

SEMANA ACADÊMICA UNIFICADA DE

**ENGENHARIA DE SOFTWARE**

**& SISTEMAS DE INFORMAÇÃO**

# INTRODUÇÃO A PROGRAMAÇÃO PARALELA

---

*Por Prof. DSc Bárbara Quintela*  
[barbara@ice.ufjf.br](mailto:barbara@ice.ufjf.br)

*Parte 1 - 25/10/2018*

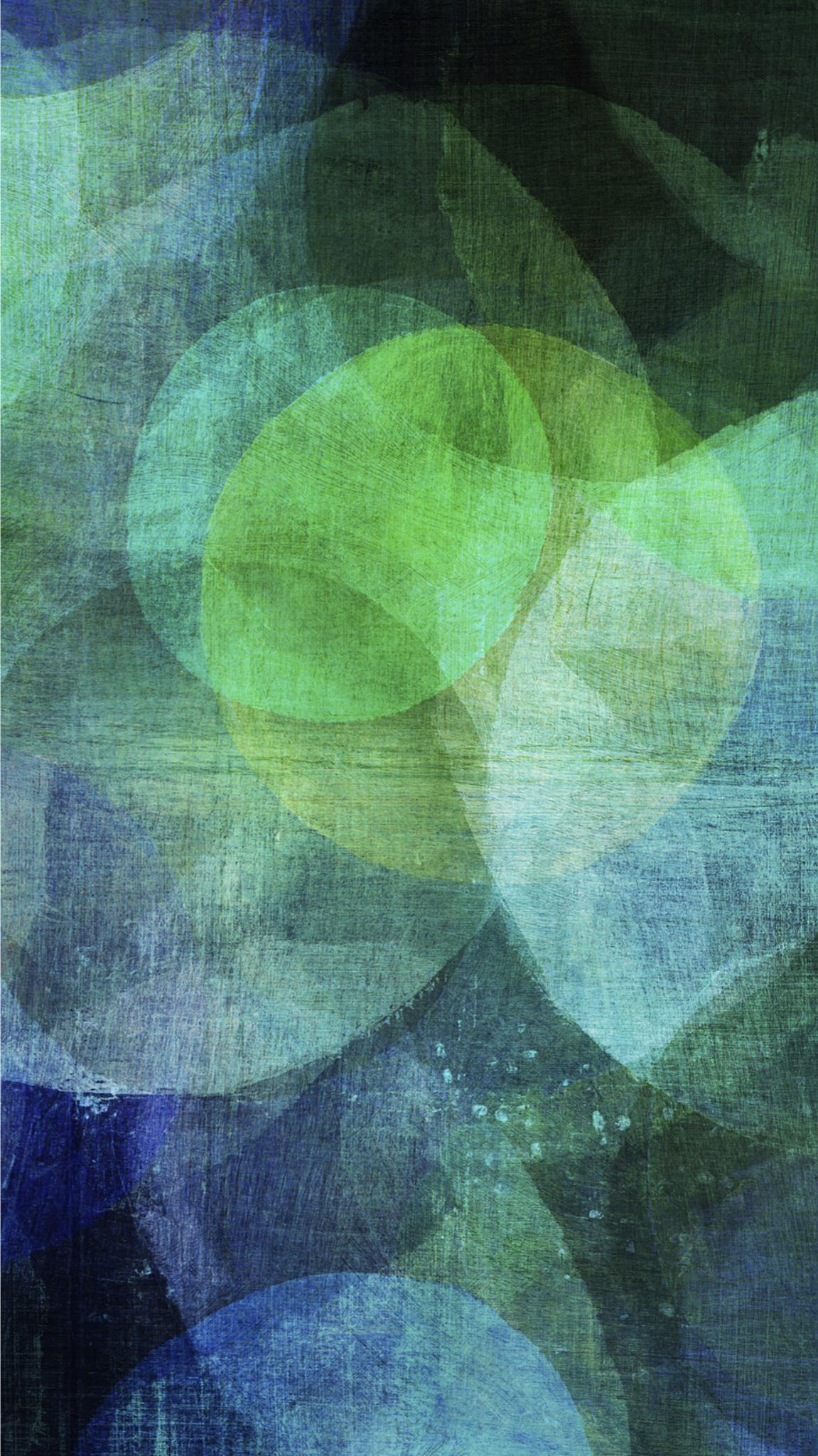
# AGENDA

---

- Por que programar em paralelo?
  - Paralelismo de Hardware
  - Paralelismo de Software
  - Medidas de Desempenho
- OpenMP
  - O que é?
  - Como usar?

# POR QUE PROGRAMAR EM PARALELO?

---



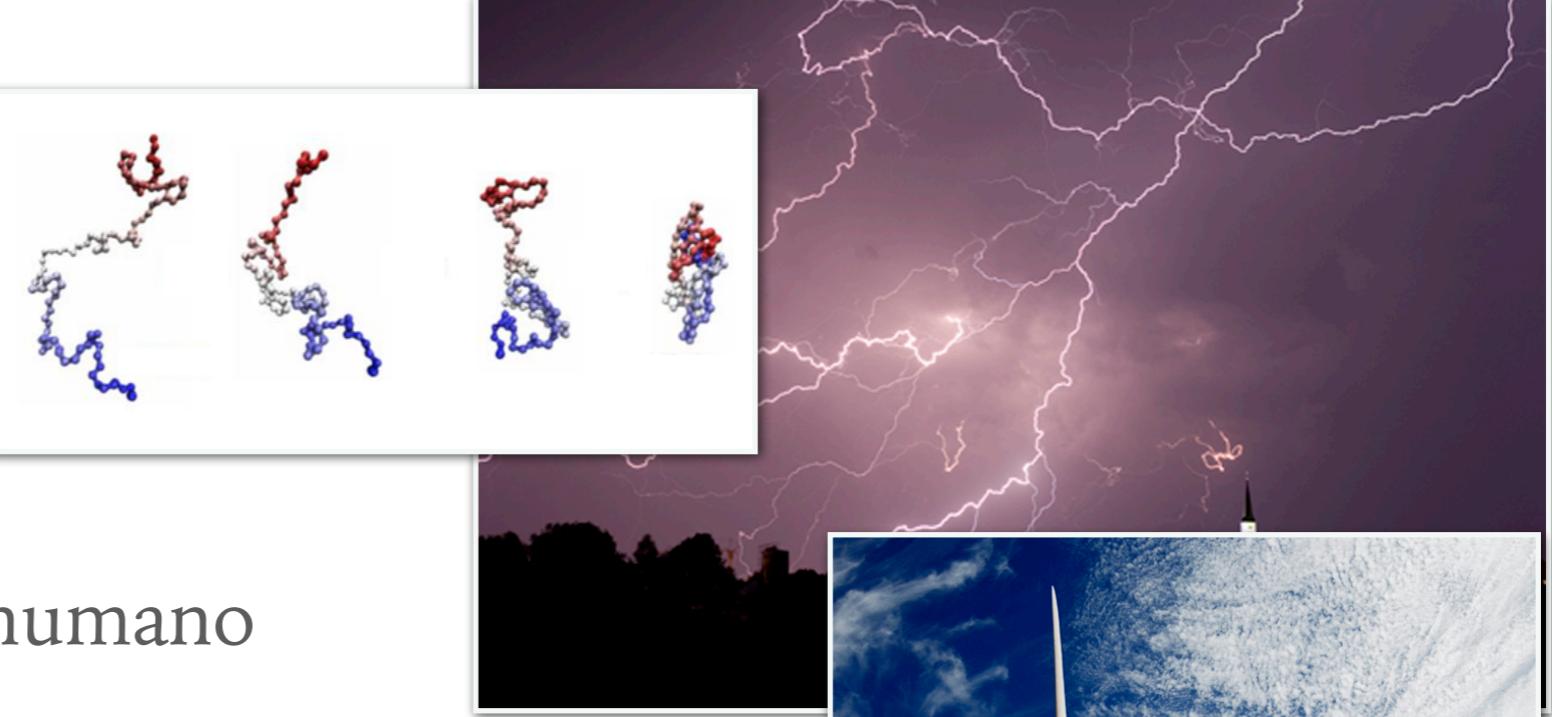
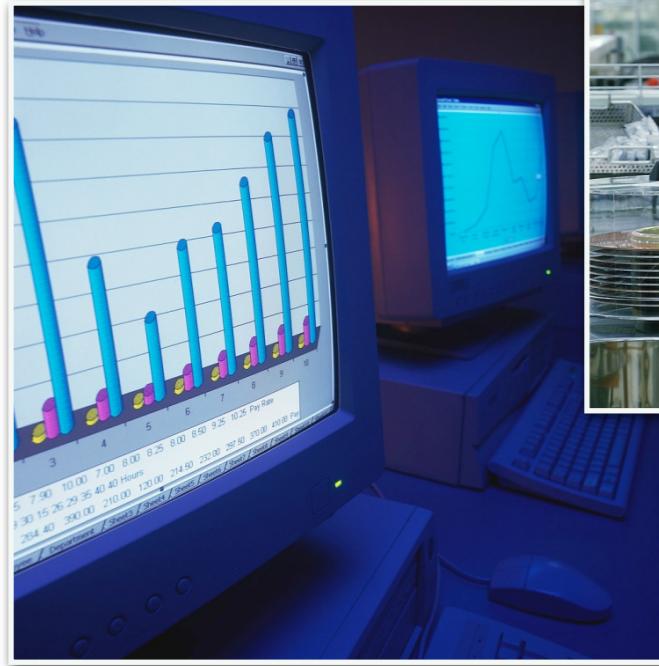
“

The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it.

*-Steve Jobs, Apple*

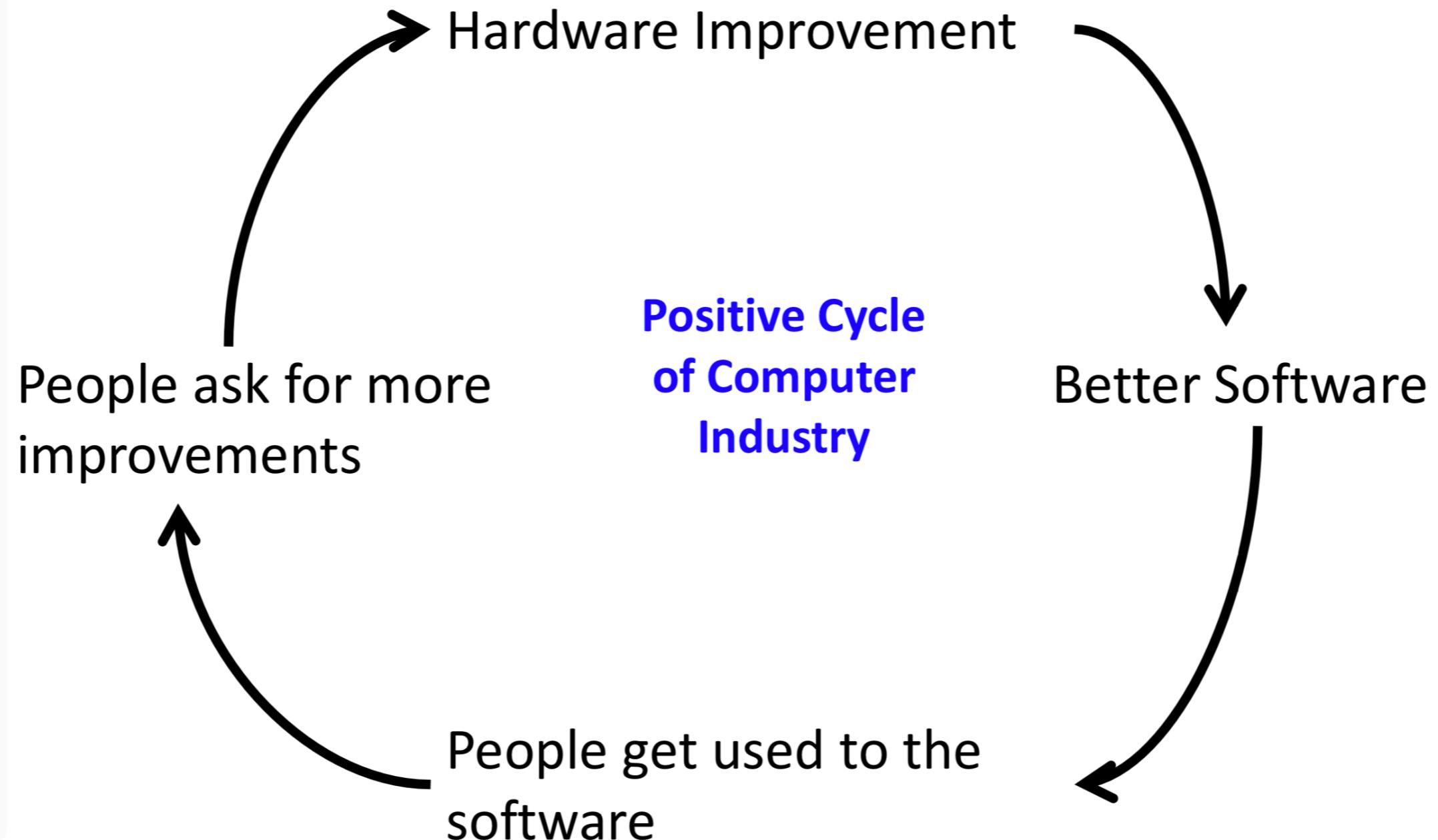
# MOTIVAÇÕES

- Jogos mais realistas
- Simulação numérica
  - Decodificar genoma humano
  - Aplicações médicas mais precisas
  - Modelagem de previsão de tempo
- Big data



# CICLO DA INDÚSTRIA

---



# COMO TORNAR O COMPUTADOR MAIS RÁPIDO?

---

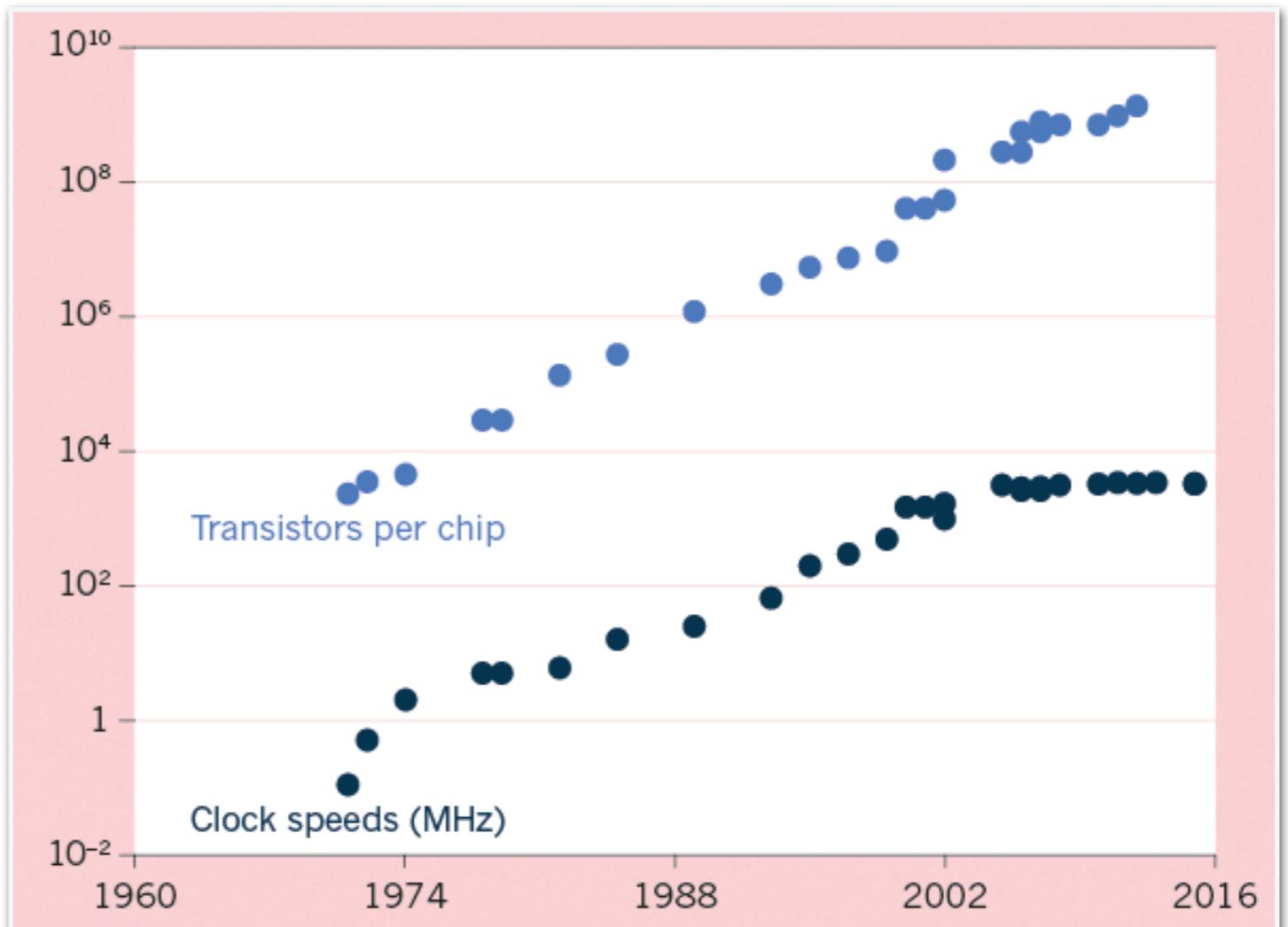
- Aumentar desempenho do core
  - Aumentar velocidade de clock
- Melhorar tempo de acesso a dados
  - Aumentar tamanho da cache
  - Melhorar acesso (latência)
- Usar computação paralela
  - Memória compartilhada
  - Memória distribuída

# LEI DE MOORE

---

- Capacidade de transistores dos processadores dobra a cada 18 meses<sup>1</sup>

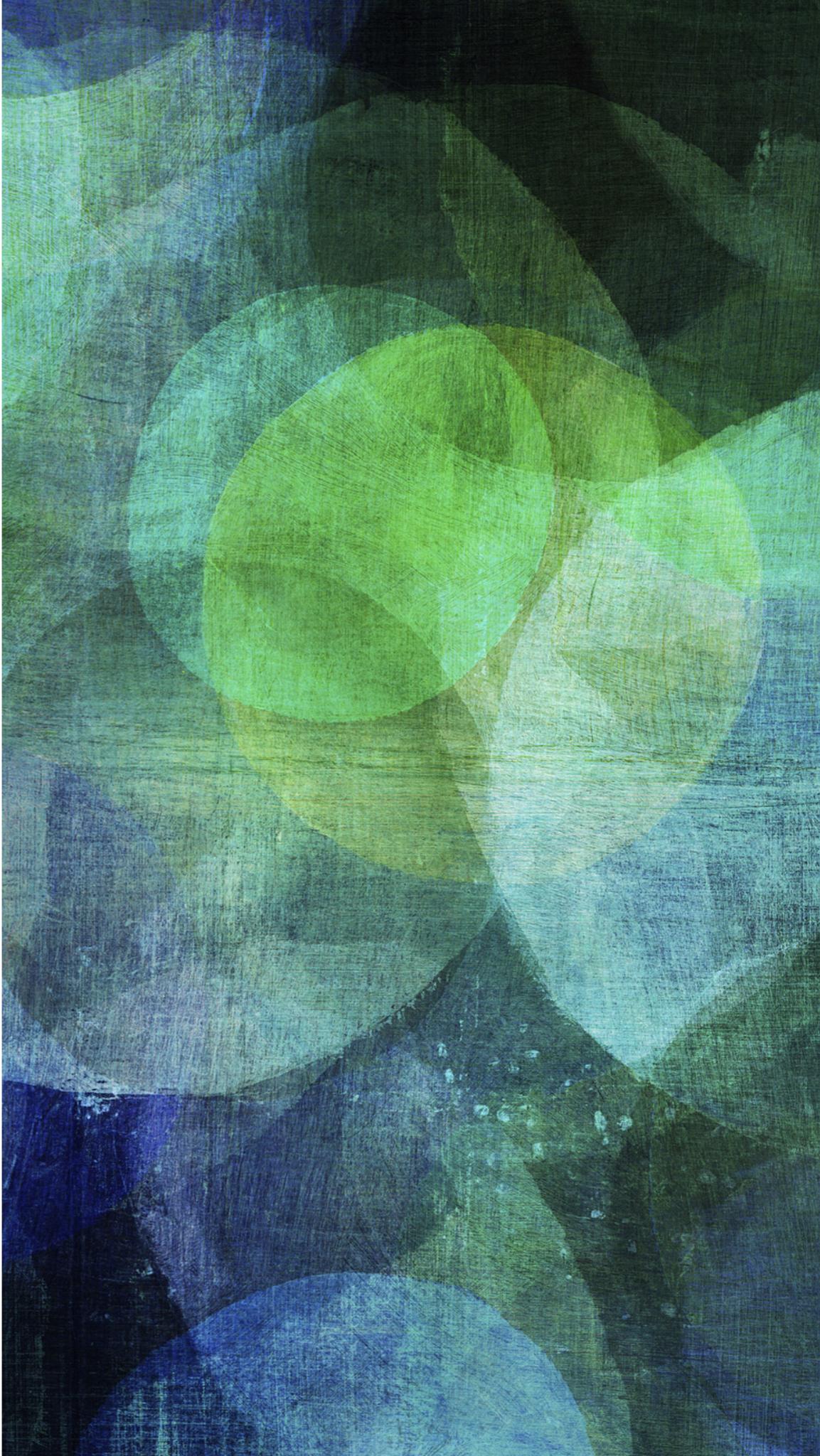
- Mais desempenho, menos calor dissipado
- Sinais que miniaturização está chegando ao fim
- Silício começa a perder propriedades físicas



<sup>1</sup><https://canaltech.com.br/hardware/estamos-chegando-ao-fim-da-lei-de-moore-57693/>

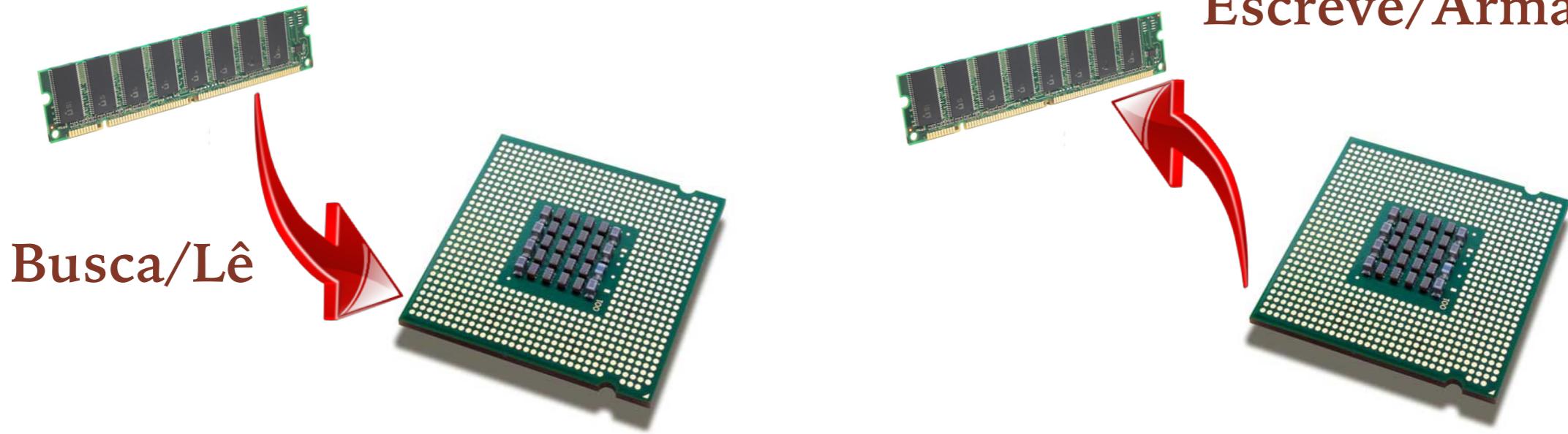
# PARALELISMO DE HARDWARE

---



# CPU E MEMÓRIA

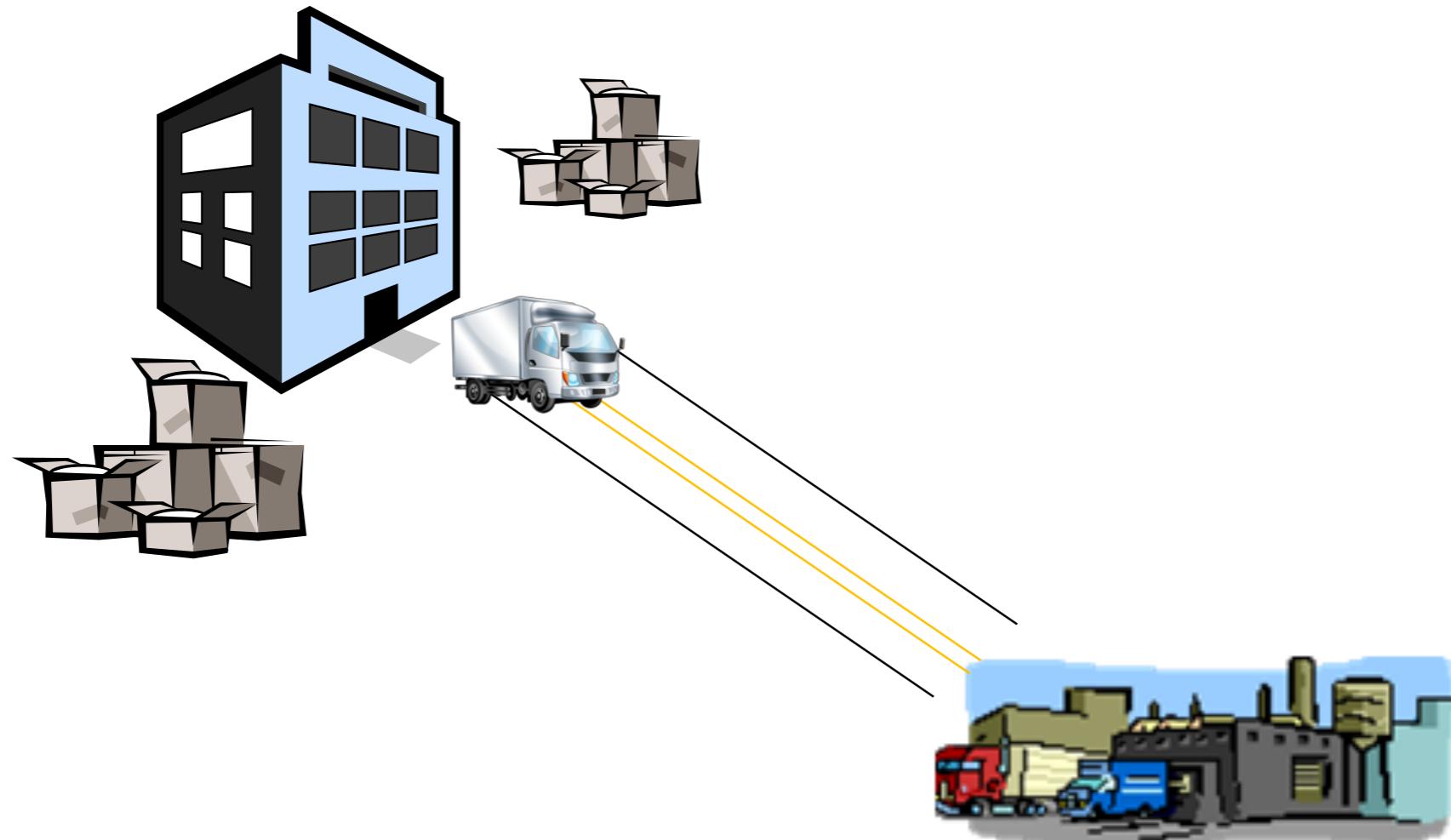
---



# GARGALO DA ARQUITETURA DE VON NEUMANN

---

- Limite de comunicação



# CACHING

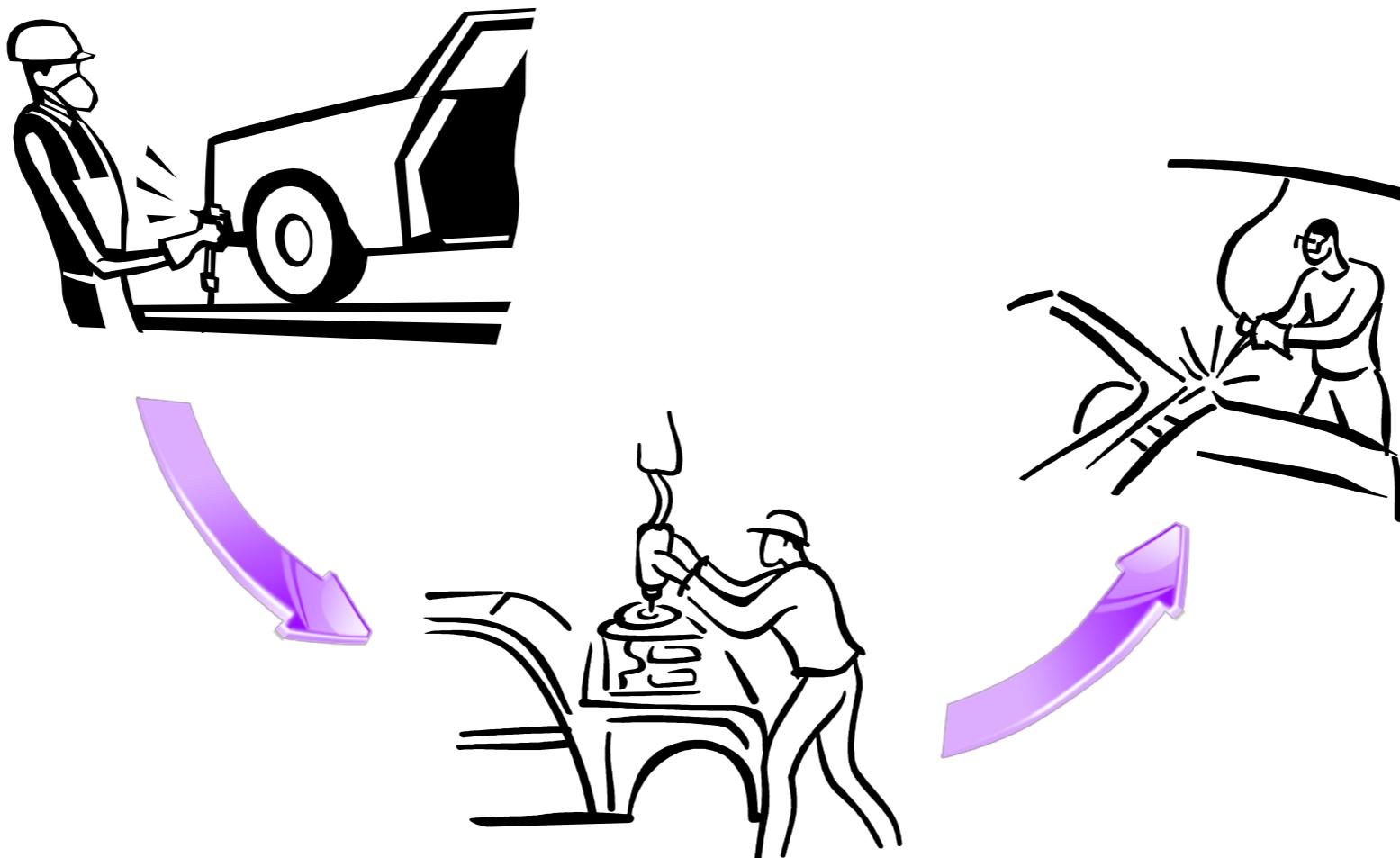
---

- Princípio de localidade
  - Espacial
    - Acessa local próximo
  - Temporal
    - Acessa novamente em futuro próximo
- Problemas
  - cache inconsistente com memória

# PARALELISMO NO NÍVEL DE INSTRUÇÃO

---

- Pipelining
- Unidade funcionais organizadas em estágios



# PARALELISMO NO NÍVEL DE INSTRUÇÃO

---

## ► Pipelining

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
:	:	:	:	:	:	:	:
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

# PARALELISMO DE HARDWARE

---

## ► Taxonomia de Flynn

<p><i>classic von Neumann</i></p> <p>SISD Single instruction stream Single data stream</p>	<p>(SIMD) Single instruction stream Multiple data stream</p>
<p>MISD Multiple instruction stream Single data stream</p>	<p>(MIMD) Multiple instruction stream Multiple data stream</p>

# PARALELISMO DE HARDWARE

---

- GPUs - Graphics Processing Units
  - Geração atual usa SIMD
- GPGPU - GPU de Propósito Geral
  - Pode atuar em conjunto com CPU
  - Tem hierarquia de memória própria



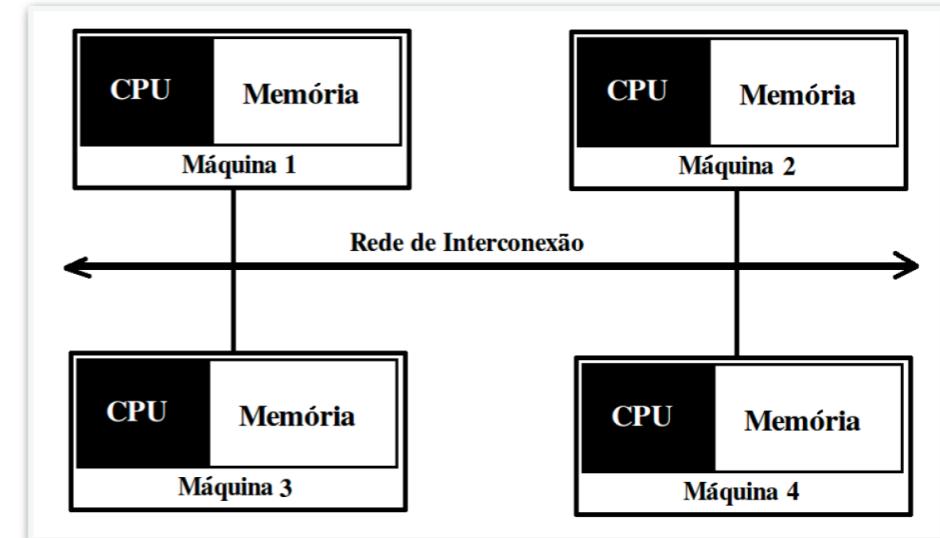
# ARQUITETURA

---

- A forma de implementar pode variar com arquitetura:

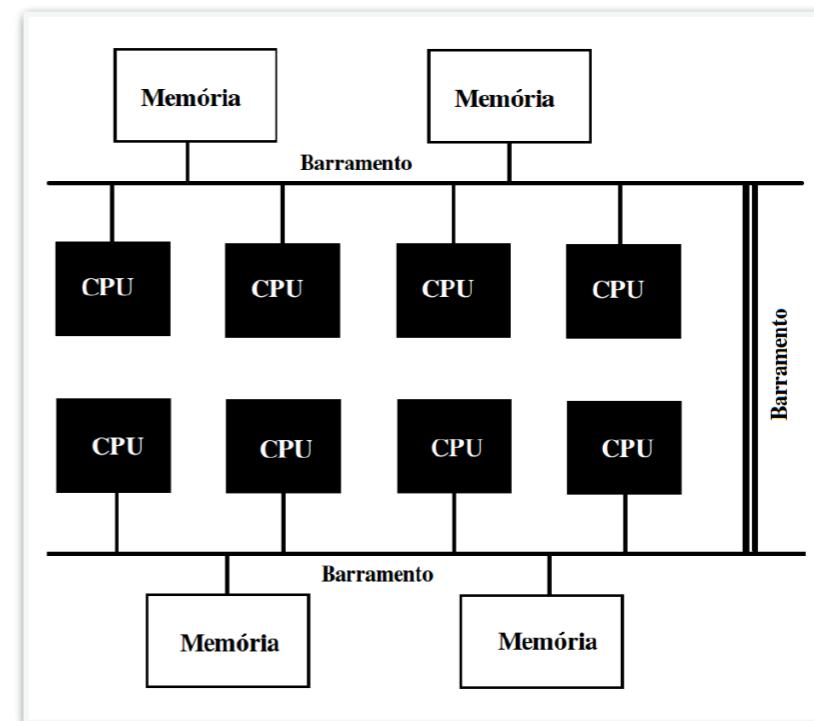
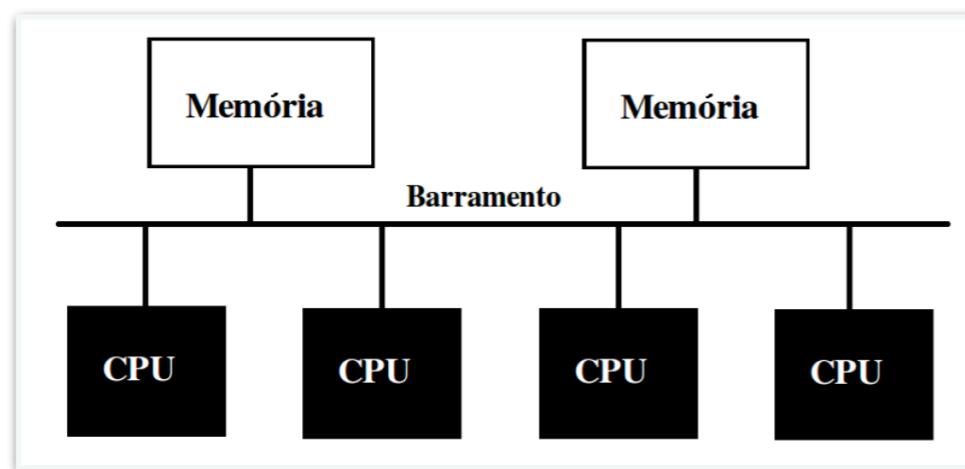
- Multicomputador

- Memória distribuída



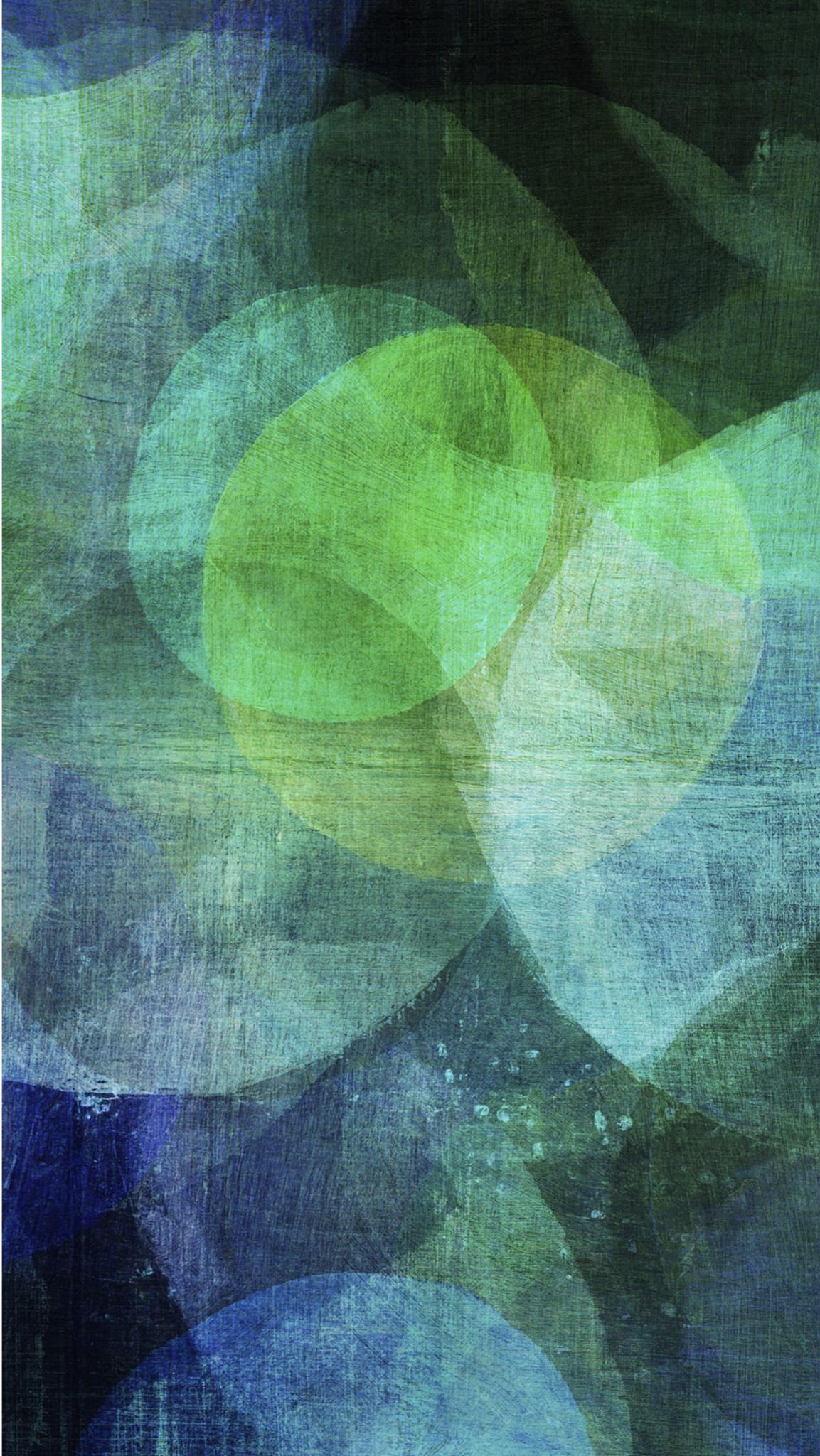
- Multiprocessador

- Memória compartilhada



# PARALELISMO DE SOFTWARE

---



# INTRODUÇÃO A PROGRAMAÇÃO PARALELA

---

- Tradicionalmente apps são desenvolvidos para executar instruções sequenciais
- Para reduzir tempo de computação:
  - Divide em partes que possam ser executadas simultaneamente
  - Núcleos distintos
  - Cada parte:
    - Processo ou fluxo de execução (thread)
  - Essa implementação é chamada de **programação paralela**

# INTRODUÇÃO A PROGRAMAÇÃO PARALELA

---

- Existem 2 modelos:
  - Troca de mensagens
    - Mecanismo explícito de comunicação
  - Memória compartilhada
    - Para passar dados basta copiar da área comum

# TROCA DE MENSAGENS

---

- Programador deve definir
  - Que dados devem ser enviados
  - Que momento ocorre comunicação
- Divide os processos em
  - Remetente
  - Destinatário
- Operação impõe **sincronização**

# TROCA DE MENSAGENS

---

- Programador deve definir
    - Que dados devem ser enviados
    - Que momento ocorre comunicação
  - Divide os processos em
    - Remetente
    - Destinatário
  - Operação impõe **sincronização**
  - Modelo compatível com
    - Multiprocessadores
  - Estabelece comunicação através da rede
- Obs: não impede de usar em multiprocessador

# TROCA DE MENSAGENS

---

- Padrão muito conhecido - MPI
  - *Message Passing Interface*
  - Define um conjunto de rotinas para facilitar comunicação entre processos (executando em paralelo)
  - Existem várias bibliotecas

# LINKS ÚTEIS SOBRE MPI

---

- <https://www.cs.usfca.edu/~peter/ppmpi/>
  
- <http://www.ufjf.br/getcomp/files/2013/03/Introdução-a-Computação-Paralela-com-o-OpenMPI.pdf>

# MEMÓRIA COMPARTILHADA

---

- Processos distintos compartilham mesma posição de memória
- Usa essa memória para comunicação
  - Copia dados da área comum
- Combina naturalmente com
  - Multiprocessadores
- Pode ser implementado em multicomputadores
  - Memória compartilhada distribuída

# MEMÓRIA COMPARTILHADA

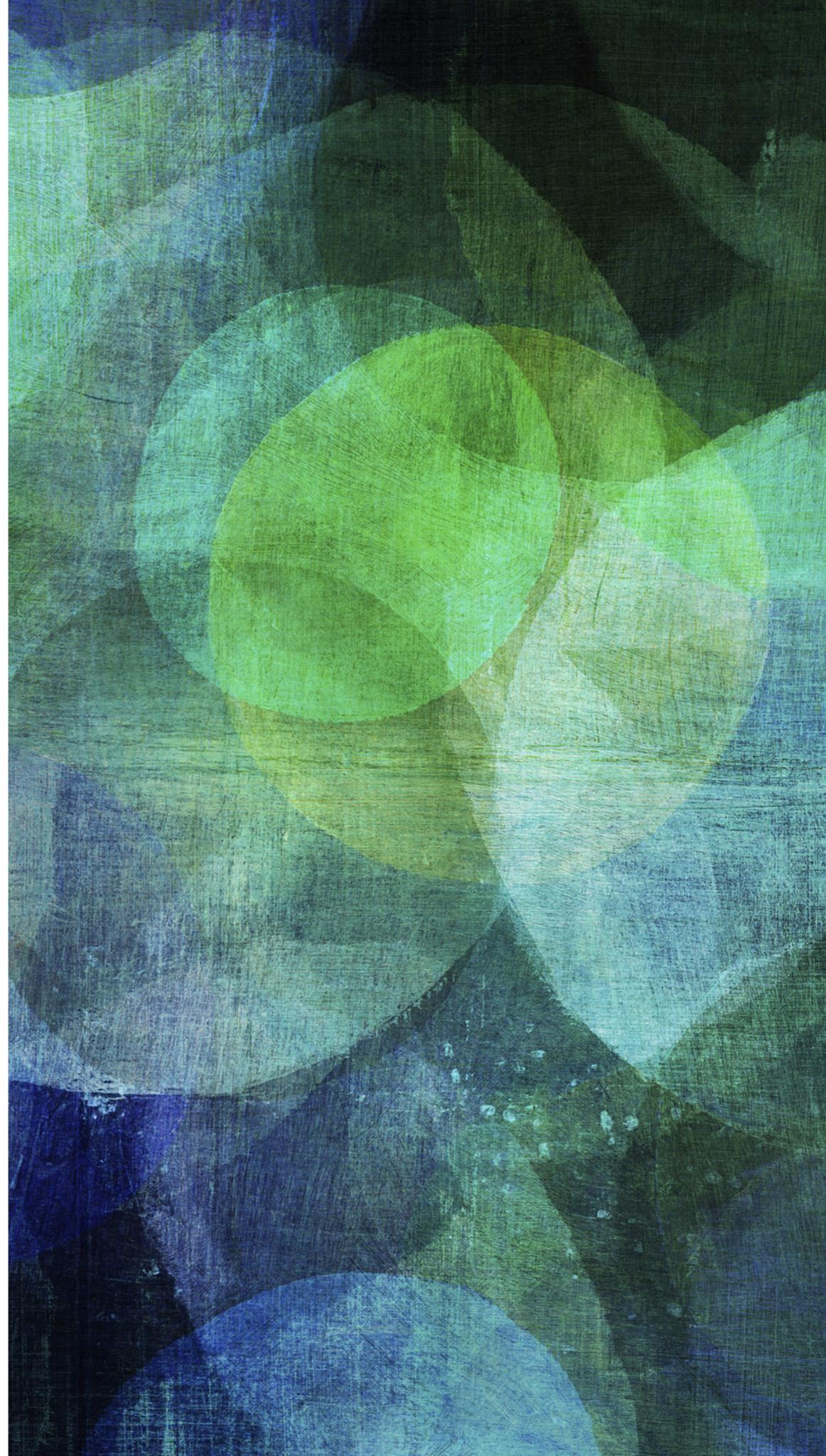
---

- Existem várias alternativas
- Veremos nesse curso:
  - Um padrão popular que utiliza diretivas de pré-compilação
    - OpenMP

# DESEMPENHO

---

*Speedup*



# MÉTRICAS DE DESEMPENHO

---

- *Speedup*
  - Quanto um algoritmo paralelo é mais rápido que o sequencial?
  - Speedup = tempo serial / tempo paralelo
  - Resultado varia de 0 (zero) a 1.
    - Se for = 1 **speedup linear**
    - Quanto mais perto de zero, maior o ganho
    - Se for maior que 1 reveja
      - Gargalos?

# MÉTRICAS DE DESEMPENHO

---

- Lei de Amdahl
  - Desempenho limitado pela fração do tempo das partes paralelizáveis
  - Speedup =  $1 / (\text{fração sequencial}) + (1 - (\text{fração sequencial}/\text{numero de processadores}))$
  - Considerada pessimista

# MÉTRICAS DE DESEMPENHO

---

- Lei de Gustafon-Barsis
- Speedup = num processadores - fração sequencial (num processadores - 1)

# MÉTRICAS DE DESEMPENHO

---

- Eficiência

$$E = \frac{S}{p} = \frac{\left( \frac{T_{\text{serial}}}{T_{\text{parallel}}} \right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}$$

# MÉTRICAS DE DESEMPENHO

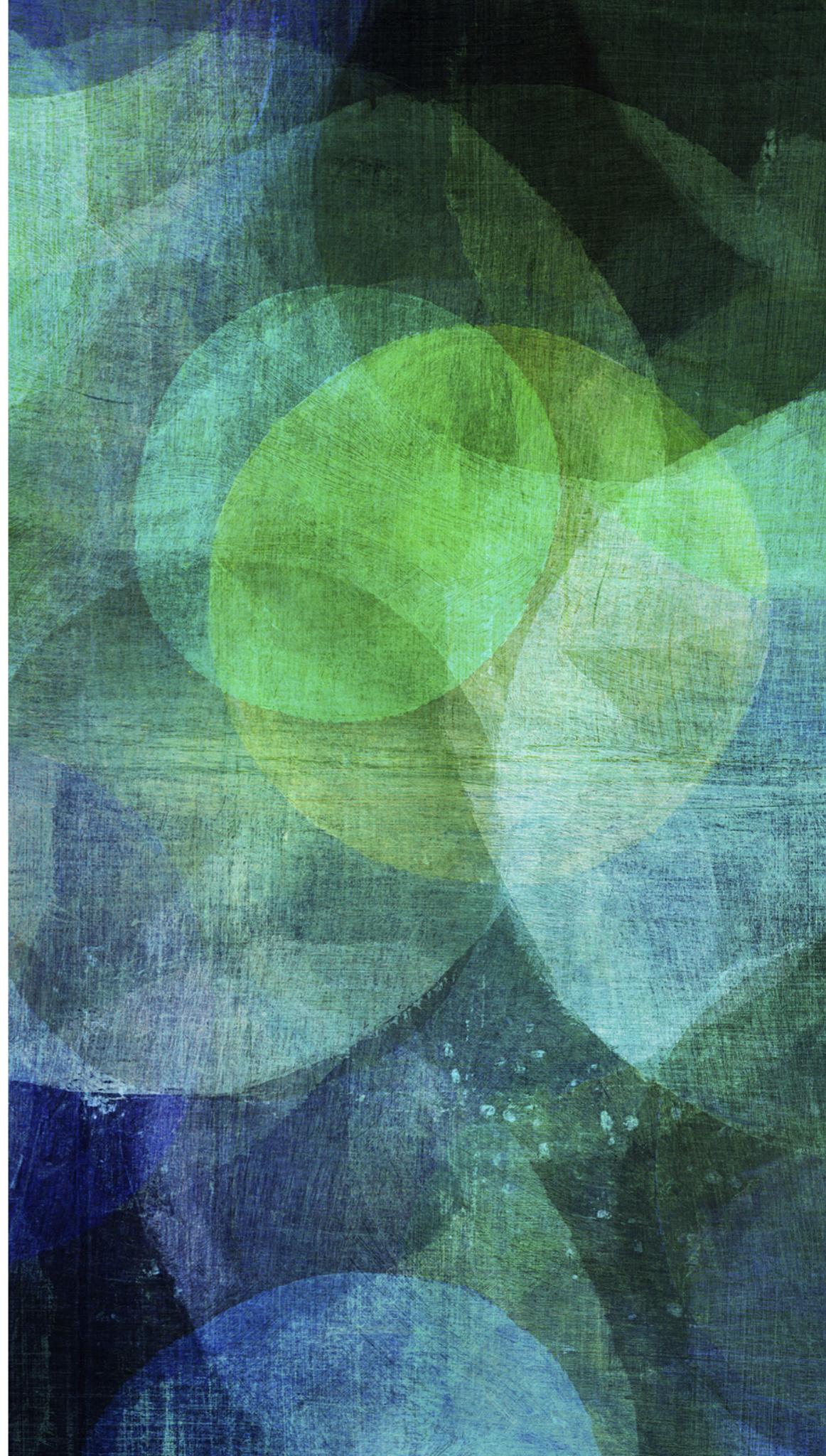
---

- Exemplo de speedup e eficiência:

$p$	1	2	4	8	16
$S$	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

# PROJETO DE PROGRAMA PARALELO

---



# METODOLOGIA DE FOSTER

---

## ➤ Particionamento

- Divide o que deve ser computado e os dados em tarefas pequenas
- Objetivo : identificar o que pode ser executado em paralelo

## ➤ Comunicação

- Determina que tipo de comunicação deve ser feita entre as tarefas

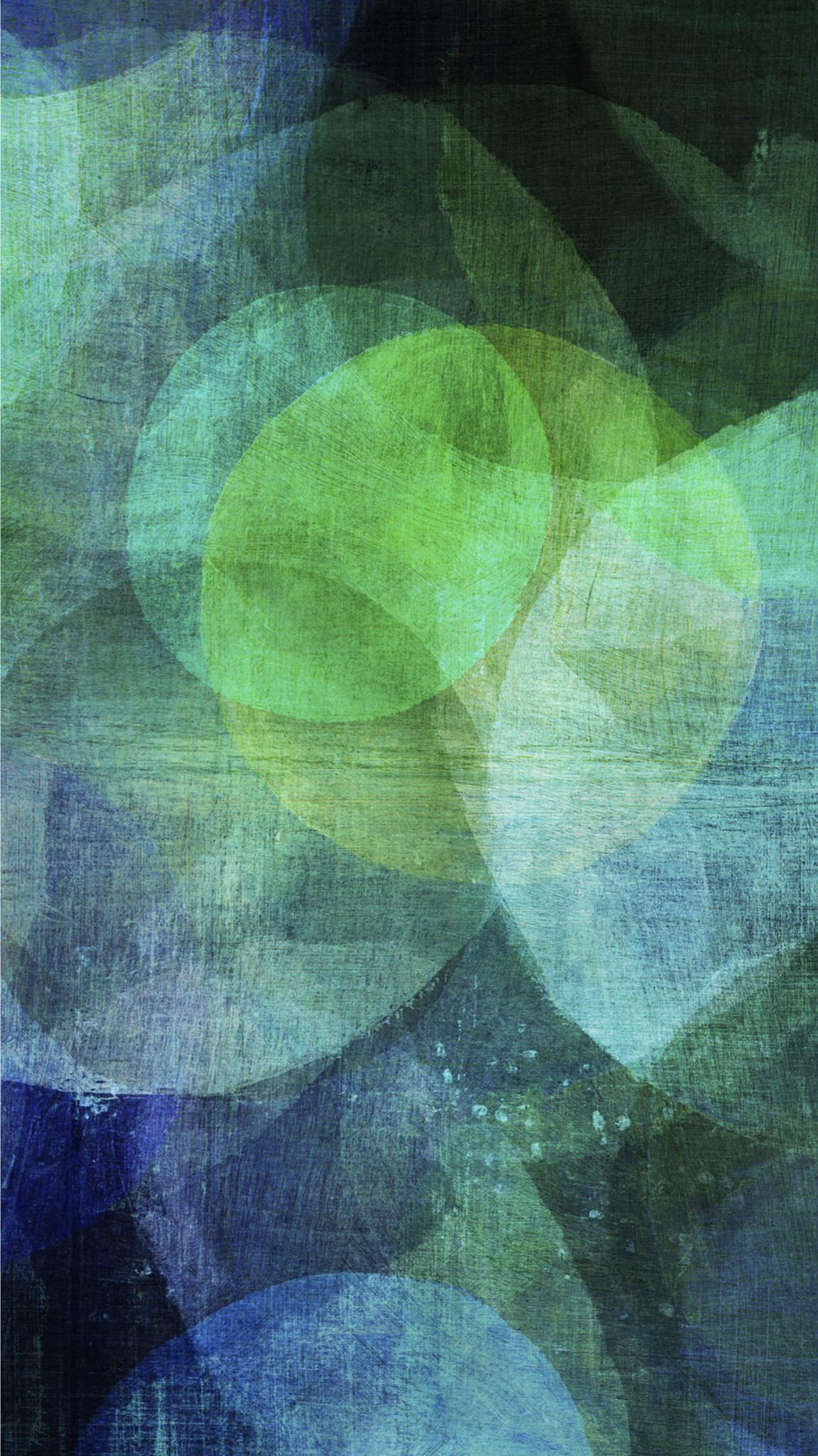
# METODOLOGIA DE FOSTER

---

- **Aglomerar ou agregação**
  - Combina tarefas e comunicações identificadas em tarefas maiores
  - Ex:
    - Se tarefa A precisa ser executada antes da B faz sentido que seja agregadas em uma tarefa só
- **Mapeamento**
  - Distribui tarefas entre os processadores/threads
  - **Objetivo :** minimizar comunicação e boa distribuição de carga de trabalho

# OPENMP

---



# O QUE É OPENMP?

---

- Open Multi processing
  - Multi-processamento aberto
- Um dos modelos de programação paralela mais usados<sup>1</sup>
- Relativamente fácil de usar
  - Diretivas de compilação
  - C/C++/Fortran
  - Funciona na maioria das arquiteturas e SO

<sup>1</sup> Matheus S. Serpa, Vinícius G. Pinto, Philippe O. A. Navaux , ERAD 2018

# OPENMP – VANTAGENS

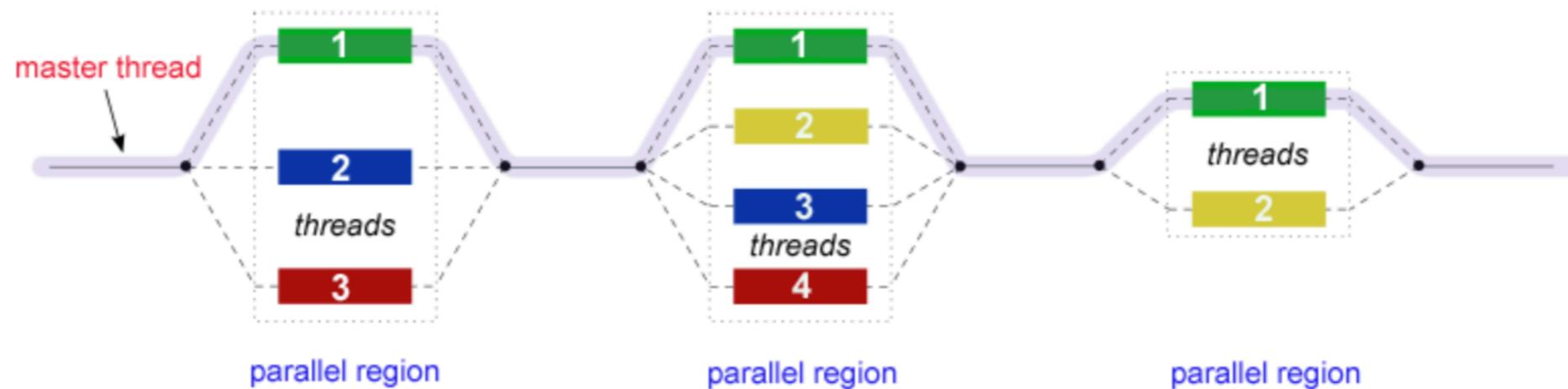
---

- Precisa de poucas alterações em código existente
- Fácil compreensão
- Suporta paralelismo aninhado
- Ajuste dinamico do numero de threads

# OPENMP - COMO FUNCIONA?

---

- Modelo fork - join



- Começa com uma thread única (mestre)
- Executa sequencialmente até a primeira região paralela
- **FORK** - thread mestre cria time de threads paralelas
- **JOIN** - quando o time completa, sincroniza e termina deixando apenas a thread mestre.

# OPENMP - COMO FUNCIONA?

---

- Como é um modelo de programação de memória compartilhada
  - Por padrão a memória em uma região paralela é compartilhada
  - Todas as threads podem acessar a mesma região simultaneamente
- Permite que se defina escopo da memória!

Cuidado!!!

# OPENMP - ALO MUNDO!

---

- Vamos testar um primeiro exemplo:
- Abra o Visual Studio Code
- Crie uma pasta para o minicurso
- Crie um programa para imprimir "Alo Mundo!" em c  
(AloMundo.c)

# OPENMP - ALO MUNDO!

---

- AloMundo.c

```
1 #include <stdio.h>
2
3 int main(){
4
5     printf("Alo Mundo!\n");
6
7 }
```

- Compile: gcc AloMundo.c -o alo
- Execute: ./alo

# OPEN MP - ALO MUNDO!

---

- Agora modifique o programa incluindo a diretiva

`#pragma omp parallel`

```
1 #include <stdio.h>
2
3 int main(){
4
5     #pragma omp parallel
6     printf("Alo Mundo!\n");
7
8 }
```

# OPEN MP - ALO MUNDO!

---

- Agora modifique o programa incluindo a diretiva

`#pragma omp parallel`

```
1 #include <stdio.h>
2
3 int main(){
4
5     #pragma omp parallel
6     printf("Alo Mundo!\n");
7
8 }
```

- Salve o arquivo com o nome Exemplo1.c

# OPEN MP - ALO MUNDO!

---

- Compile: gcc-8 -fopenmp Exemplo1.c Ex1
- Execute: ./Ex1

```
1 #include <stdio.h>
2
3 int main(){
4
5     #pragma omp parallel
6     printf("Alo Mundo!\n");
7
8 }
```

A diretiva faz com que o fluxo se divida nesse ponto

- Quantas vezes imprimiu o texto?

# OPENMP PRAGMA

---

- Instrução de pre-processamento especial
- Ignorada por compilador que não suporta

`#pragma omp parallel`

- Diretiva paralela mais básica
- Número de threads simultâneas determinada pelo sistema

# OPEN MP - NÚMERO DE THREADS

---

- Quantos cores tem disponível?
- Pode definir número de threads?

# OPEN MP – NUMERO DE THREADS

---

- Quantos cores tem disponível?
- Pode definir número de threads? Sim!

```
1 #include <stdio.h>
2
3 int main(){
4
5     #pragma omp parallel num_threads(2)
6     printf("Alo Mundo!\n");
7
8 }
```



# OPEN MP - NUMERO DE THREADS

---

- Salve como Exemplo2.c
- Compile: gcc-8 -fopenmp Exemplo2.c -o Ex2
- Execute: ./Ex2

```
1 #include <stdio.h>
2
3 int main(){
4
5     #pragma omp parallel num_threads(2)
6     printf("Alo Mundo!\n");
7
8 }
```

# OPENMP - CLÁUSULAS

---

- Texto que modifica uma diretiva
- A cláusula num\_threads() pode ser adicionada
- Permite ao programador especificar o número de threads

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

# OPENMP - TERMINOLOGIA

---

- A thread original mais as novas formam um time
- A thread original é chamada mestre
- Threads novas são chamadas threads filhas

# OPENMP – MAIS DETALHES

---

- Possui 3 tipos de componentes:
  - Diretivas de compilação
  - Biblioteca de Rotinas
  - Variáveis de ambiente

# OPENMP – MAIS DETALHES – DIRETIVAS DE COMPILAÇÃO

---

- As diretivas de compilação do OpenMP pode ser usadas para:
  - Definir uma região paralela
  - Dividir blocos de código entre threads
  - Distribuir iterações de laços entre threads
  - Serializar seções de código
  - Sincronizar trabalho entre as threads

# OPENMP – MAIS DETALHES – BIBLIOTECA DE ROTINAS

---

- OpenMP inclui uma série de **rotinas** que podem ser usadas para:
  - Definir e consultar número de threads
  - Consultar identificador único de uma thread
  - Consultar se está em região paralela e em qual nível
  - Definir, inicializar e terminar travamentos (locks)
  - Consultar tempo de relógio
- **Obs:** para C/C++ geralmente precisa incluir `<omp.h>`

# OPENMP – SUPORTE

---

- Caso compilador não suporte

```
#ifdef _OPENMP  
# include <omp.h>  
#endif
```

# OPENMP – MAIS DETALHES – VARIÁVEIS DE AMBIENTE

---

- Variáveis de ambiente podem controlar:
  - Definição do número de threads
  - Especificar como laços são divididos
  - Ligar threads a processadores
  - habilitar/desabilitar paralelismo aninhado
  - habilitar/desabilitar threads dinâmicas
  - Definir tamanho da pilha de threads
  - Definir política de espera da thread

# OPENMP

---

- Cada thread possui um identificador
- Vamos imprimir o número de cada thread?

```
1 #include <stdio.h>
2 #include <omp.h>
3 int main(){
4     #pragma omp parallel
5         printf("Alo Mundo da thread %d de um total de %d!\n",
6                 omp_get_thread_num(), omp_get_num_threads());
7     return 0;
8 }
```

- Execute várias vezes e observe os números impressos.
- Tente definir um número de threads e observe.

# OPENMP - CALLS

---

## ► Chamadas comuns:

Chamada	Descrição
<code>int omp_get_num_threads()</code>	Retorna o número de threads no time concorrente
<code>int omp_get_thread_num()</code>	Retorna o id da thread dentro do time
<code>int omp_get_num_procs()</code>	Retorna o número de processadores na máquina
<code>int omp_get_max_threads()</code>	Retorna o número máximo de threads que serão usadas na próxima região paralela
<code>double omp_get_wtime()</code>	Retorna o número de segundos que passou
<code>bool omp_in_parallel()</code>	Retorna 1 se em região paralela e 0 caso contrário

# OPENMP - CALLS

---

- Vamos testar algumas chamadas:

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main(){
5     double inicio = omp_get_wtime();
6     if(!omp_in_parallel()){
7         printf("Numero de processadores: %d\n", omp_get_num_procs());
8         printf("Numero maximo de threads: %d\n", omp_get_max_threads());
9     }
10    double final = omp_get_wtime();
11    printf("inicio = %.16g\nfinal = %.16g\ndiff = %.16g\n",
12          inicio, final, final-inicio);
13    return 0;
14 }
```

SEMANA ACADÊMICA UNIFICADA DE

**ENGENHARIA DE SOFTWARE**

**& SISTEMAS DE INFORMAÇÃO**

# INTRODUÇÃO A PROGRAMAÇÃO PARALELA

---

*Por Prof. DSc Bárbara Quintela*  
[barbara@ice.ufjf.br](mailto:barbara@ice.ufjf.br)

*Parte 1 - 25/10/2018*