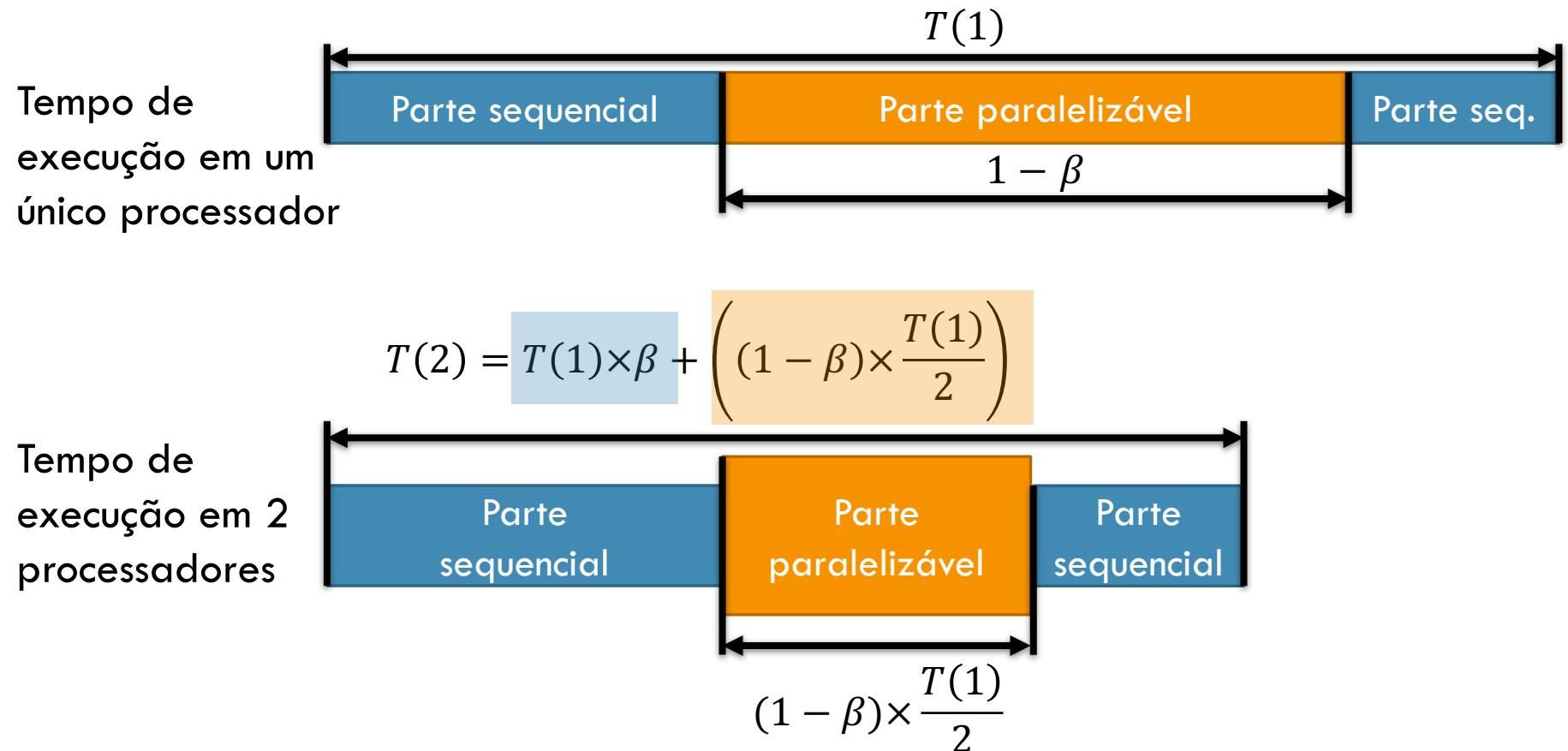
LEI DE AMDAHL

Tempo de execução em um único processador:



β = fração de código que é puramente sequencial

LEI DE AMDAHL



LEI DE AMDAHL

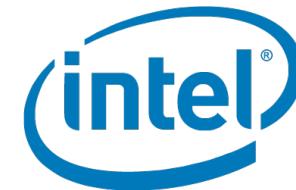
Seja $0 \leq \beta \leq 1$ a fração da computação que só pode ser realizada sequencialmente.

A lei de Amdahl diz-nos que o speedup máximo que uma aplicação paralela com p processadores pode obter é:

$$S(p) = \frac{1}{\beta + \frac{(1 - \beta)}{p}}$$

A lei de Amdahl também pode ser utilizada para determinar o limite máximo de speedup que uma determinada aplicação poderá alcançar independentemente do número de processadores a utilizar (limite máximo teórico).

INTRODUÇÃO AO OPENMP



INTRODUÇÃO

OpenMP é um dos modelos de programação paralelas mais usados hoje em dia.

Esse modelo é relativamente fácil de usar, o que o torna um bom modelo para iniciar o aprendizado sobre escrita de programas paralelos.

Observações:

- Assumo que todos sabem programar em linguagem C. OpenMP também suporta Fortran e C++, mas vamos nos restringir a C.

SINTAXE BÁSICA - OPENMP

Tipos e protótipos de funções no arquivo:

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

A maioria das construções OpenMP são diretivas de compilação.

```
#pragma omp construct [clause [clause]...]
```

- Exemplo:

```
#pragma omp parallel private(var1, var2) shared(var3, var4)
```

A maioria das construções se aplicam a um **bloco estruturado**.

Bloco estruturado: Um bloco com um ou mais declarações com um ponto de entrada no topo e um ponto de saída no final.

Podemos ter um **exit()** dentro de um bloco desses.

NOTAS DE COMPILAÇÃO

Linux e OS X com **gcc** or **intel icc**:

```
gcc -fopenmp foo.c #GCC
```

```
icc -qopenmp foo.c #Intel ICC
```

```
export OMP_NUM_THREADS=40
```

```
./a.out
```

Para shell bash

Por padrão é o nº de proc. virtuais.

Também
funciona no
Windows!

Até mesmo
no Visual
Studio!

**Mas vamos
usar Linux**
😊

FUNÇÕES

Funções da biblioteca OpenMP.

```
// 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();

// Comando para compilação habilitando o OpenMP.
icc -o hello hello.c -fopenmp
```

DIRETIVAS

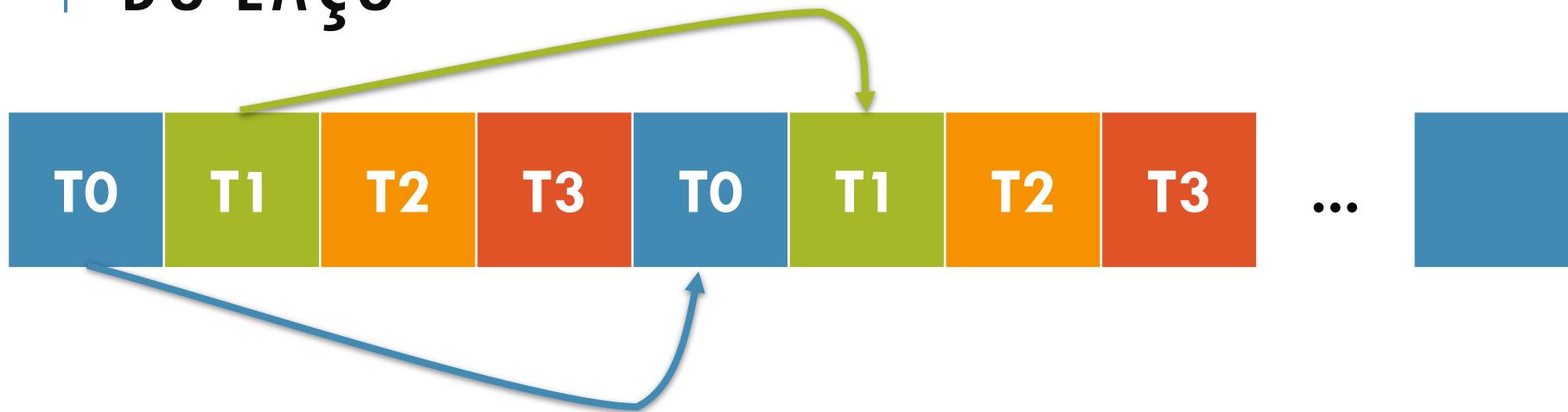
Diretivas do OpenMP.

```
// 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

}
```

DISTRIBUIÇÃO CÍCLICA DE ITERAÇÕES DO LAÇO



```
// Laço original
for(i = 0; i < N; i++)
```

```
// Distribuição cíclica
for(i = myid; i < N; i += nthreads)
```

FUNÇÕES

Funções da biblioteca OpenMP.

```
// 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();

// Comando para compilação habilitando o OpenMP.
icc -o hello hello.c -fopenmp
```

DIRETIVAS

Diretivas do OpenMP.

```
// 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

}
```

COMO AS THREADS INTERAGEM?

OpenMP é um modelo de *multithreading* de memória compartilhada.

- Threads se comunicam através de variáveis compartilhadas.

→ Compartilhamento não intencional de dados causa **condições de corrida**.

- Condições de corrida: quando a saída do programa muda quando as threads são escalonadas de forma diferente.

Apesar de este ser um aspecto mais poderoso da utilização de threads, também pode ser um dos mais problemáticos.

→ O problema existe quando dois ou mais threads tentam acessar/alterar as mesmas estruturas de dados (condições de corrida).

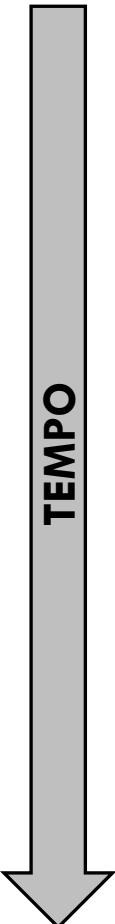
Para controlar condições de corrida:

- Usar sincronização para proteger os conflitos por dados

Sincronização é cara, por isso:

- Tentaremos mudar a forma de acesso aos dados para minimizar a necessidade de sincronizações.

CONDIÇÕES DE CORRIDA: EXEMPLO

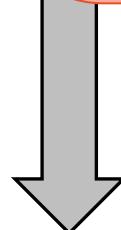
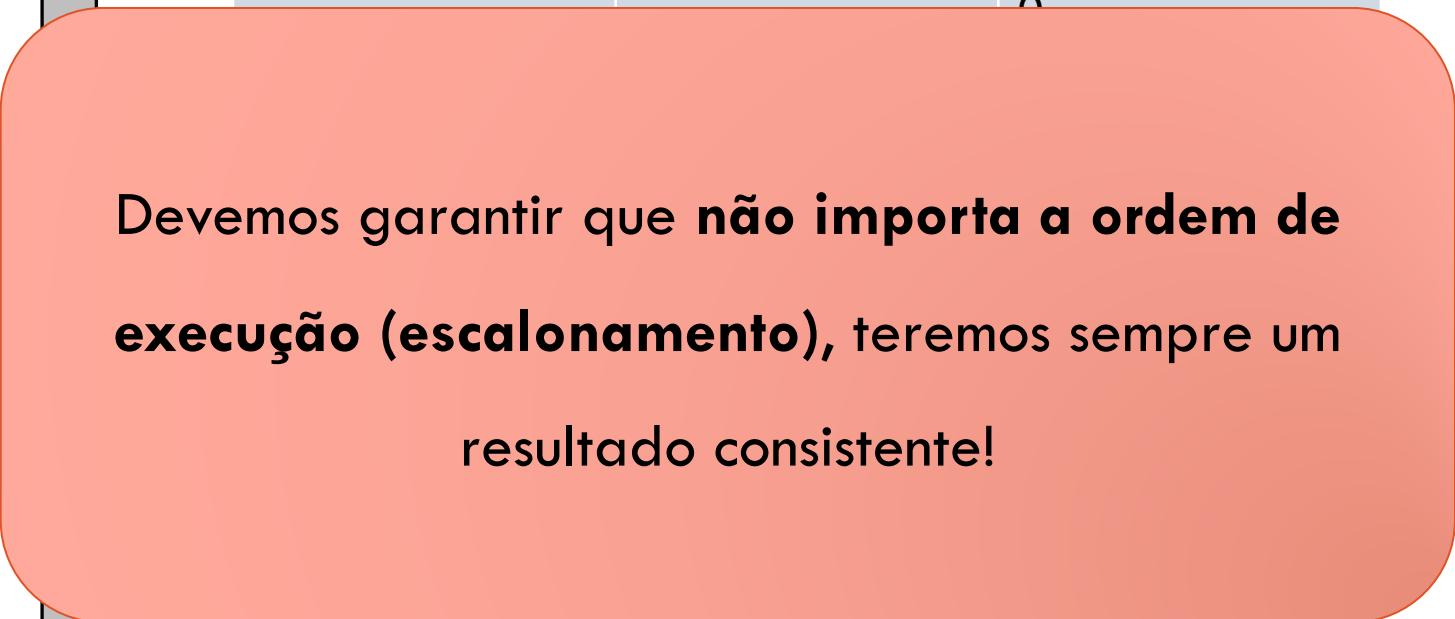


	Thread 0	Thread 1	sum
			0
Leia sum			0
0			
		Leia sum	0
		0	
		Some 0, 5	0
		5	
Some 0, 10			0
10			
		Escreva 5, sum	5
		5	
Escreva 10, sum			10
10			
			15 !?

CONDIÇÕES DE CORRIDA: EXEMPLO

	Thread 0	Thread 1	sum
			0

Devemos garantir que **não importa a ordem de execução (escalonamento)**, teremos sempre um resultado consistente!



	Escreva 5, sum 5	0
Escreva 10, sum 10		10

15 !?

SÍNCRONIZAÇÃO

Assegura que uma ou mais *threads* estão em um estado bem definido em um ponto conhecido da execução.

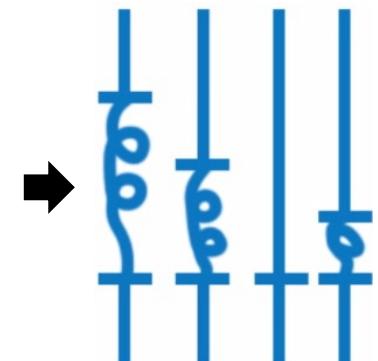
As duas formas mais comuns de sincronização são:

SÍNCRONIZAÇÃO

Assegura que uma ou mais *threads* estão em um estado bem definido em um ponto conhecido da execução.

As duas formas mais comuns de sincronização são:

Barreira: Cada *thread* espera na barreira até a chegada de todas as demais



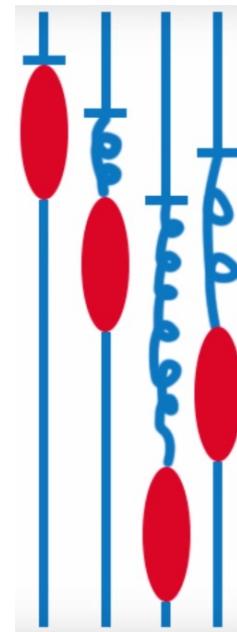
SÍNCRONIZAÇÃO

Assegura que uma ou mais *threads* estão em um estado bem definido em um ponto conhecido da execução.

As duas formas mais comuns de sincronização são:

Barreira: Cada *thread* espera na barreira até a chegada de todas as demais

Exclusão mútua: Define um bloco de código onde apenas uma *thread* pode executar por vez.



SÍNCRONIZAÇÃO: BARRIER

Barrier: Cada *thread* espera até que as demais cheguem.

```
#pragma omp parallel
{
    int id = omp_get_thread_num(); // variável privada
    A[id] = big_calc1(id);

    #pragma omp barrier

    B[id] = big_calc2(id, A);
} // Barreira implícita
```

SINCRONIZAÇÃO: CRITICAL

Exclusão mútua: Apenas uma *thread* pode entrar por vez

```
#pragma omp parallel
{
    float B; // variável privada
    int i, myid, nthreads; // variáveis privadas
    myid = omp_get_thread_num();
    nthreads = omp_get_num_threads();
    for(i = myid; i < niters; i += nthreads){
        B = big_job(i); // Se for pequeno, muito overhead
        #pragma omp critical
        res += consume (B);
    }
}
```

As *threads* esperam sua vez,
apenas uma chama `consume()` por vez.

SINCRONIZAÇÃO: ATOMIC

atomic prove exclusão mútua para operações específicas.

```
#pragma omp parallel
{
    double tmp, B;
    B = DOIT();
    tmp = big_ugly(B);
    #pragma omp atomic
    X += tmp;
}
```

Instruções especiais da
arquitetura (se
disponível)

Algumas operações aceitáveis:

v = x;
x = expr;
x++; ++x; x--; --x;
x op= expr;
v = x op expr;
v = x++; v = x--; v = ++x; v = --x;

PRINCÍPIO DA LOCALIDADE

Programas repetem trechos de código e acessam repetidamente dados próximos.

Localidade Temporal: posições de memória, uma vez acessadas, tendem a ser acessadas novamente em um espaço curto de tempo.

Localidade Espacial: se um item é referenciado, itens cujos endereços sejam próximos dele tendem a ser referenciados em um espaço curto de tempo.

ACESSOS INTERCALADOS



Cache 0 – Thread 0				Cache 1 – Thread 1				Cache 2 – Thread 2				Cache 3 – Thread 3			
v[0]	v[1]	v[2]	v[3]												
Cache 0 – Thread 0				Cache 1 – Thread 1				Cache 2 – Thread 2				Cache 3 – Thread 3			
v[4]	v[5]	v[6]	v[7]												
Cache 0 – Thread 0				Cache 1 – Thread 1				Cache 2 – Thread 2				Cache 3 – Thread 3			
v[8]	v[9]	v[10]	v[11]												
Cache 0 – Thread 0				Cache 1 – Thread 1				Cache 2 – Thread 2				Cache 3 – Thread 3			
v[12]	v[13]	v[14]	v[15]												

```
for(i = myid; i < N; i += nthreads)
```

25% do conteúdo trazido para a cache é utilizado.

ACESSOS CONSECUTIVOS

TEMPO

Cache 0 – Thread 0				Cache 1 – Thread 1				Cache 2 – Thread 2				Cache 3 – Thread 3			
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]	v[10]	v[11]	v[12]	v[13]	v[14]	v[15]
Cache 0 – Thread 0				Cache 1 – Thread 1				Cache 2 – Thread 2				Cache 3 – Thread 3			
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]	v[10]	v[11]	v[12]	v[13]	v[14]	v[15]
Cache 0 – Thread 0				Cache 1 – Thread 1				Cache 2 – Thread 2				Cache 3 – Thread 3			
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]	v[10]	v[11]	v[12]	v[13]	v[14]	v[15]
Cache 0 – Thread 0				Cache 1 – Thread 1				Cache 2 – Thread 2				Cache 3 – Thread 3			
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]	v[10]	v[11]	v[12]	v[13]	v[14]	v[15]

```
for(i = ini; i < end; i++)
```

100% do conteúdo trazido para a cache é utilizado.

CONSTRUÇÕES DE DIVISÃO DE LAÇOS

A construção de divisão de trabalho em laços divide as iterações do laço entre as *threads* do time.

```
#pragma omp parallel private(i) shared(N)
{
    #pragma omp for
    for(i = 0; i < N; i++)
        NEAT_STUFF(i);
}
```

A variável *i* será feita privada para cada *thread* por padrão. Você poderia fazer isso explicitamente com a cláusula **private(i)**

CONSTRUÇÕES DE DIVISÃO DE LAÇOS

UM EXEMPLO MOTIVADOR

Código sequencial

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

CONSTRUÇÕES DE DIVISÃO DE LAÇOS

UM EXEMPLO MOTIVADOR

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];
}
```

CONSTRUÇÕES DE DIVISÃO DE LAÇOS

UM EXEMPLO MOTIVADOR

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];
```

CONSTRUÇÕES PARALELA E DIVISÃO DE LAÇOS COMBINADAS

Algumas cláusulas podem ser combinadas.

```
double res[MAX]; int i;  
#pragma omp parallel  
{  
    #pragma omp for  
    for(i=0; i < MAX; i++)  
        res[i] = huge();  
}
```



```
double res[MAX]; int i;  
#pragma omp parallel for  
for(i=0; i < MAX; i++)  
    res[i] = huge();
```

REDUÇÃO

Combinação de variáveis locais de uma *thread* em uma variável única.

- Essa situação é bem comum, e chama-se **redução**.
- O suporte a tal operação é fornecido pela maioria dos ambientes de programação paralela.

DIRETIVA REDUCTION

`reduction(op : list_vars)`

Dentro de uma região paralela ou de divisão de trabalho:

- Será feita uma cópia local de cada variável na lista
- Será inicializada dependendo da **op** (ex. 0 para +, 1 para *).
- Atualizações acontecem na cópia local.
- Cópias locais são “reduzidas” para uma única variável original (global).

#pragma omp for reduction(* : var_mult)

CONSTRUÇÕES DE DIVISÃO DE LAÇOS : A DECLARAÇÃO **SCHEDULE**

A declaração **schedule** afeta como as iterações do laço serão mapeadas entre as threads



Como o laço
será mapeado
para as
threads?

CONSTRUÇÕES DE DIVISÃO DE LAÇOS : A DECLARAÇÃO SCHEDULE

schedule(static [,chunk])

- Distribui iterações de tamanho “chunk” para cada thread
- Se “chunk” é omitido, as iterações tem tamanho aproximadamente igual para cada thread

schedule(dynamic[,chunk])

- Inicialmente cada thread recebe um “chunk” de iterações, quando termina este “chunk” a thread acessa uma fila compartilhada para pegar o próximo “chunk” até que todas as iterações sejam executadas.
- Se “chunk” é omitido, cada pedaço tem tamanho 1

schedule(guided[,chunk])

- As threads pegam blocos de iterações dinamicamente, iniciando de blocos grandes reduzindo até o tamanho “chunk”.
 - Variação de dynamic para reduzir o overhead de escalonamento. Inicia com blocos grandes o que diminui o número de decisões de escalonamento.

CONSTRUÇÕES DE DIVISÃO DE LAÇOS : A DECLARAÇÃO **SCHEDULE**

schedule(runtime)

- O modelo de distribuição e o tamanho serão pegos da variável de ambiente OMP_SCHEDULE.
 - programador decide na hora da execução.
 - Exemplo: OMP_SCHEDULE="guided,2"

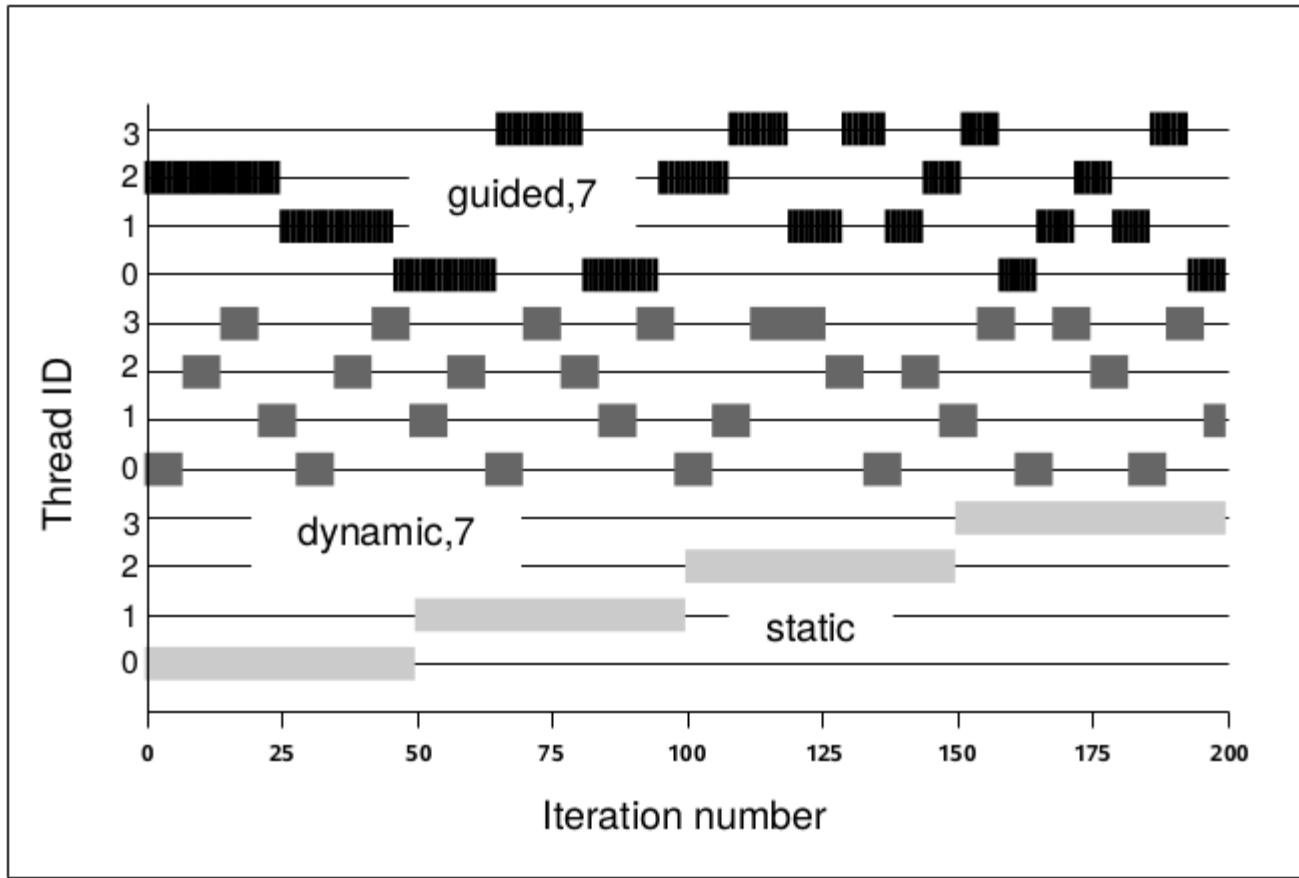
schedule(auto)

- Deixa a divisão por conta da biblioteca em tempo de execução (pode fazer algo diferente dos acima citados).

CONSTRUÇÕES DE DIVISÃO DE LAÇOS : A DECLARAÇÃO SCHEDULE

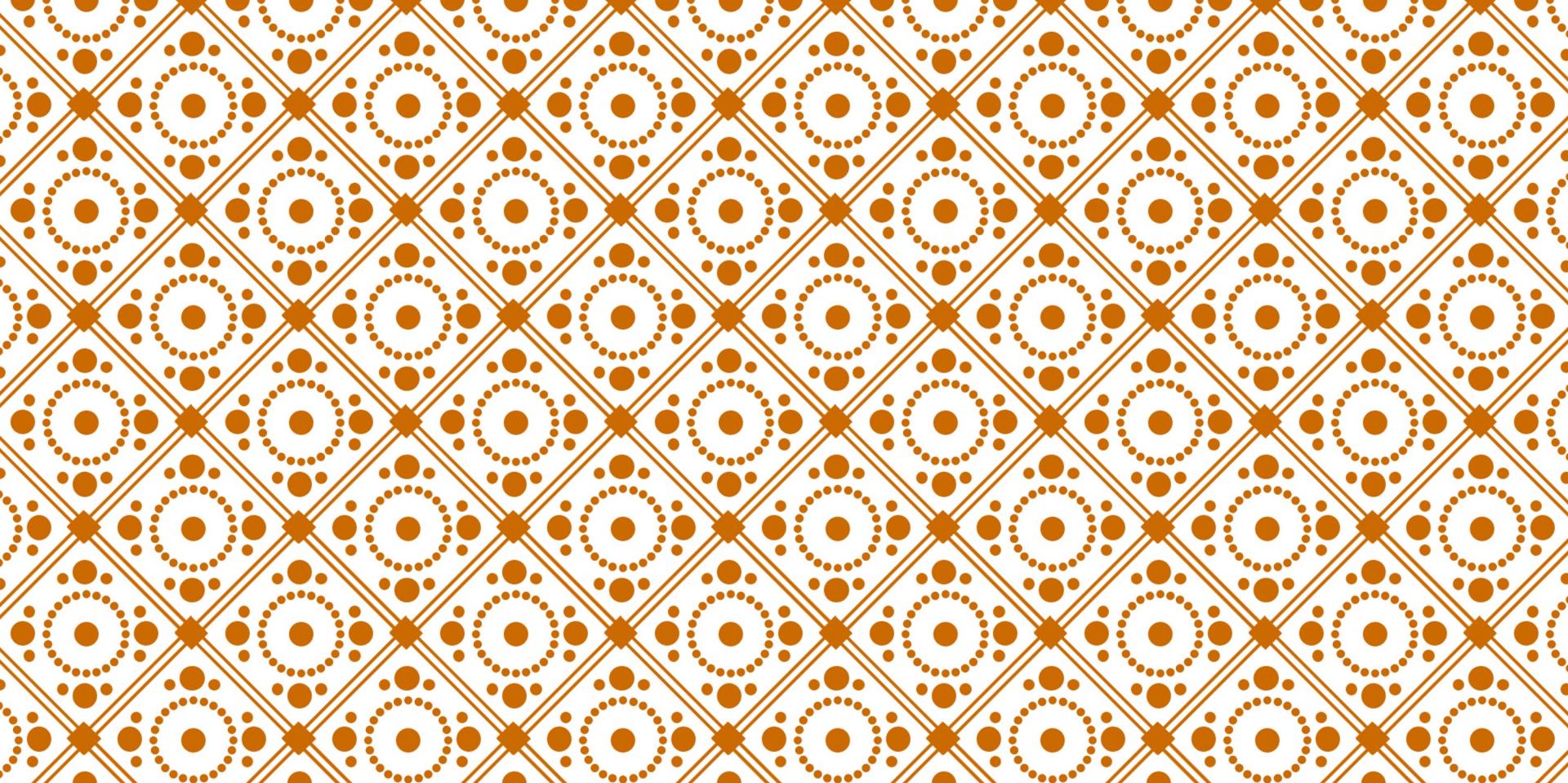
Tipo de Schedule	Quando usar	
STATIC	Pré determinado e previsível pelo programador	Menos trabalho durante a execução (mais durante a compilação)
DYNAMIC	Imprevisível, quantidade de trabalho por iteração altamente variável	Mais trabalho durante a execução (lógica complexa de controle)
GUIDED	Caso especial do dinâmico para reduzir o overhead dinâmico	
AUTO	Quando o tempo de execução pode “aprender” com as iterações anteriores do mesmo laço	

CONSTRUÇÕES DE DIVISÃO DE LAÇOS : A DECLARAÇÃO SCHEDULE



Exemplo de escalonamento para 200 iterações.

Fonte: Chapman, B. *Using OpenMP : portable shared memory parallel programming*



INTRODUÇÃO A PROGRAMAÇÃO VETORIAL



SINGLE INSTRUCTION MULTIPLE DATA (SIMD)

Técnica aplicada por unidade de execução

- Opera em mais de um elemento por iteração.
- Reduz número de instruções significativamente.

Elementos são armazenados em registradores SIMD

Scalar

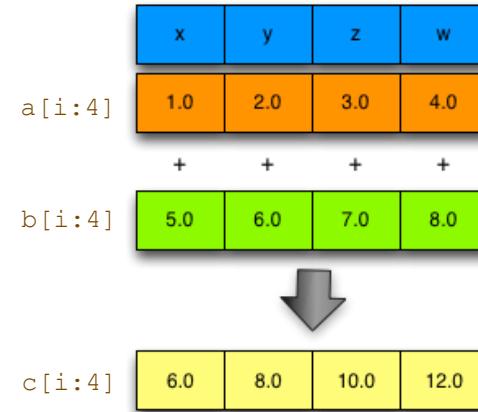
Uma instrução. Uma operação.

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

Vector

Uma instrução. Quatro operações, **por exemplo.**

```
for(i = 0; i < N; i += 4)  
    c[i:4] = a[i:4] + b[i:4];
```



SINGLE INSTRUCTION MULTIPLE DATA (SIMD)

Técnica aplicada por unidade de execução

- Opera em mais de um elemento por iteração.
- Reduz número de instruções significativamente.

Elementos são armazenados em registradores SIMD

Scalar

Uma instrução. Uma operação.

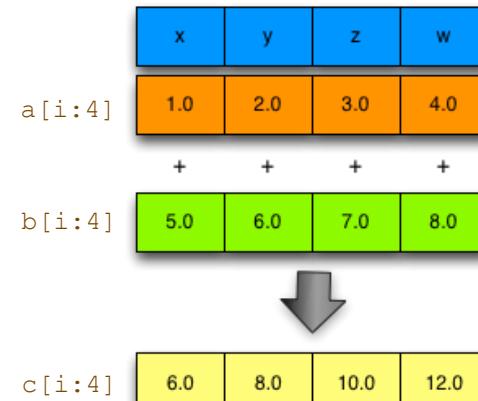
```
for(i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

Vector

Uma instrução. Quatro operações, **por exemplo.**

```
for(i = 0; i < N; i += 4)  
    c[i:4] = a[i:4] + b[i:4];
```

Dados contíguos para desempenho ótimo
 $c[0] \ c[1] \ c[2] \ c[3] \dots$



ALINHAMENTO DE MEMÓRIA

Alinhamento de dados

Funções do compilador **icc**.

```
void* _mm_malloc(size_t size, size_t align);  
void mm_free(void *ptr);
```

Indicar ao compilador que dados estão alinhados

Ajuda na auto vetorização.

```
#pragma vector aligned  
for(i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

PROGRAMAÇÃO VETORIAL

Vetorização

```
#pragma vector aligned
#pragma omp simd
for(i = 0; i < N; i++)
    c[i] = a[i] + b[i];
```

Vetorização com redução

```
#pragma vector aligned
#pragma omp simd reduction(+ : v)
for(i = 0; i < N; i++)
    v += a[i] + b[i];
```

INSTRUÇÕES AVX

Conjunto de Instruções utilizadas para melhorar o desempenho de operações de ponto flutuante.

- IA / Processamento de Imagens / Criptografia / Compressão de dados

- <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

```
#include <immintrin.h>
```

Tipos de dados: `_int64` `_m256d` `_m256i`

Instruções: `_mm256_set_pd` `_mm256_load_pd` `_mm256_mul_pd`
`_mm256_add_pd` `_mm256_storeu_pd`

INSTRUÇÕES AVX

```
_m256d _mm256_set_pd (double e3, double e2, double  
e1, double e0)
```

Description

Set packed double-precision (64-bit) floating-point elements in `dst` with the supplied values.

Operation

```
dst[63:0] := e0  
dst[127:64] := e1  
dst[191:128] := e2  
dst[255:192] := e3  
dst[MAX:256] := 0
```

INSTRUÇÕES AVX

```
_m256d _mm256_load_pd (double const * mem_addr)
```

Description

Load 256-bits (composed of 4 packed double-precision (64-bit) floating-point elements) from memory into `dst`. `mem_addr` must be aligned on a 32-byte boundary or a general-protection exception may be generated.

Operation

```
dst[255:0] := MEM[mem_addr+255:mem_addr]
dst[MAX:256] := 0
```

INSTRUÇÕES AVX

```
__m256d _mm256_mul_pd (__m256d a, __m256d b)
```

Description

Multiply packed double-precision (64-bit) floating-point elements in **a** and **b**, and store the results in **dst**.

Operation

```
FOR j := 0 to 3
    i := j*64
    dst[i+63:i] := a[i+63:i] * b[i+63:i]
ENDFOR
dst[MAX:256] := 0
```

INSTRUÇÕES AVX

```
__m256d _mm256_add_pd (__m256d a, __m256d b)
```

Description

Add packed double-precision (64-bit) floating-point elements in `a` and `b`, and store the results in `dst`.

Operation

```
FOR j := 0 to 3
    i := j*64
    dst[i+63:i] := a[i+63:i] + b[i+63:i]
ENDFOR
dst[MAX:256] := 0
```

INSTRUÇÕES AVX

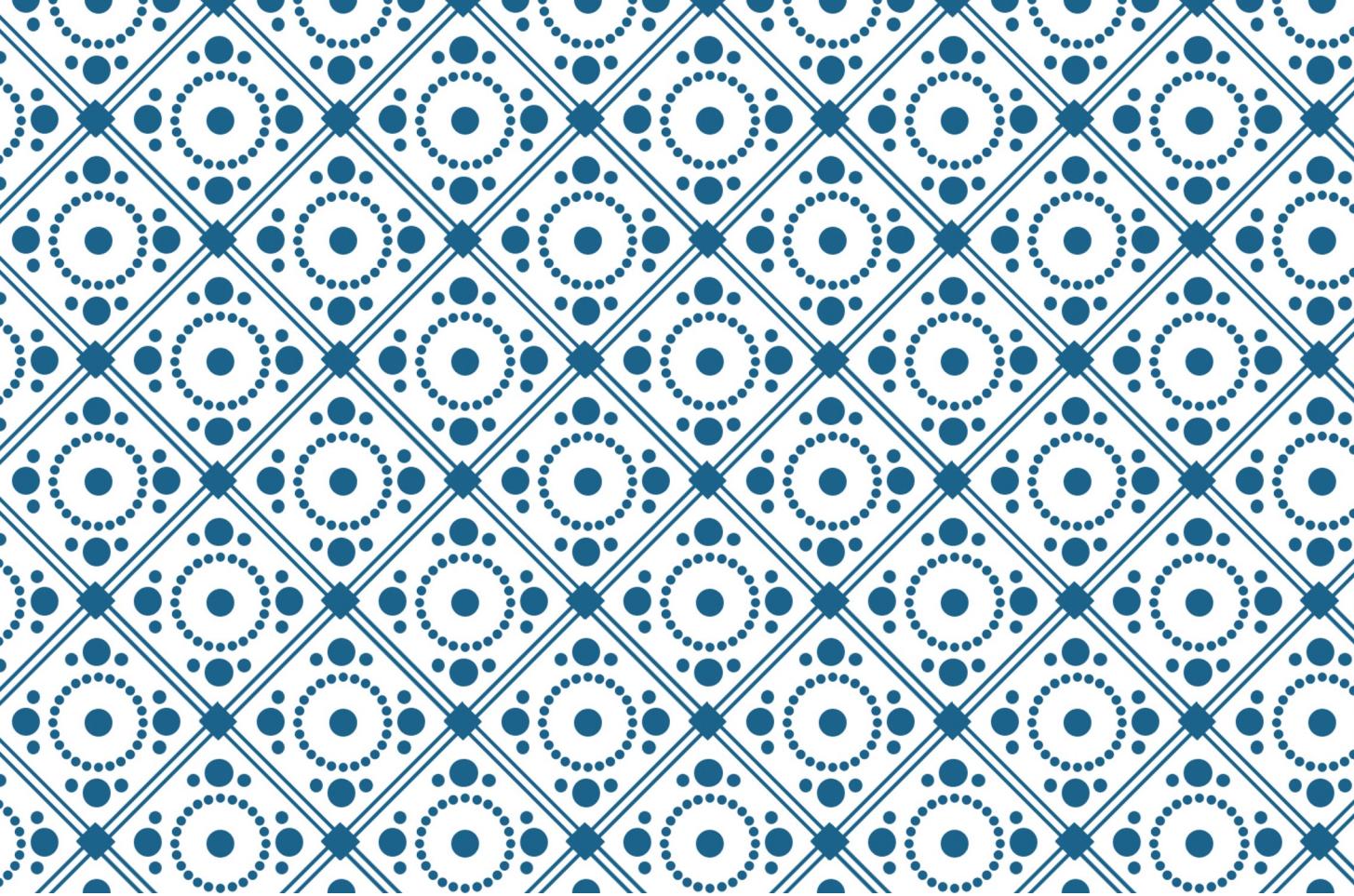
```
void _mm256_storeu_pd (double  
* mem_addr, __m256d a)
```

Description

Store 256-bits (composed of 4 packed double-precision (64-bit) floating-point elements) from `a` into memory. `mem_addr` does not need to be aligned on any particular boundary.

Operation

$$\text{MEM}[\text{mem_addr}+255:\text{mem_addr}] := \text{a}[255:0]$$



INTRODUÇÃO À PROGRAMAÇÃO PARALELA E VETORIAL

Matheus S. Serpa

msserpa@inf.ufrgs.br

Escola Supercomputador



MC-SD02-II