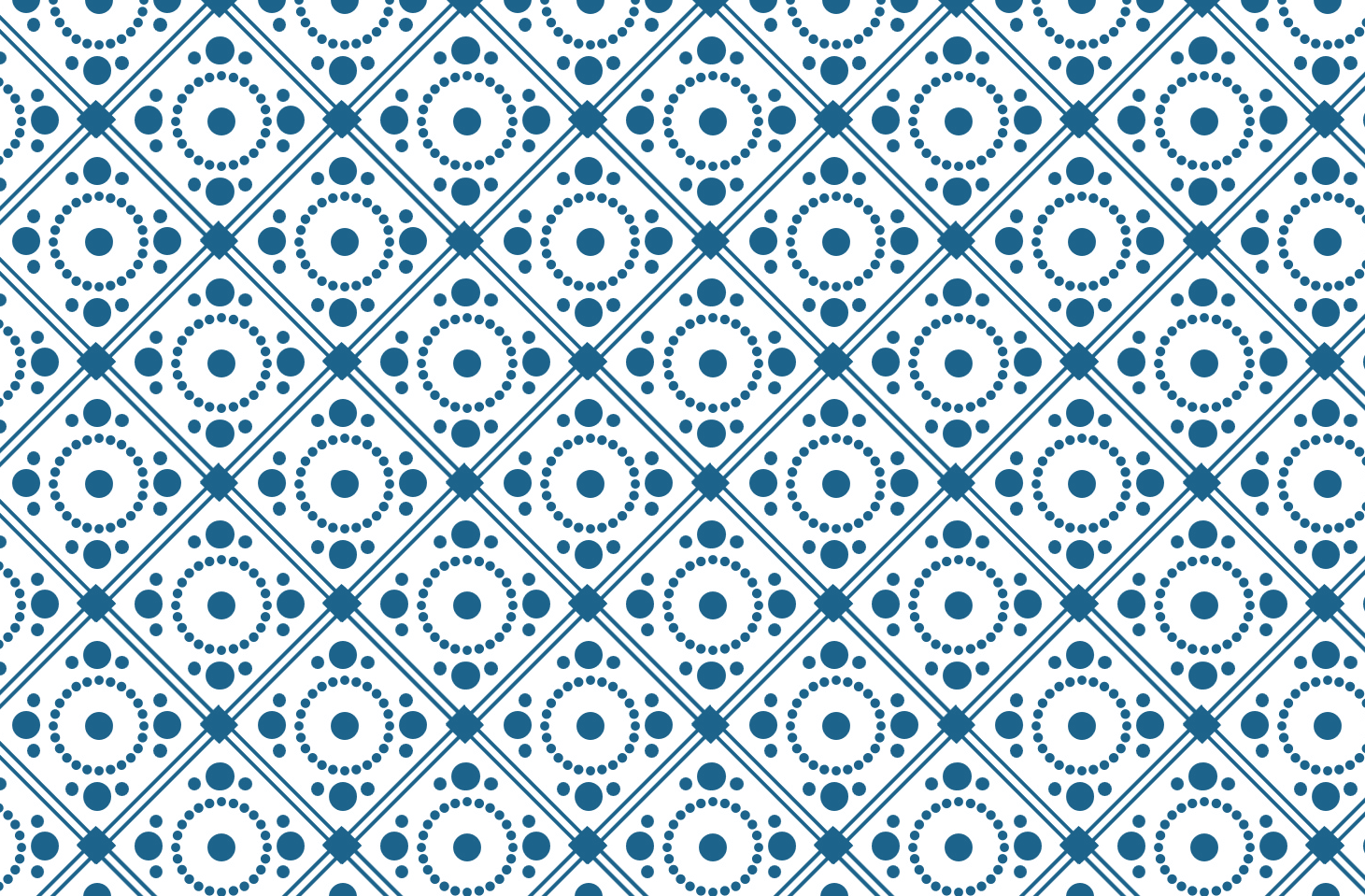




# INTEL MODERN CODE PARTNER

## OPENMP — AULA 01

# 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.

## Premissas:

- Assumo que todos sabem programar em linguagem C. OpenMP também suporta Fortran e C++, mas vamos nos restringir a C.
- Assumo que todos são novatos em programação paralela.
- Assumo que todos terão acesso a um compilador que suporte OpenMP

# AGRADECIMENTOS

Esse curso é baseado em uma longa série de tutoriais apresentados na conferência Supercomputing.

As seguintes pessoas prepararam esse material:

- **Tim Mattson (Intel Corp.)**
- J. Mark Bull (the University of Edinburgh)
- Rudi Eigenmann (Purdue University)
- Barbara Chapman (University of Houston)
- Larry Meadows, Sanjiv Shah, and Clay Breshears (Intel Corp).

Alguns slides são baseados no curso que Tim Mattson ensina junto com Kurt Keutzer na UC Berkeley.

- O curso chama-se “CS194: Architecting parallel applications with design patterns”.



Tim Mattson

Senior Research Scientist, Intel

# PRELIMINARES

Nosso plano ... Active learning!

Iremos misturar **pequenos comentários** com **exercícios curtos**

Por favor, sigam essas regras básicas:

- **Faça os exercícios** propostos e então **mude algumas coisas de lugar** e faça experimentos.
- Adote o **active learning**!
- Não trapaceie: **Não olhe as soluções antes de completar** os exercícios... mesmo que você fique muito frustrado.

# AGENDA GERAL

## Unit 1: Getting started with OpenMP

- Mod1: Introduction to parallel programming
- Mod 2: The boring bits: Using an OpenMP compiler (hello world)
- Disc 1: Hello world and how threads work

## Unit 2: The core features of OpenMP

- Mod 3: Creating Threads (the Pi program)
- Disc 2: The simple Pi program and why it sucks
- Mod 4: Synchronization (Pi program revisited)
- Disc 3: Synchronization overhead and eliminating false sharing
- Mod 5: Parallel Loops (making the Pi program simple)
- Disc 4: Pi program wrap-up

## Unit 3: Working with OpenMP

- Mod 6: Synchronize single masters and stuff
- Mod 7: Data environment
- Disc 5: Debugging OpenMP programs

## Unit 4: a few advanced OpenMP topics

- Mod 8: Skills practice ... linked lists and OpenMP
- Disc 6: Different ways to traverse linked lists
- Mod 8: Tasks (linked lists the easy way)
- Disc 7: Understanding Tasks

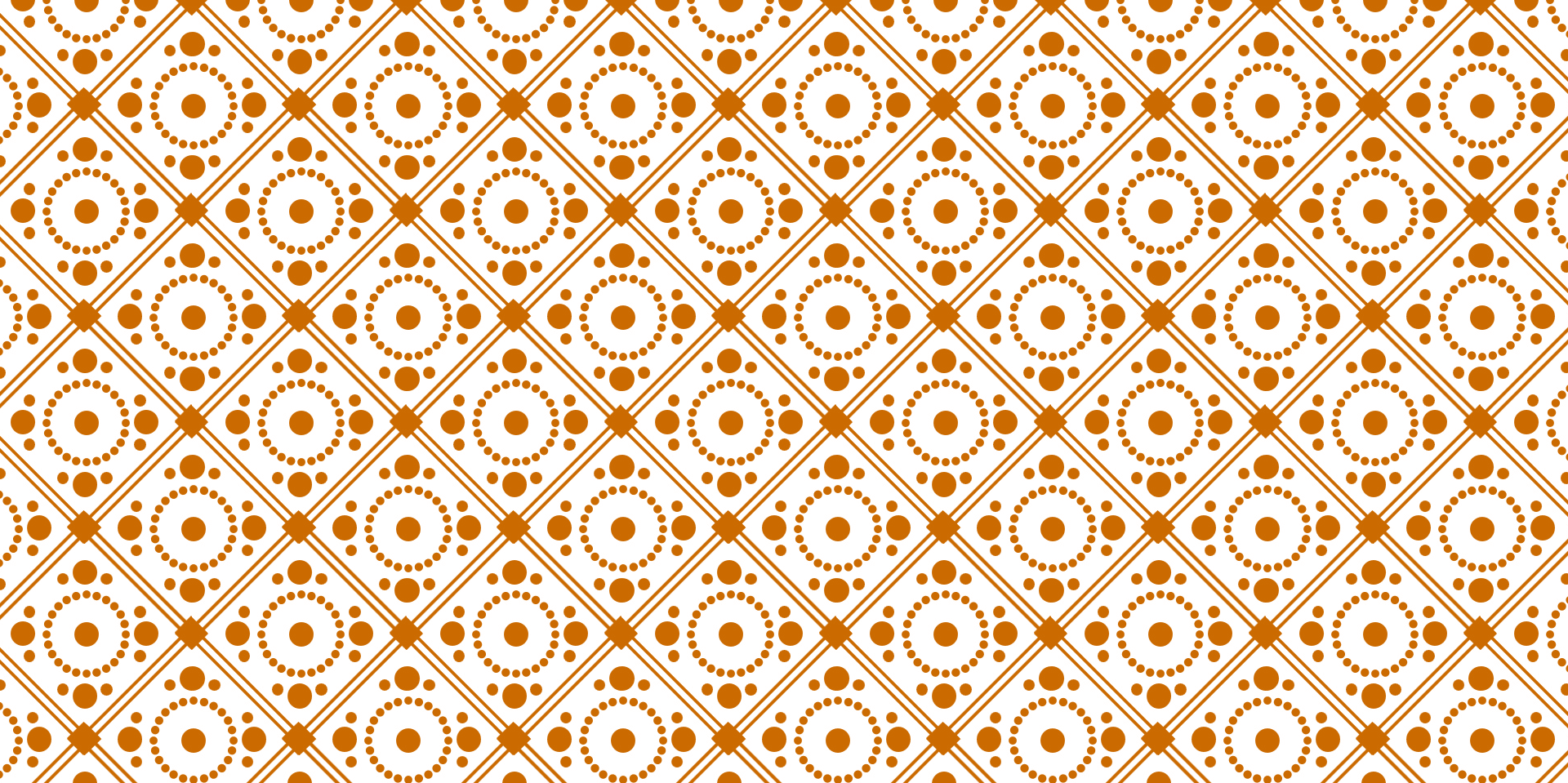
## Unit 5: Recapitulation

- Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
- Disc 8: The pitfalls of pairwise synchronization
- Mod 9: Threadprivate Data and how to support libraries (Pi again)
- Disc 9: Random number generators

# AGENDA DESSA AULA

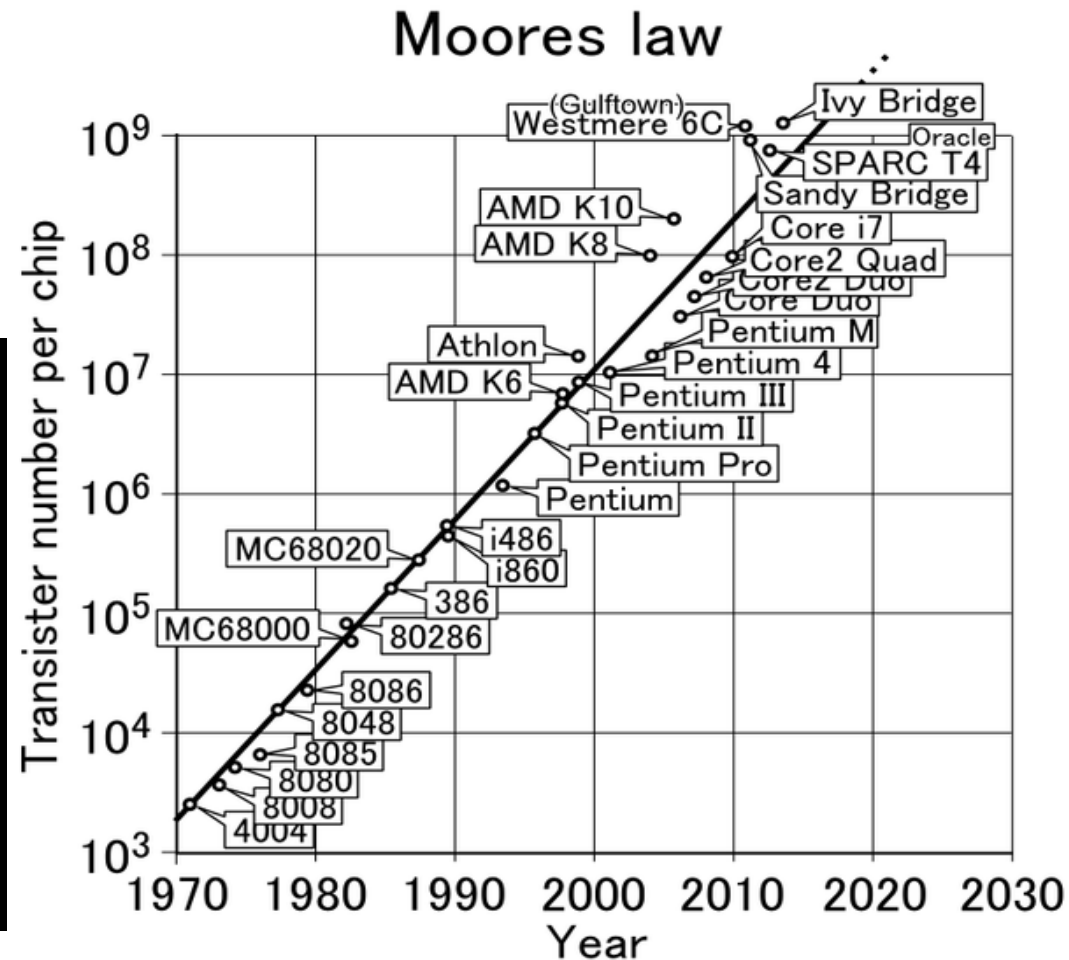
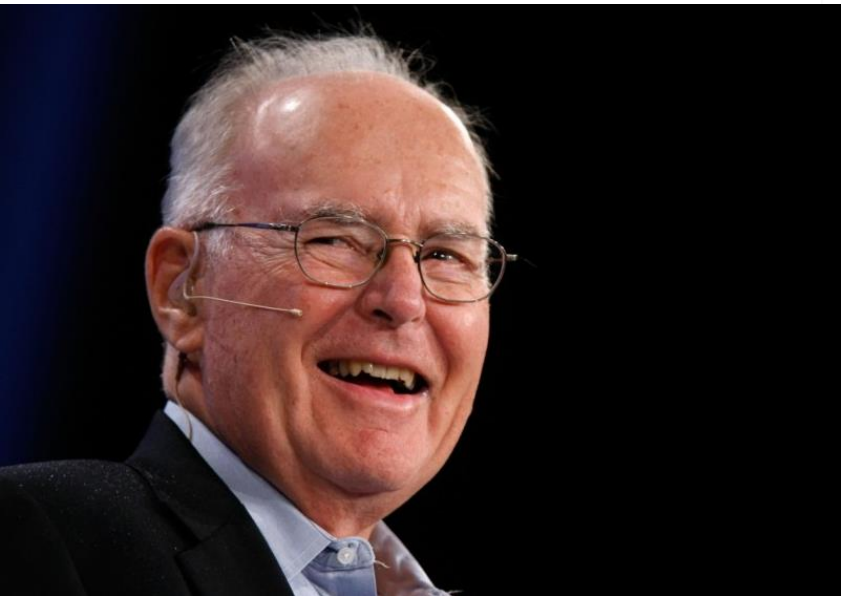
## Unit 1: Getting started with OpenMP

- Mod1: Introduction to parallel programming
- Mod 2: The boring bits: Using an OpenMP compiler (hello world)
- Disc 1: Hello world and how threads work



# INTRODUÇÃO A PROGRAMAÇÃO PARALELA

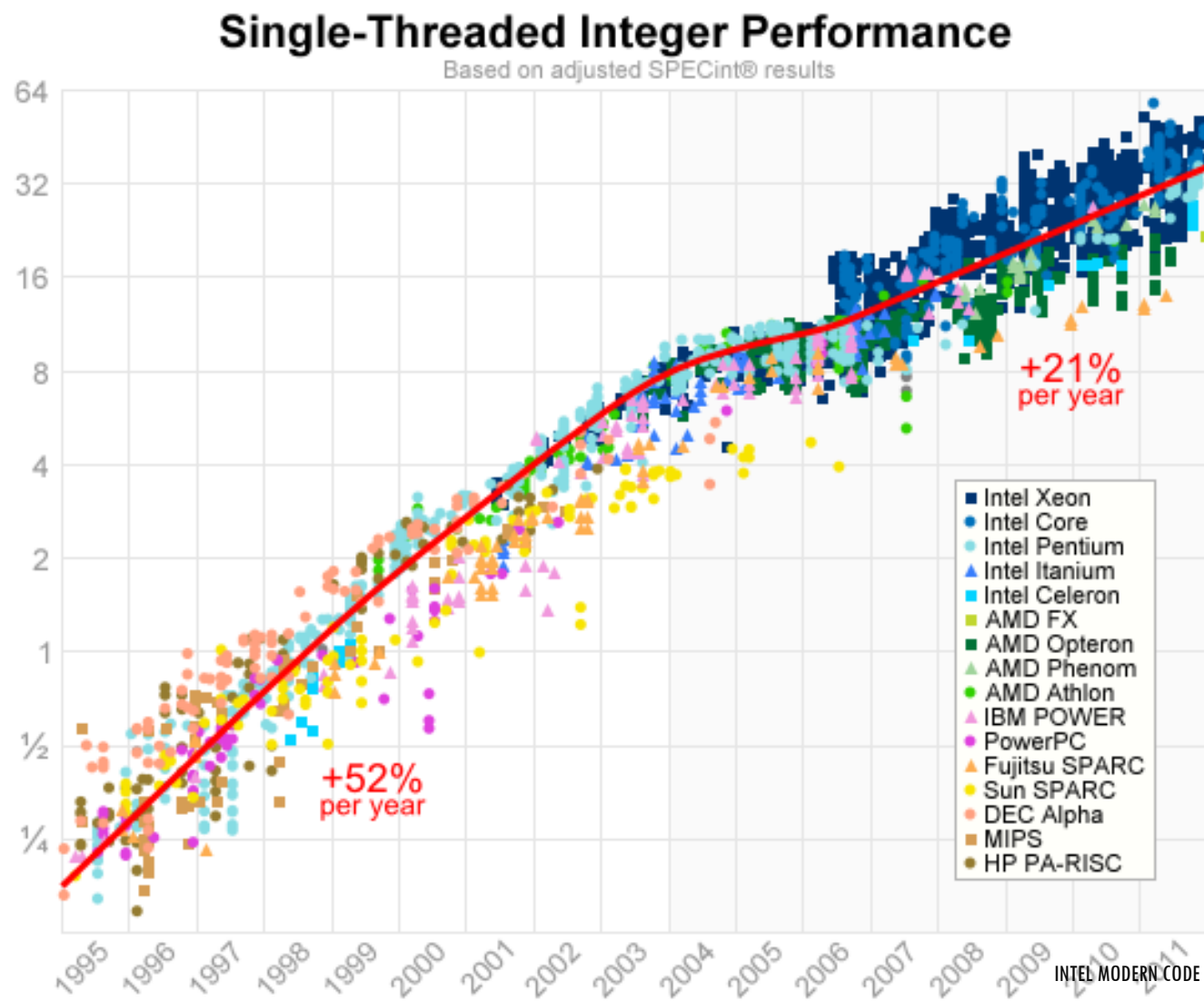
# LEI DE MOORE



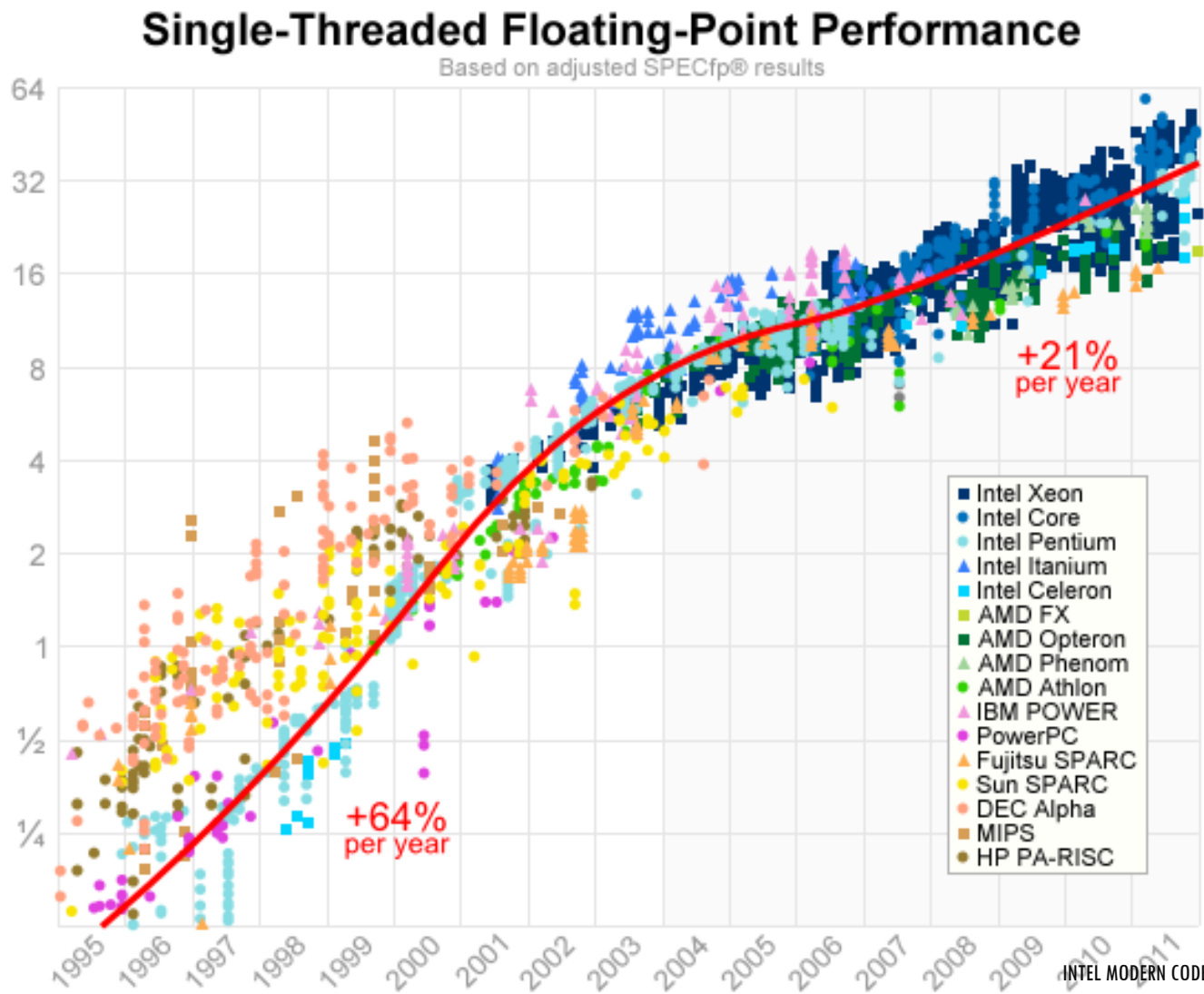
Em 1965, o co-fundador da Intel, Gordon Moore previu (com apenas 3 pontos de dados!) que a densidade dos semicondutores iria dobrar a cada 18 meses.

Ele estava certo!, Os transistores continuam diminuindo conforme ele projetou

# CONSEQUÊNCIAS DA LEI DE MOORE...



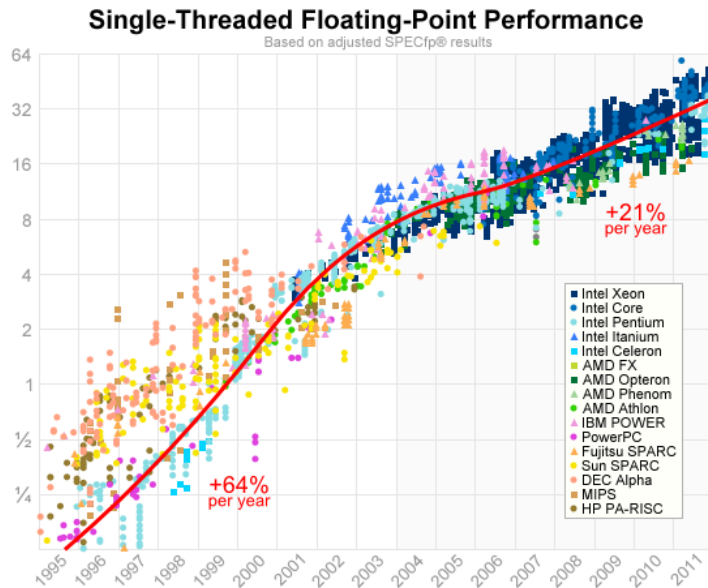
# CONSEQUÊNCIAS DA LEI DE MOORE...



# CONSEQUÊNCIAS DA LEI DE MOORE...

**Treinamos pessoas** dizendo que o desempenho viria do hardware.

Escreva seu programa como desejar e os Gênios-do-Hardware vamos cuidar do desempenho de forma “mágica”.

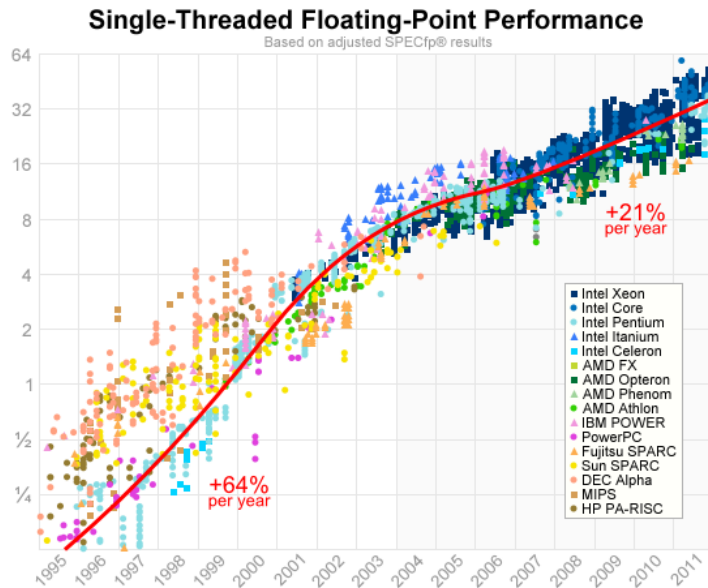


# CONSEQUÊNCIAS DA LEI DE MOORE...

**Treinamos pessoas** dizendo que o desempenho viria do hardware.

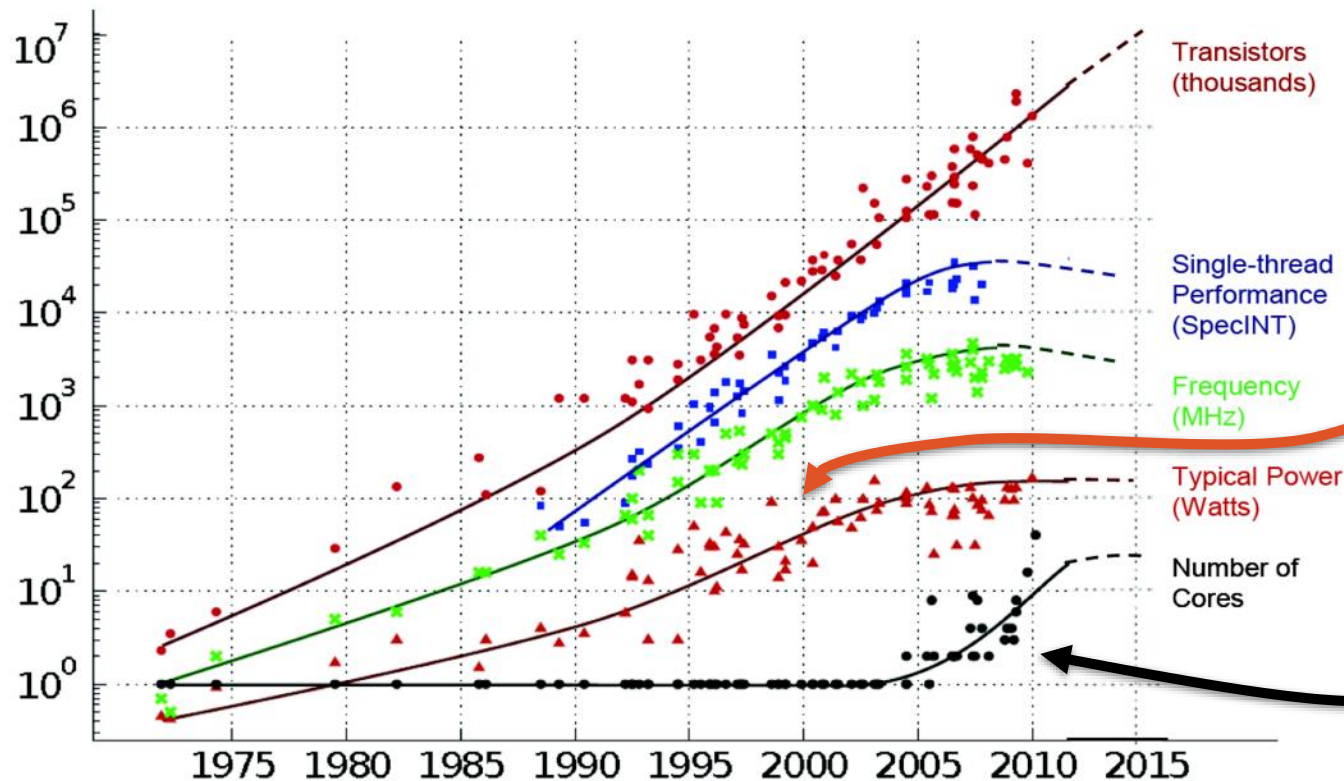
Escreva seu programa como desejar e os Gênios-do-Hardware vamos cuidar do desempenho de forma “mágica”.

**O resultado:** Gerações de engenheiros de software “ignorantes”, usando linguagens ineficientes em termos de performance (como Java)... o que era OK, já que o desempenho é trabalho do Hardware.



# ... ARQUITETURA DE COMPUTADORES E O POWER WALL

## 35 YEARS OF MICROPROCESSOR TREND DATA



Entre 486 e Pentium4  
 $Power = Perf^{1.74}$

Núcleos mais simples  
 $< Freq$

# ... O RESULTADO



Um novo contrato (**the free lunch is over**):

...os **especialistas em HW** irão fazer o que é natural para eles  
(**muitos núcleos de processamento pequenos....**

...os **especialistas de SW** vão ter que se adaptar (**reescrever tudo**)

# ... O RESULTADO



Um novo contrato (**the free lunch is over**):

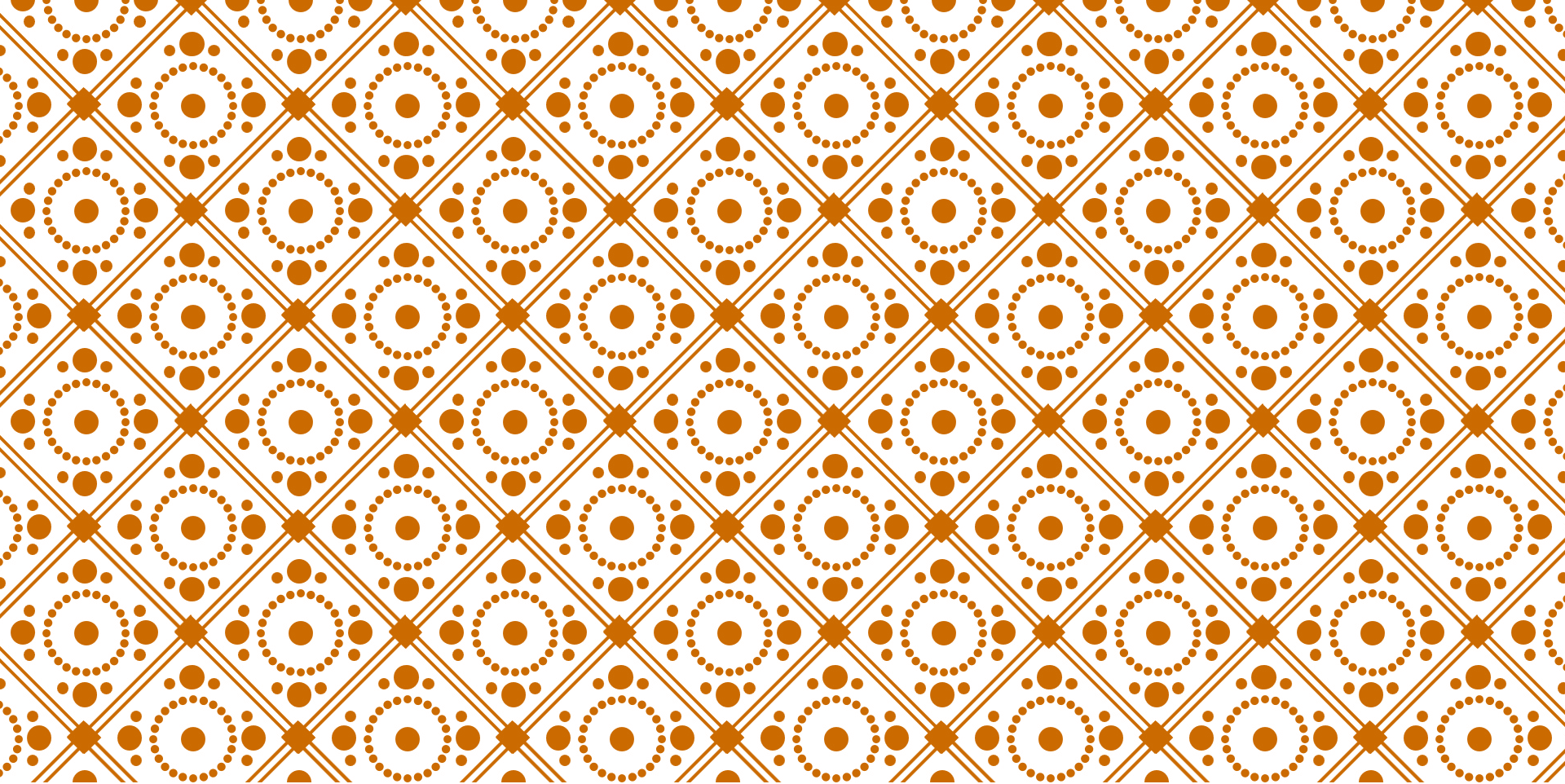
...os **especialistas em HW** irão fazer o que é natural para eles  
(**muitos núcleos de processamento pequenos....**

...os **especialistas de SW** vão ter que se adaptar (**reescrever tudo**)

**O problema** é que isso foi apresentado como um ultimato... ninguém perguntou se estaríamos OK com esse novo contrato...

...o que foi bastante rude.

**Só nos resta colocar a mão na massa e usar  
programação paralela!**

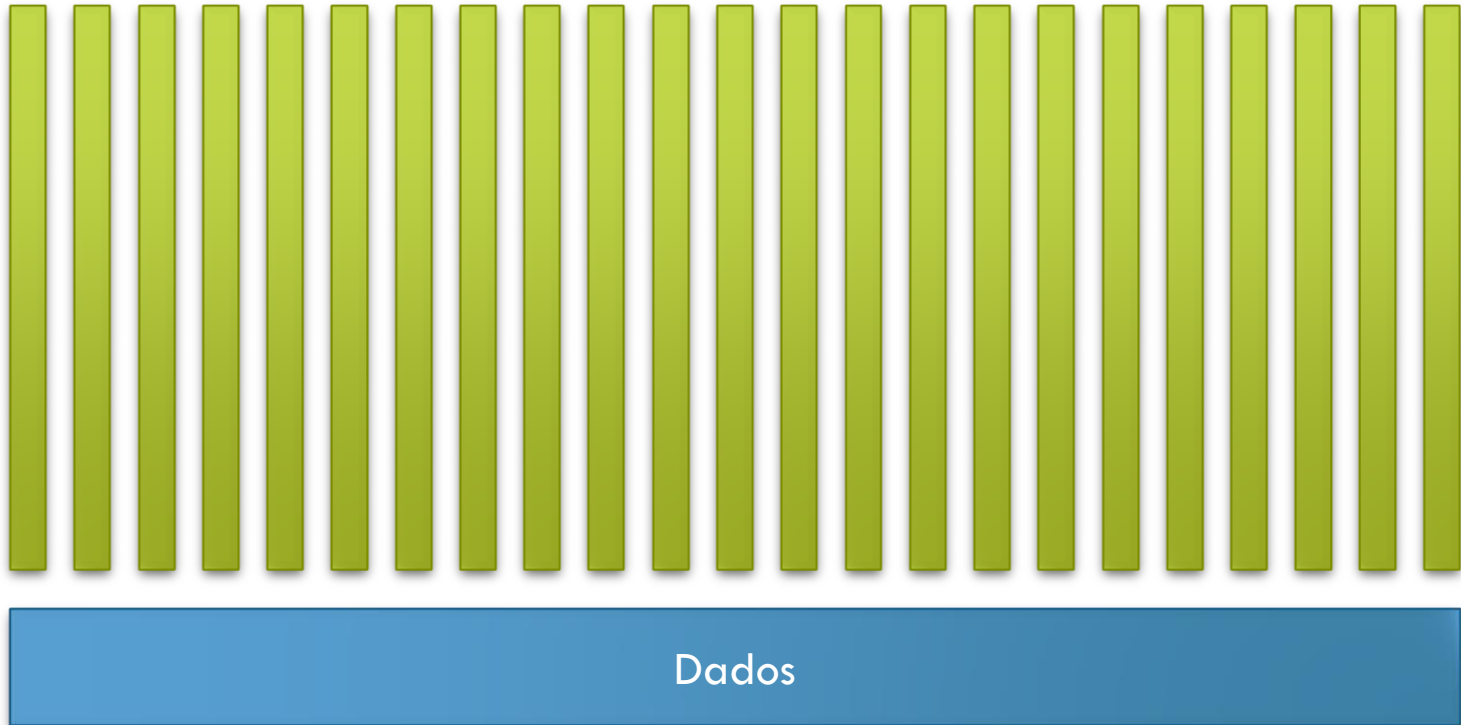


# DEFINIÇÕES BÁSICAS

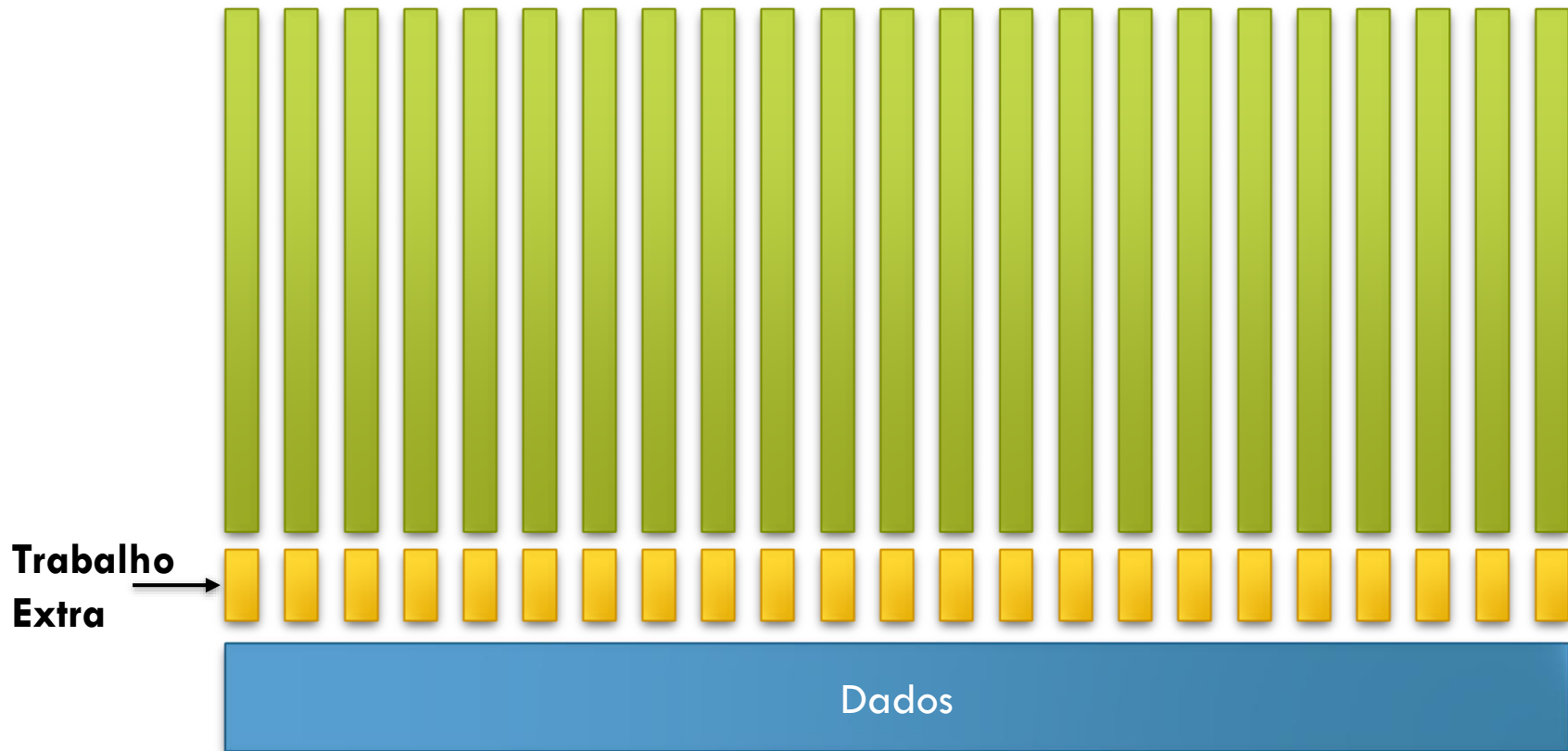
# COMO IREMOS PARALELIZAR? **PENSANDO!**



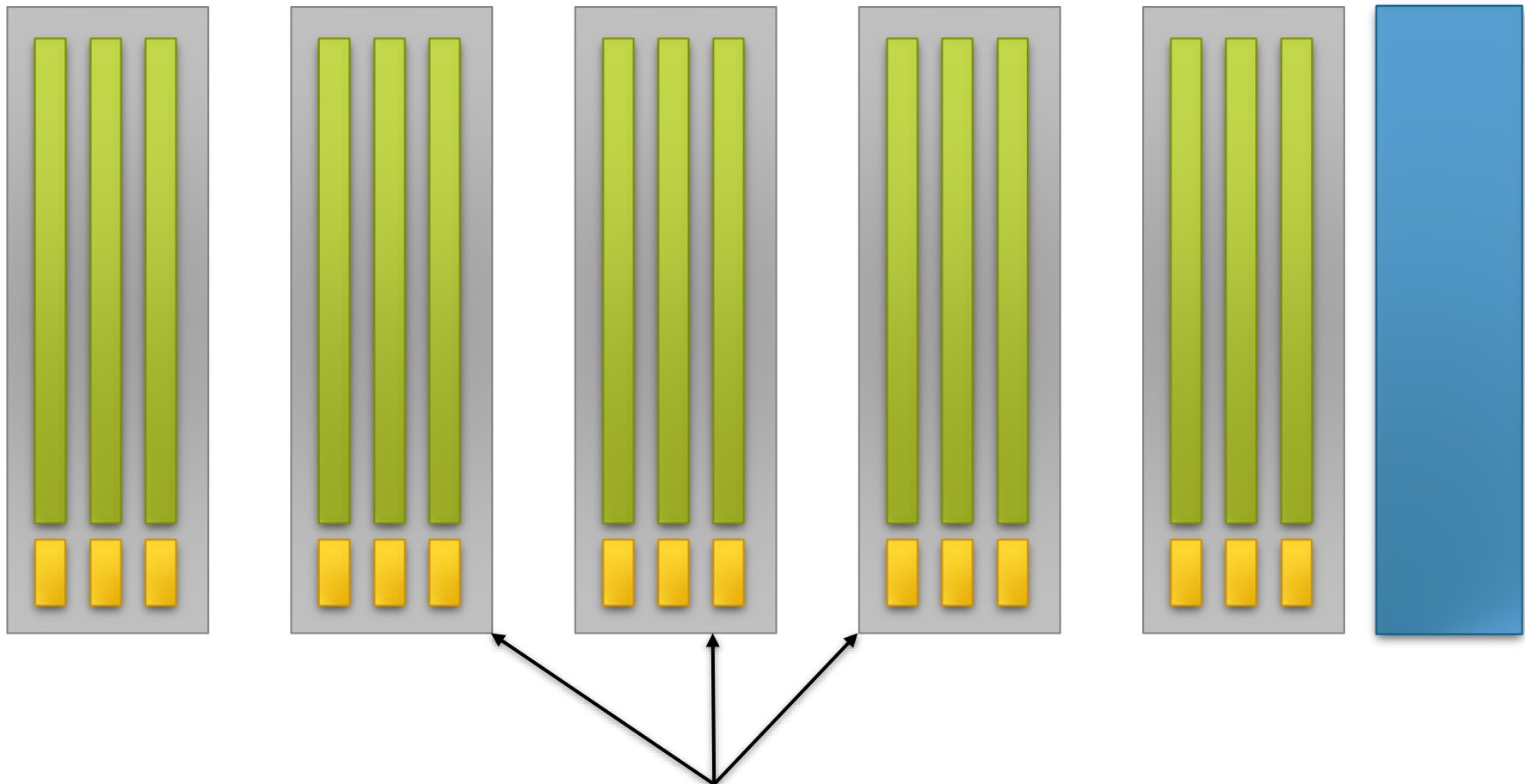
# COMO IREMOS PARALELIZAR? PENSANDO!



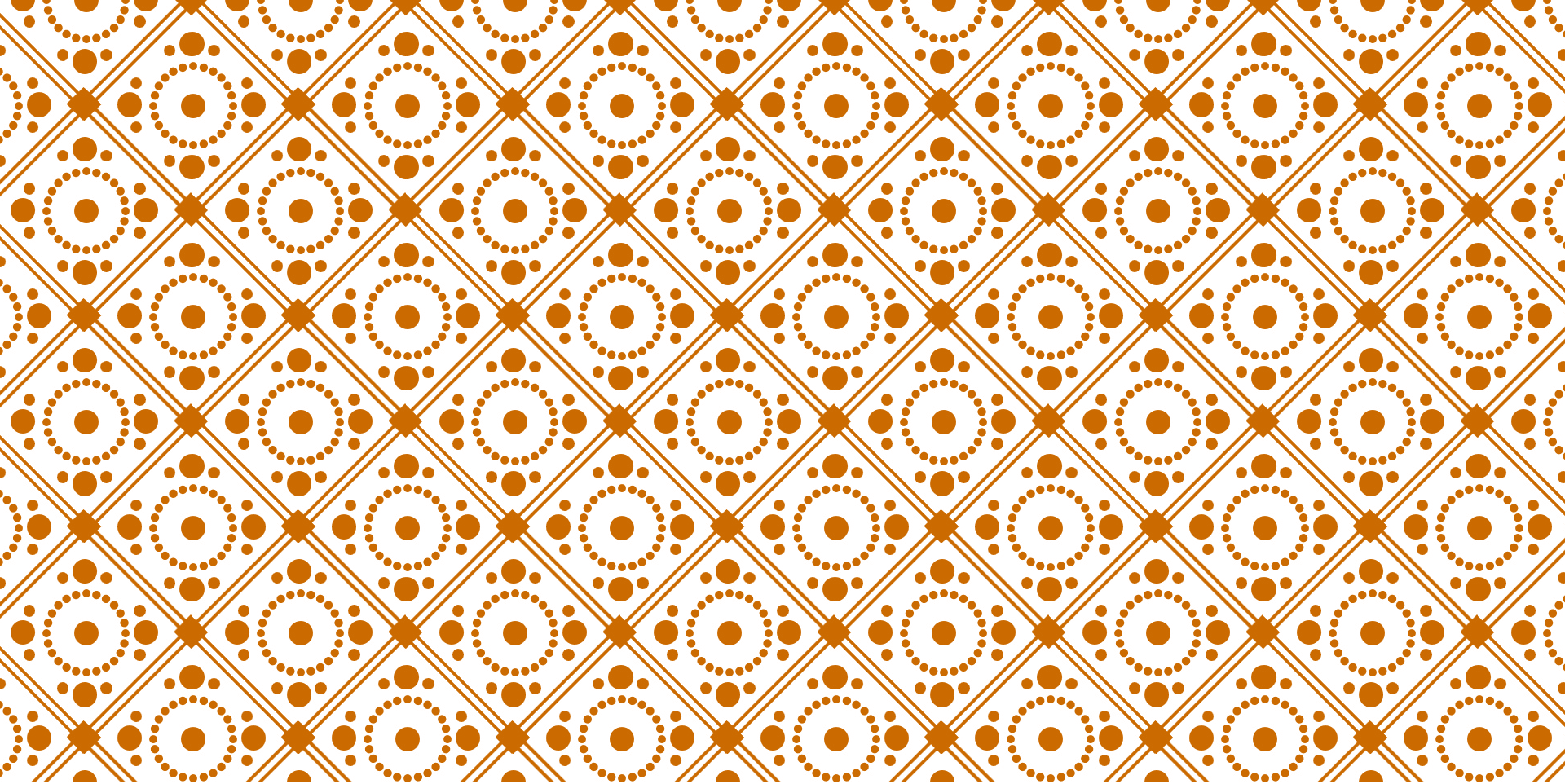
# COMO IREMOS PARALELIZAR? PENSANDO!



# COMO IREMOS PARALELIZAR? PENSANDO!



**Divisão e Organização lógica  
do nosso algoritmo paralelo**



# OPENMP

# VISÃO GERAL OPENMP:

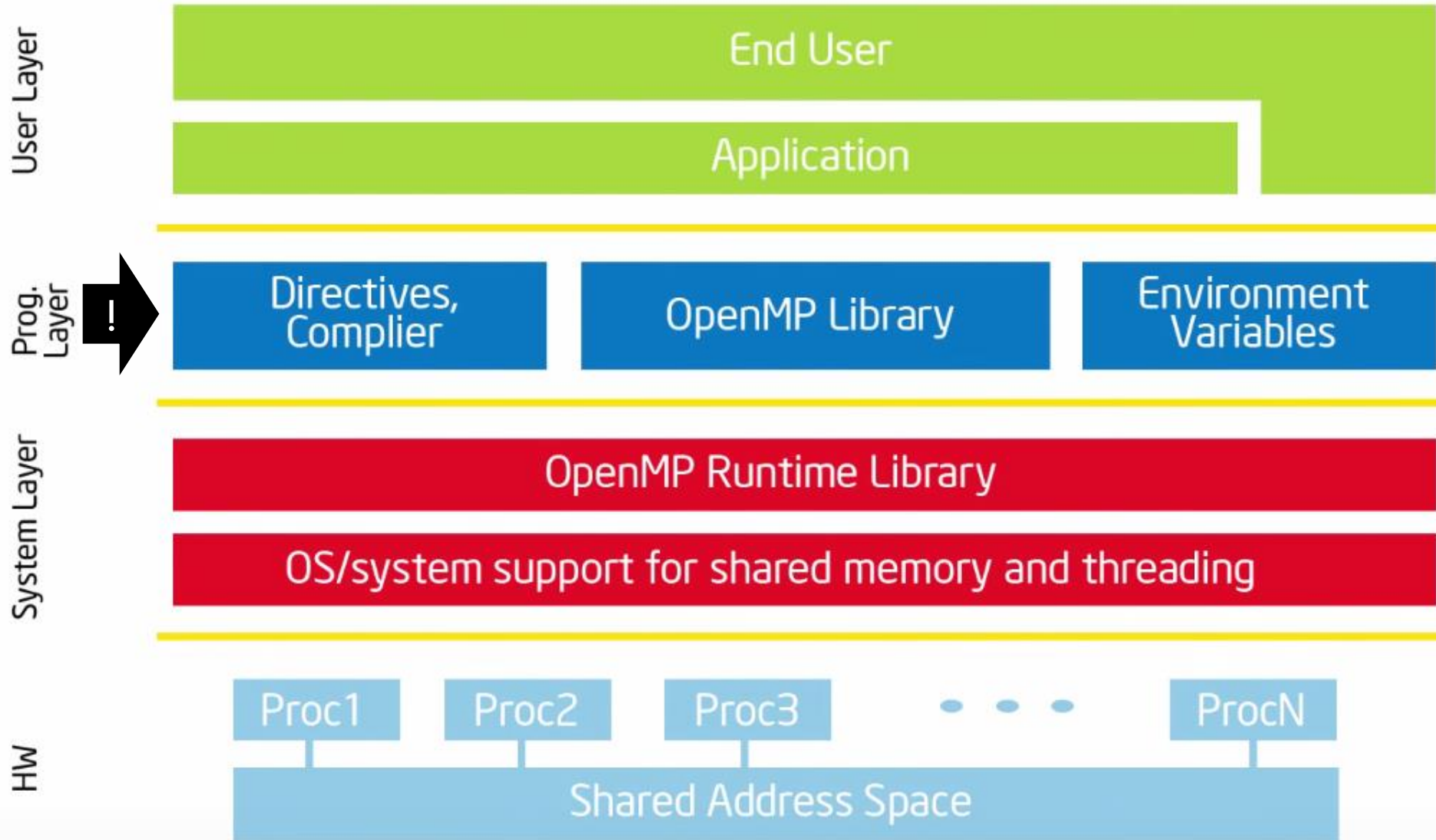
OpenMP: Uma API para escrever aplicações Multithreaded

Um conjunto de diretivas do compilador e biblioteca de rotinas para programadores de aplicações paralelas

Simplifica muito a escrita de programas multi-threaded (MT)

Padroniza 20 anos de prática SMP

# OPENMP DEFINIÇÕES BÁSICAS: PILHA SW



# 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]...]
```

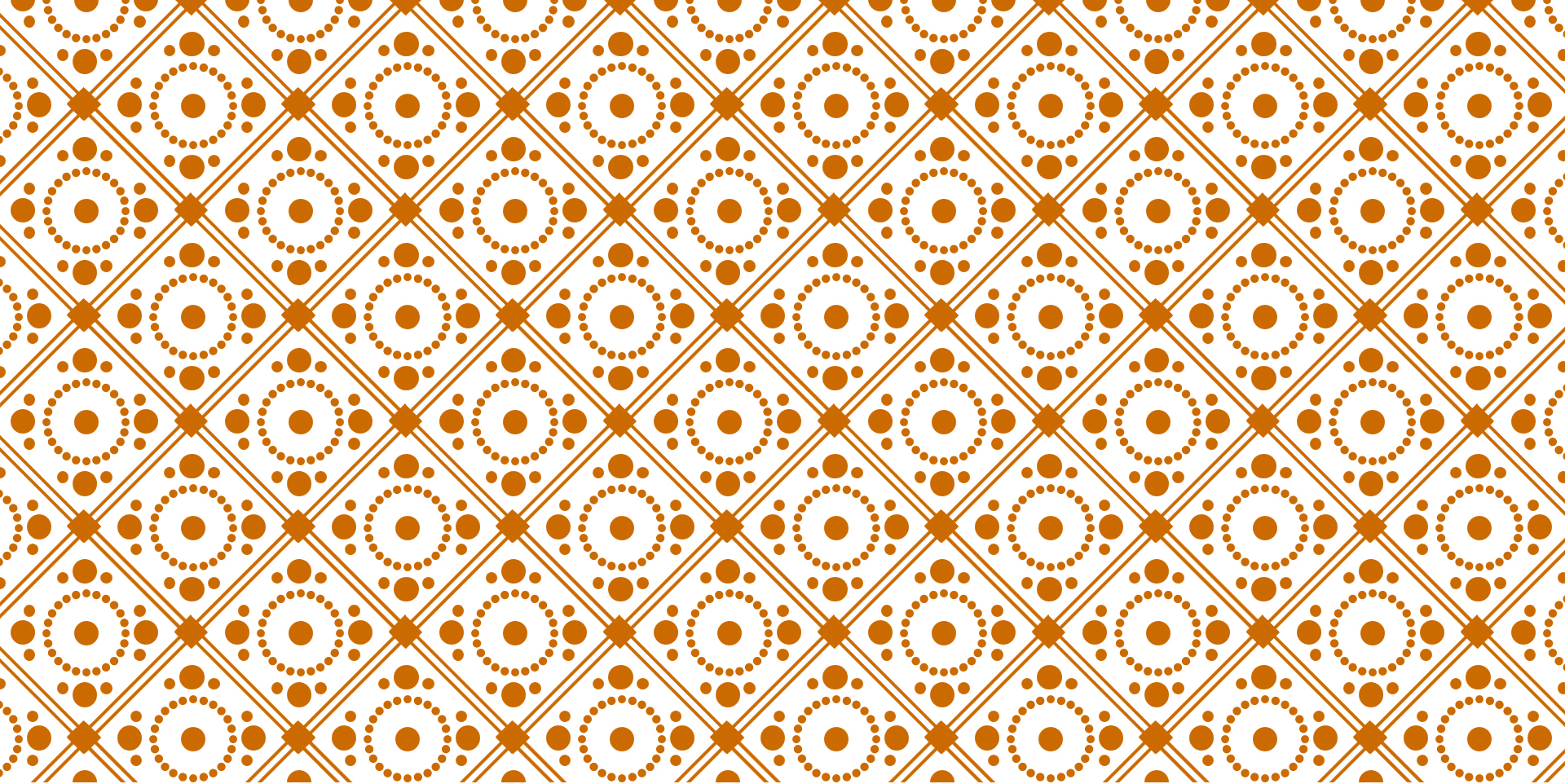
- Exemplo:

```
#pragma omp parallel num_threads(4)
```

A maioria das construções se aplicam a um “**bloco estruturado**” (basic block).

**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.



# ALGUNS BITS CHATOS... USANDO O COMPILADOR OPENMP (HELLO WORLD)

# NOTAS DE COMPILAÇÃO

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

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

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

```
export OMP_NUM_THREADS=4
```

```
./a.out
```

Para shell Bash

Também  
funciona no  
Windows!

Até mesmo  
no  
VisualStudio!

**Mas vamos  
usar Linux!**

# EXERCÍCIO 1, PARTE A: HELLO WORLD

Verifique se seu ambiente funciona

**Escreva um programa** que escreva “hello world”.

```
#include <stdio.h>

int main()
{
    int ID = 0;

    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);

}
```

gcc  
ou  
icc

# EXERCÍCIO 1, PARTE B: HELLO WORLD

Verifique se seu ambiente funciona

**Escreva um programa multithreaded** que escreva “hello world”.

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

```
int main() {
    int ID = 0;
```

```
    #pragma omp parallel
    {
```

```
        printf(" hello(%d) ", ID);
        printf(" world(%d) \n", ID);
```

```
    }
```

```
}
```

```
gcc -fopenmp
ou
Icc -fopenmp
```

```
// Queremos escrever o
// id de cada thread
```

# EXERCÍCIO 1, PARTE C: HELLO WORLD

Verifique se seu ambiente funciona

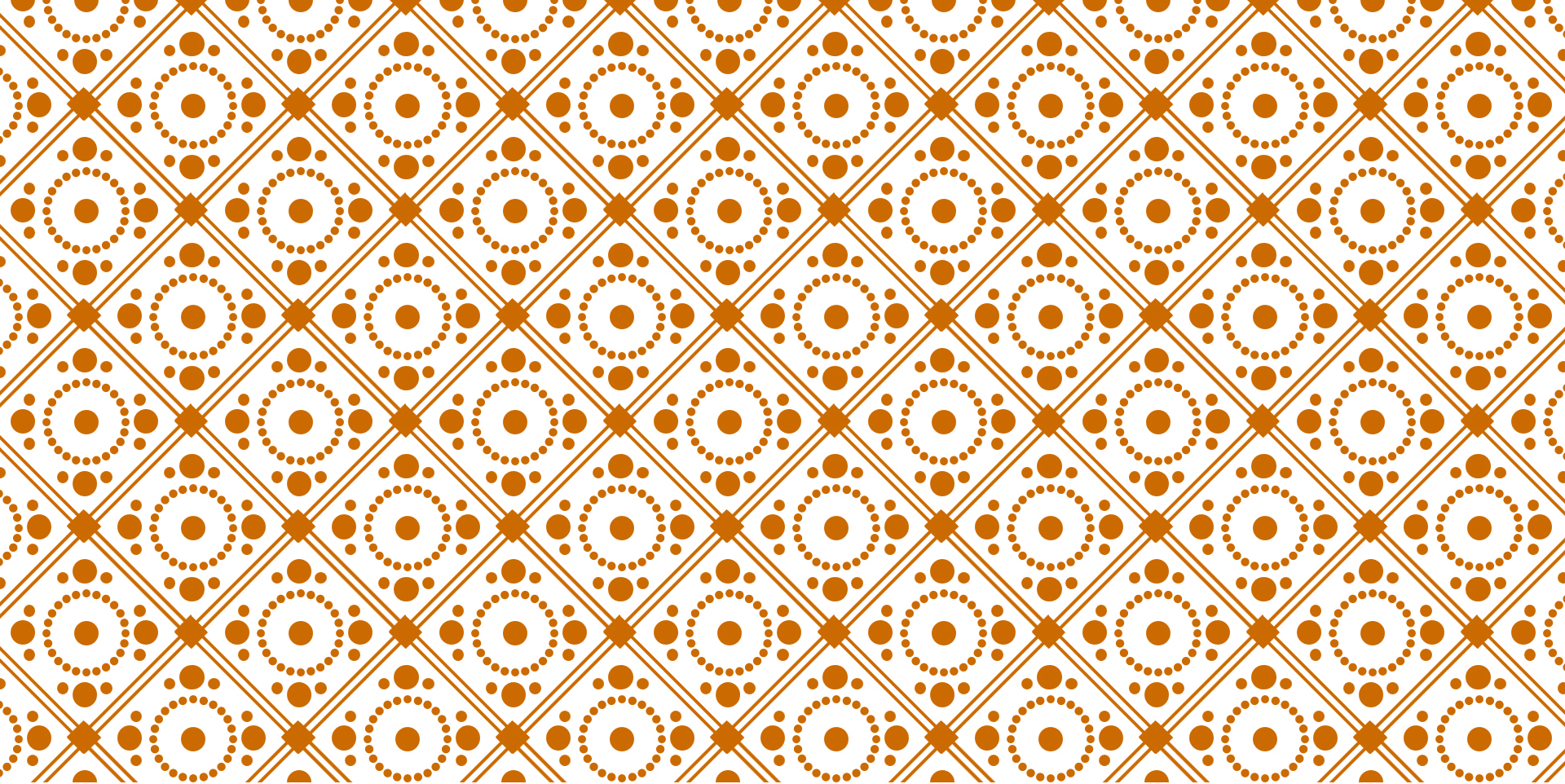
Vamos **adicionar o número da thread** ao “hello world”.

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

int main() {

    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf(" hello(%d) ", ID);
        printf(" world(%d) \n", ID);
    }
}
```

```
gcc -fopenmp
ou
Icc -fopenmp
```



# HELLO WORLD E COMO AS THREADS FUNCIONAM

# EXERCÍCIO 1: SOLUÇÃO

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

Arquivo OpenMP

```
int main() {
```

```
    #pragma omp parallel
    {
```

Região paralela com um número padrão de threads

```
        int ID = omp_get_thread_num();
        printf(" hello(%d) ", ID);
        printf(" world(%d) \n", ID);
```

Função da biblioteca que retorna o thread ID.

```
    }
```

Fim da região paralela

```
}
```

# EXERCÍCIO 1: SOLUÇÃO

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

int main() {

    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf(" hello(%d) \n", ID);
        printf(" world(%d) \n", ID);
    }
}
```

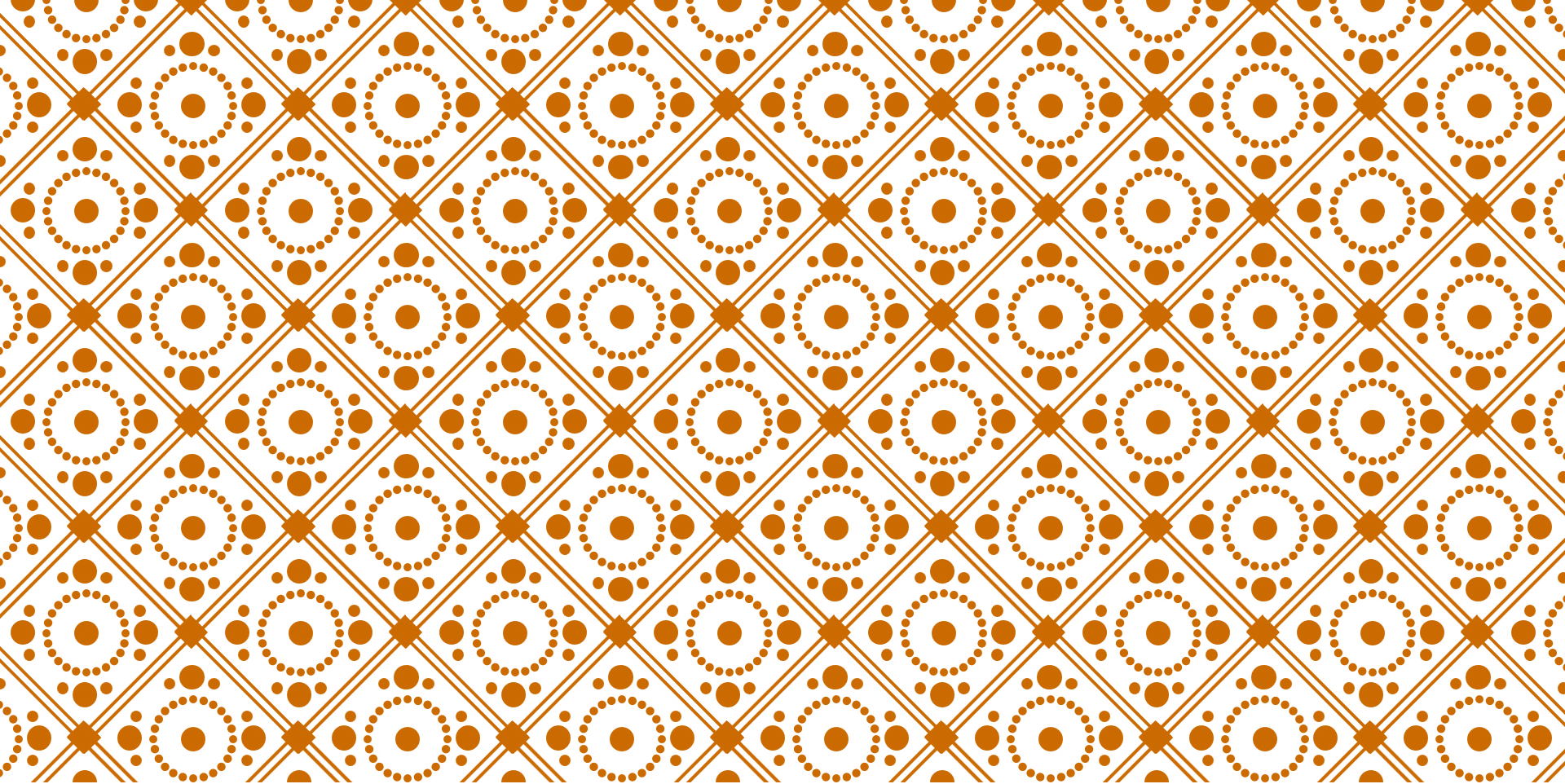
Arquivo OpenMP

Região paralela com um  
número padrão de threads

Função da biblioteca que  
retorna o thread ID.

Fim da região paralela

Como foi a saída  
do programa?  
Organizada?



# FUNCIONAMENTO DOS PROCESSOS VS. THREADS

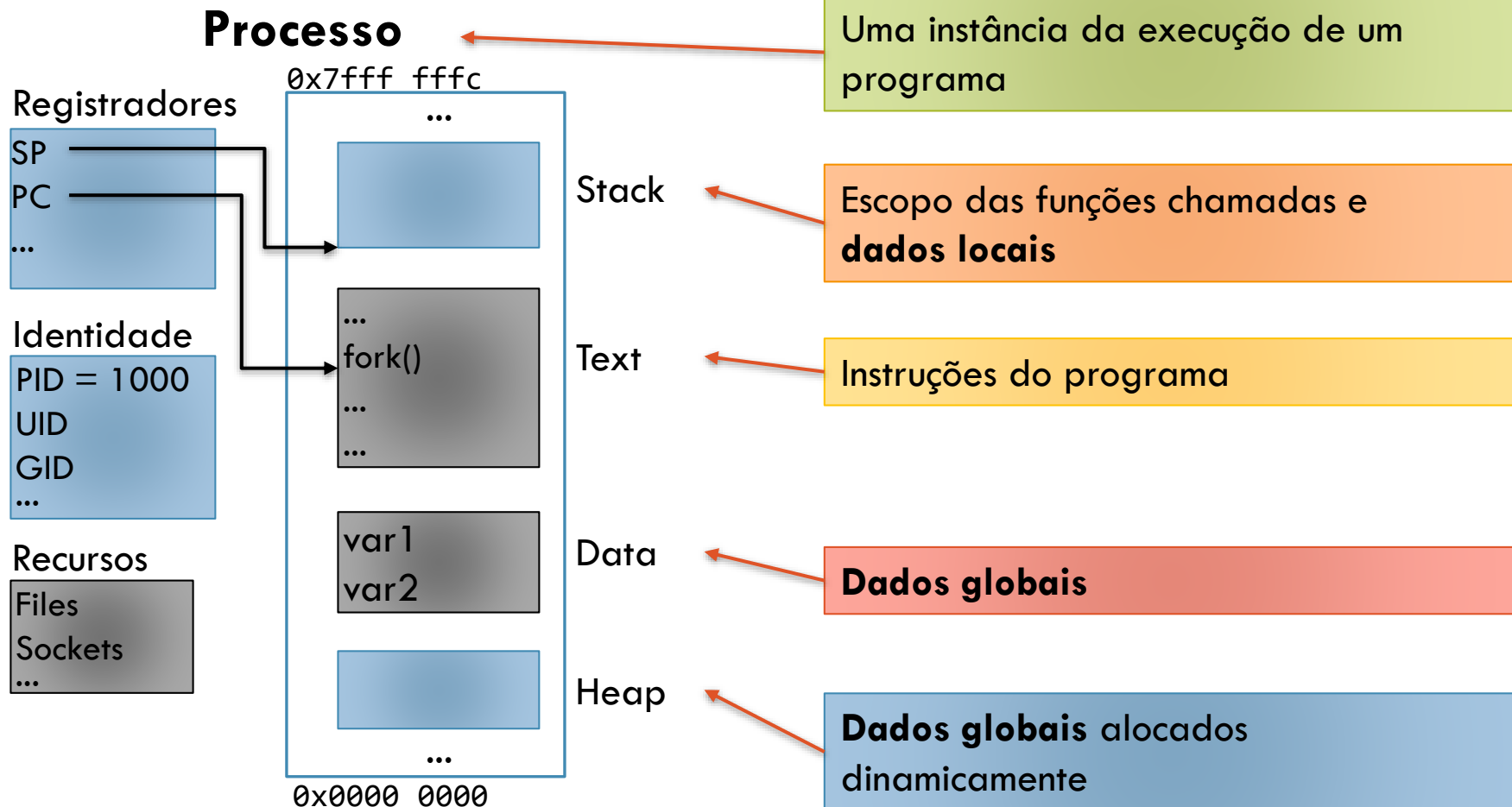
# MULTIPROCESSADORES MEMÓRIA COMPARTILHADA

Um multiprocessadores é um computador em que todos os processadores partilham o acesso à memória física.

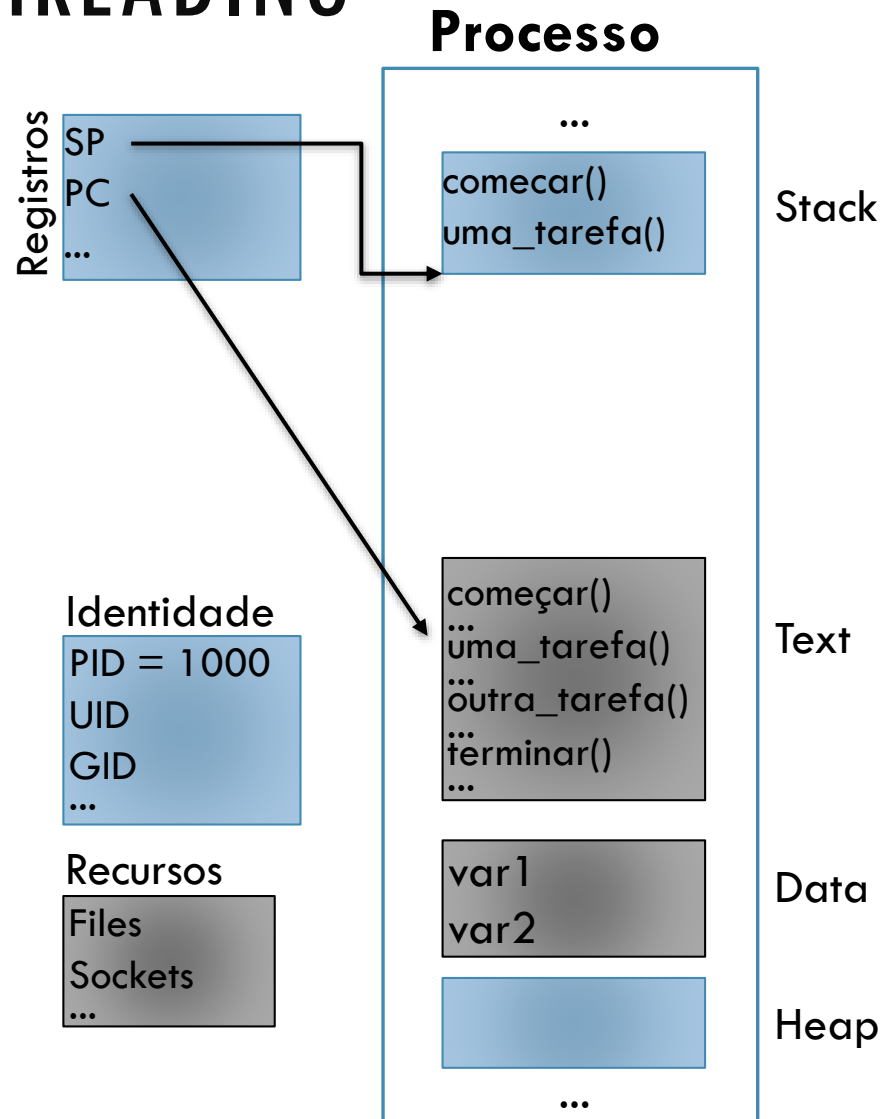
Os processadores executam de forma independente mas o espaço de endereçamento global é partilhado.

Qualquer alteração sobre uma posição de memória realizada por um determinado processador é igualmente visível por todos os restantes processadores.

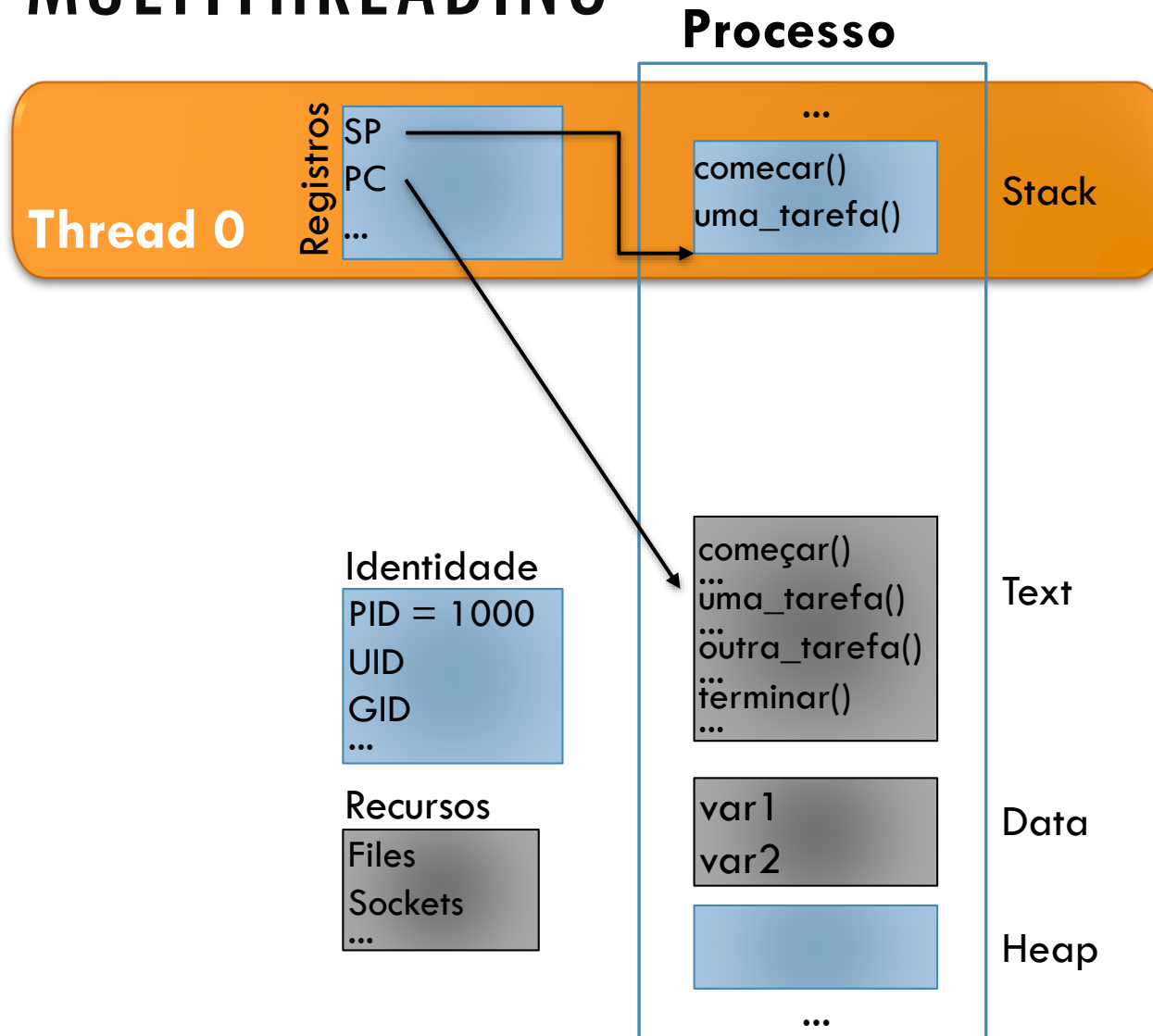
# PROCESSOS



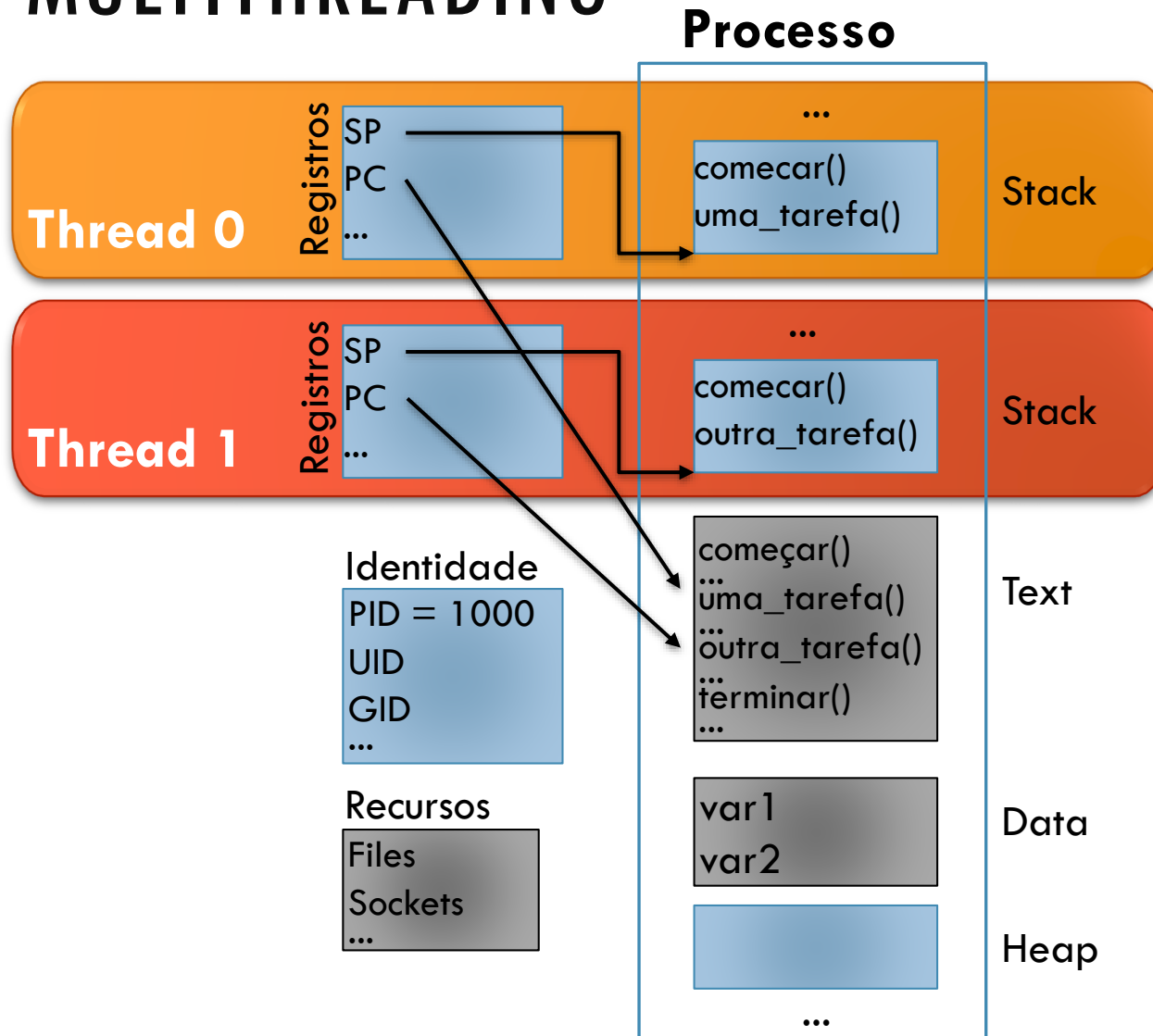
# MULTITHREADING



# MULTITHREADING



# MULTITHREADING



- Cada thread mantém suas chamadas de funções (stack) e suas variáveis locais
- Espaço de endereçamento único
- Variáveis globais/dinâmicas podem ser acessadas por qualquer thread
- Troca de contexto rápida (dados compartilhados)

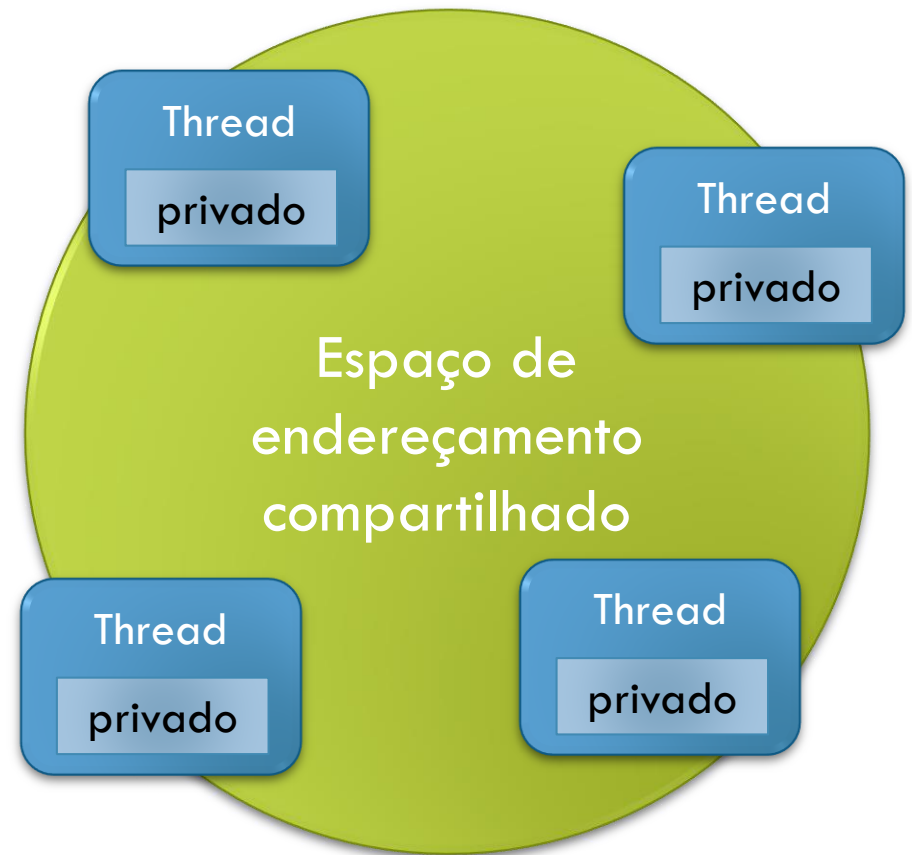
# UM PROGRAMA DE MEMÓRIA COMPARTILHADA

Uma instância do programa:

Um processo e muitas threads.

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

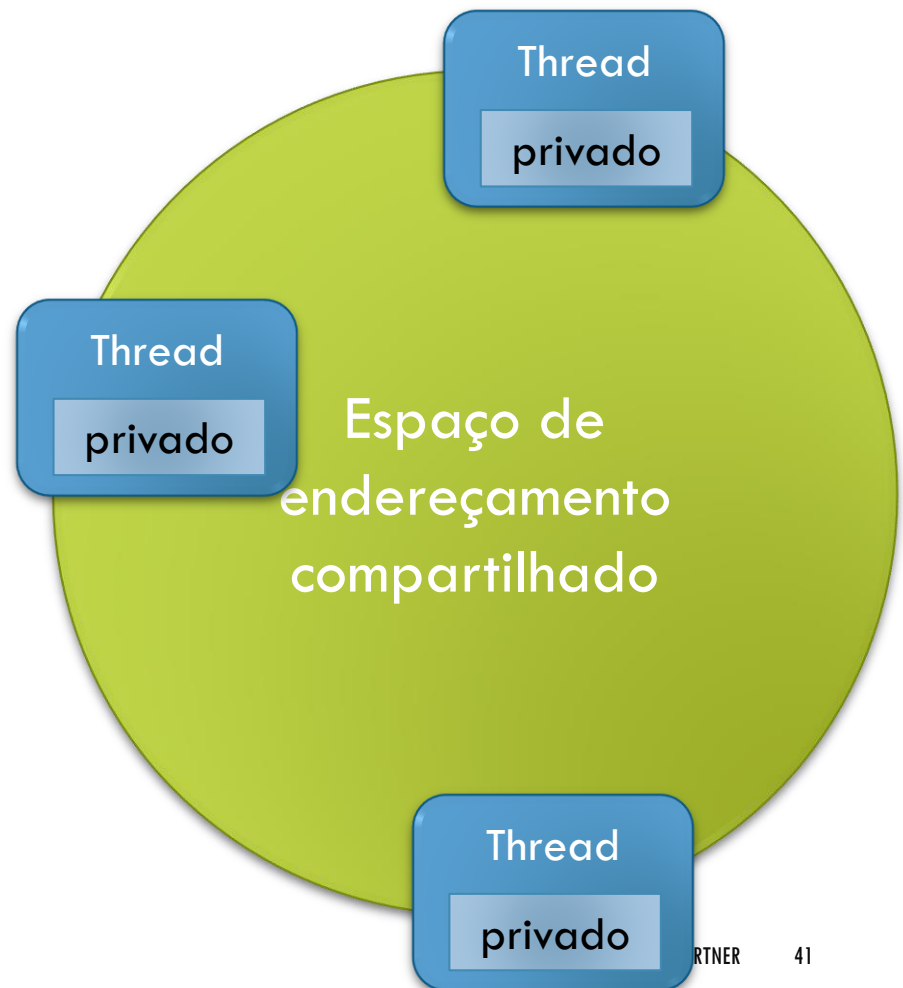
Escalonador SO decide quando executar cada thread (entrelaçado para ser justo).



# UM PROGRAMA DE MEMÓRIA COMPARTILHADA

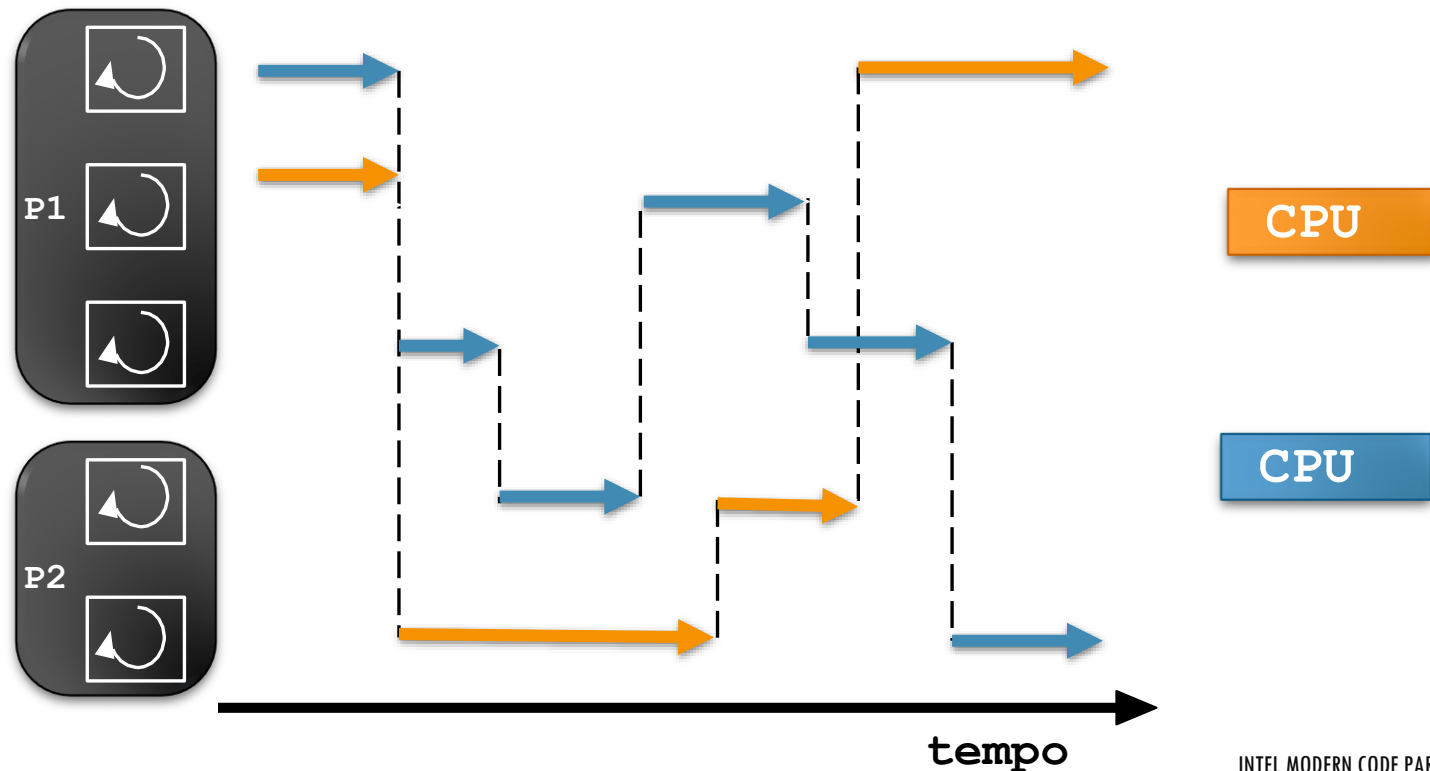
Threads podem ser criadas e destruídas ao longo da execução do programa.

Sincronização garante a ordem correta dos resultados.



# EXECUÇÃO DE PROCESSOS MULTITHREADED

Todos os threads de um processo podem ser executados concorrentemente e em diferentes processadores, caso existam.



# EXERCÍCIO 1: SOLUÇÃO

Agora é possível  
entender esse  
comportamento!



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

int main() {

    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf(" hello(%d) ", ID);
        printf(" world(%d) \n", ID);
    }
}
```

Sample Output:

```
hello(1) hello(0) world(1)
world(0)
hello (3) hello(2) world(3)
world(2)
```

# VISÃO GERAL DE OPENMP: 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 a threads são escalonadas de forma diferente.

Apesar de este ser um aspectos mais poderosos 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:

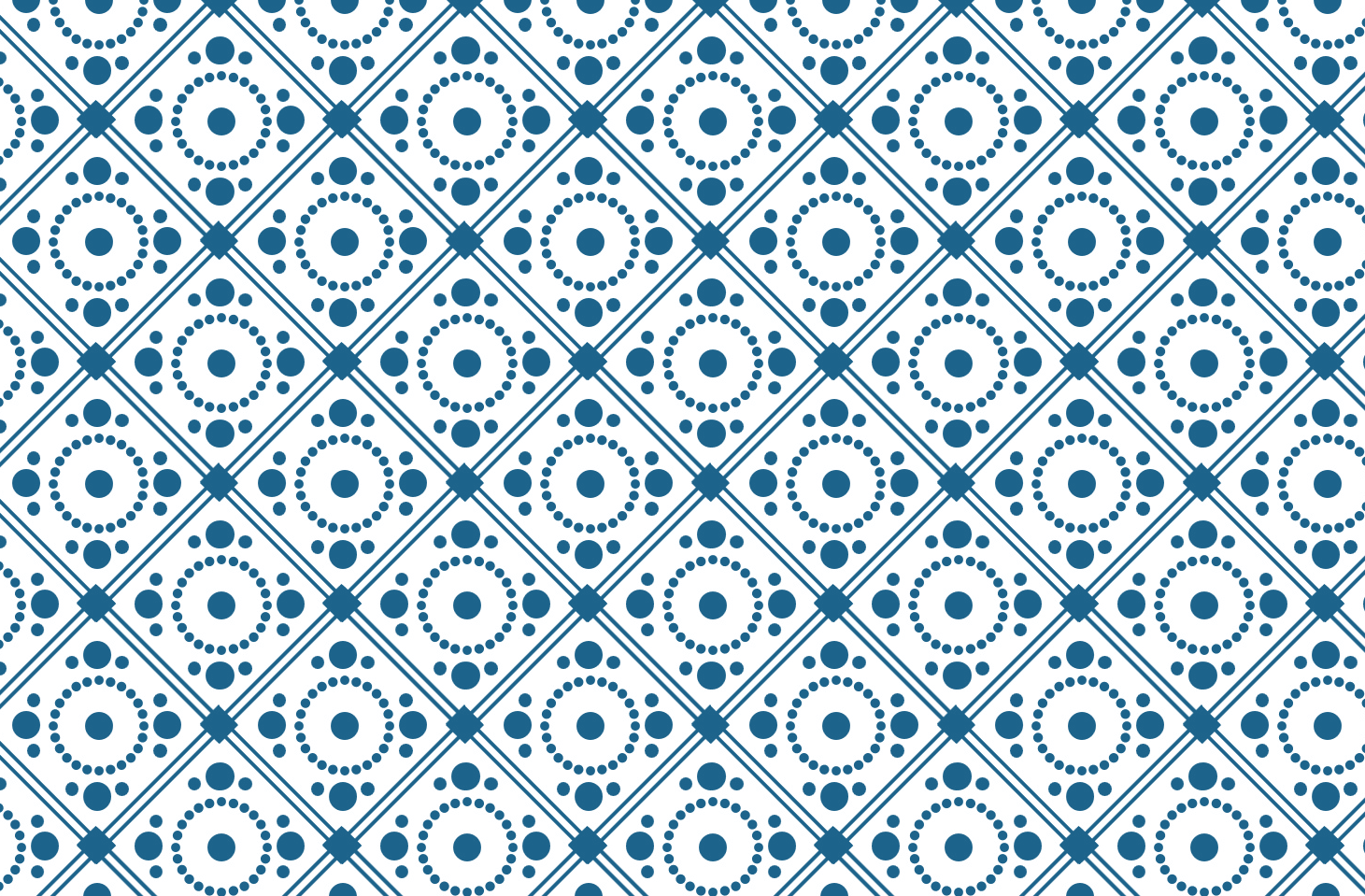
- Tentaremos mudar a forma de acesso aos dados para minimizar a necessidade de sincronizações.

# SINCRONIZAÇÃO E REGIÕES CRÍTICAS: EXEMPLO

| Tempo | Th1                    | Th2                    | Saldo |
|-------|------------------------|------------------------|-------|
| T0    |                        |                        | \$200 |
| T1    | Leia Saldo<br>\$200    |                        | \$200 |
| T2    |                        | Leia Saldo<br>\$200    | \$200 |
| T3    |                        | Some \$100<br>\$300    | \$200 |
| T4    | Some \$150<br>\$350    |                        | \$200 |
| T5    |                        | Escreva Saldo<br>\$300 | \$300 |
| T6    | Escreva Saldo<br>\$350 |                        | \$350 |

# SINCRONIZAÇÃO E REGIÕES CRÍTICAS: EXEMPLO

| Tempo | Th1                                                                                                                          | Th2                    | Saldo |
|-------|------------------------------------------------------------------------------------------------------------------------------|------------------------|-------|
| T0    | <p>Devemos garantir que <b>não importa a ordem de execução (escalonamento)</b>, teremos sempre um resultado consistente!</p> |                        |       |
| T1    |                                                                                                                              |                        |       |
| T2    |                                                                                                                              |                        |       |
| T3    |                                                                                                                              |                        |       |
| T4    |                                                                                                                              |                        |       |
| T5    |                                                                                                                              | Escreva Saldo<br>\$300 | \$300 |
| T6    | Escreva Saldo<br>\$350                                                                                                       |                        | \$350 |



# INTEL MODERN CODE PARTNER

## OPENMP — AULA 02

# AGENDA GERAL

## Unit 1: Getting started with OpenMP

- Mod 1: Introduction to parallel programming
- Mod 2: The boring bits: Using an OpenMP compiler (hello world)
- Disc 1: Hello world and how threads work

## Unit 2: The core features of OpenMP

- Mod 3: Creating Threads (the Pi program)
- Disc 2: The simple Pi program and why it sucks
- Mod 4: Synchronization (Pi program revisited)
- Disc 3: Synchronization overhead and eliminating false sharing
- Mod 5: Parallel Loops (making the Pi program simple)
- Disc 4: Pi program wrap-up

## Unit 3: Working with OpenMP

- Mod 6: Synchronize single masters and stuff
- Mod 7: Data environment
- Disc 5: Debugging OpenMP programs

## Unit 4: a few advanced OpenMP topics

- Mod 8: Skills practice ... linked lists and OpenMP
- Disc 6: Different ways to traverse linked lists
- Mod 8: Tasks (linked lists the easy way)
- Disc 7: Understanding Tasks

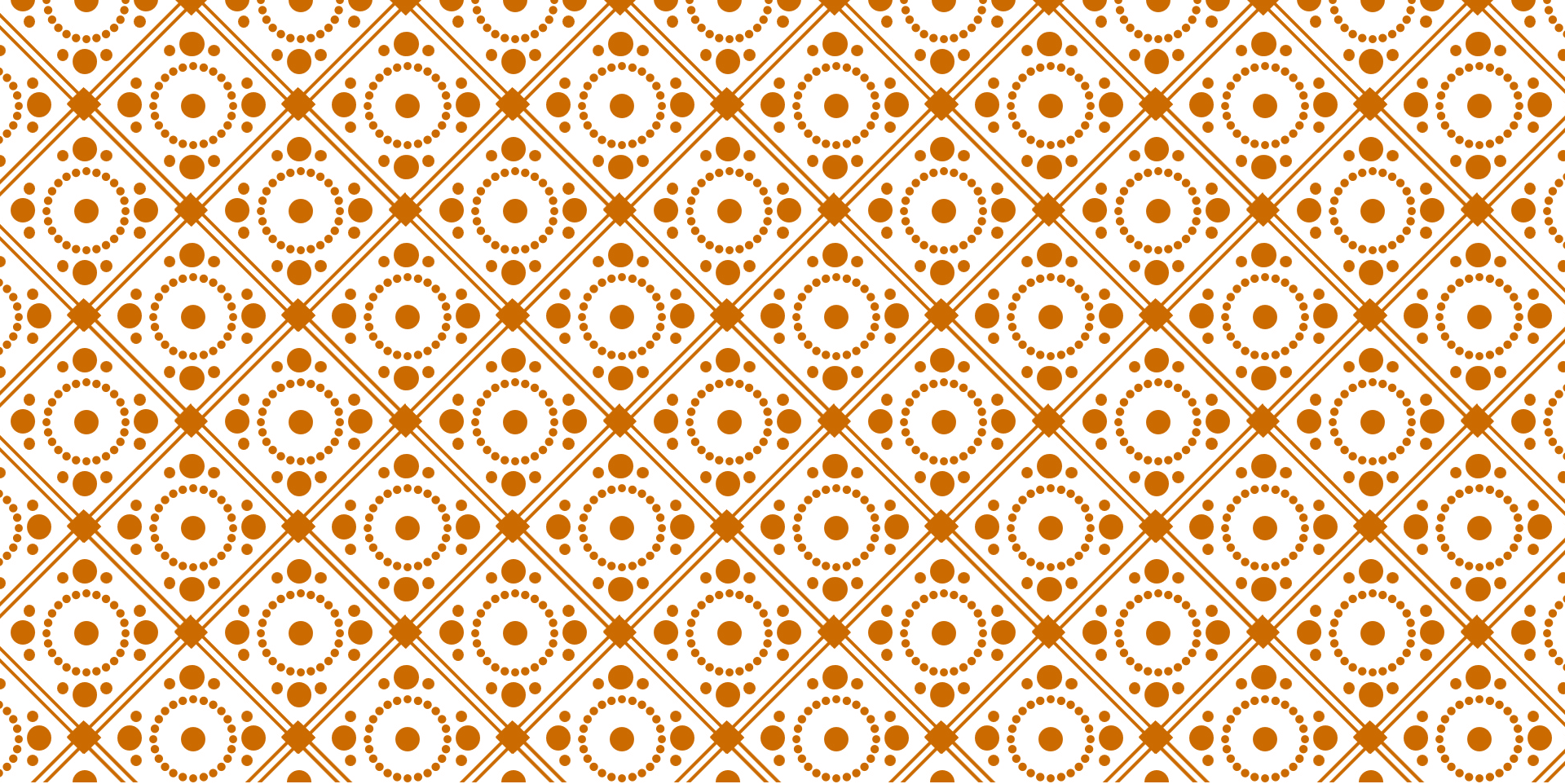
## Unit 5: Recapitulation

- Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
- Disc 8: The pitfalls of pairwise synchronization
- Mod 9: Threadprivate Data and how to support libraries (Pi again)
- Disc 9: Random number generators

# AGENDA DESSA AULA

## Unit 2: The core features of OpenMP

- Mod 3: Creating Threads (the Pi program)
- Disc 2: The simple Pi program and why it sucks
- Mod 4: Synchronization (Pi program revisited)
- Disc 3: Synchronization overhead and eliminating false sharing
- Mod 5: Parallel Loops (making the Pi program simple)
- Disc 4: Pi program wrap-up

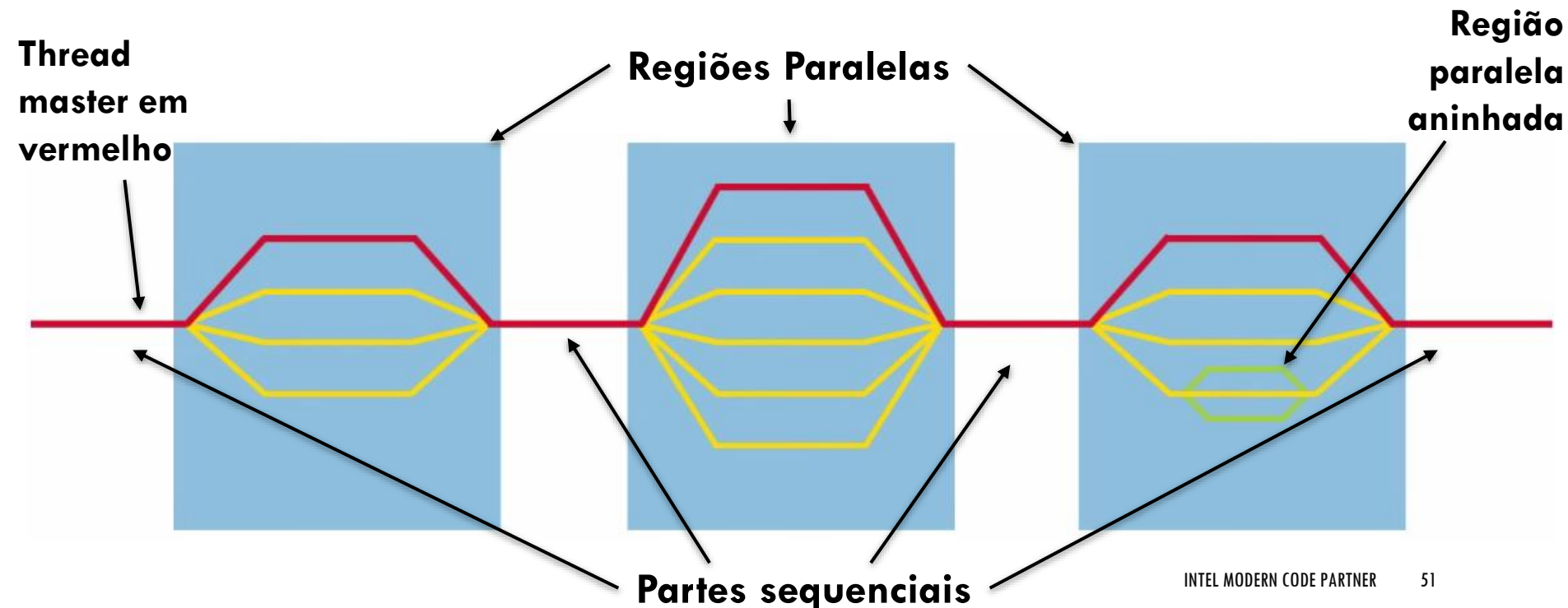


# CRIANDO THREADS (O PROGRAMA PI)

# MODELO DE PROGRAMAÇÃO OPENMP: PARALELISMO FORK-JOIN

A thread Master despeja um time de threads como necessário

Paralelismo é adicionado aos poucos até que o objetivo de desempenho é alcançado. Ou seja o programa sequencial evolui para um programa paralelo



# criação de threads: regiões paralelas

Criamos threads em OpenMP com construções **parallel**.

Por exemplo, para criar uma região paralela com 4 threads:

Cada thread chama `pooh(ID,A)` para os IDs = 0 até 3

```
double A[1000];  
omp_set_num_threads(4);  
  
#pragma omp parallel  
{  
    int ID = omp_get_thread_num();  
    pooh(ID,A);  
}
```

Função para requerer uma certa quantidade de threads

Cria threads em OpenMP

Função que retorna o ID de cada thread

Cada thread executa uma cópia do código dentro do bloco estruturado

# criação de threads: regiões paralelas

Criamos threads em OpenMP com construções **parallel**.

Por exemplo, para criar uma região paralela com 4 threads:

Cada thread chama `pooh(ID,A)` para os IDs = 0 até 3

```
double A[1000];
```

Um cópia única de A é **compartilhada** entre todas as threads

```
#pragma omp parallel num_threads(4)
```

Forma reduzida

```
{  
  int ID = omp_get_thread_num();  
  pooh(ID,A);  
}
```

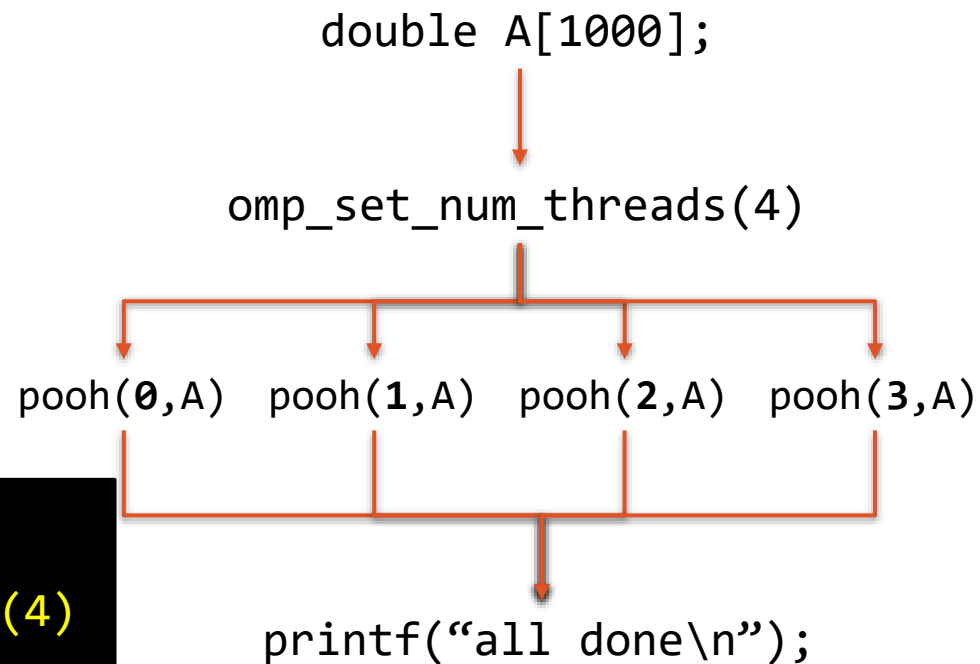
O inteiro ID é **privada** para cada thread

# criação de threads: regiões paralelas

Cada thread executa o mesmo código de forma redundante.

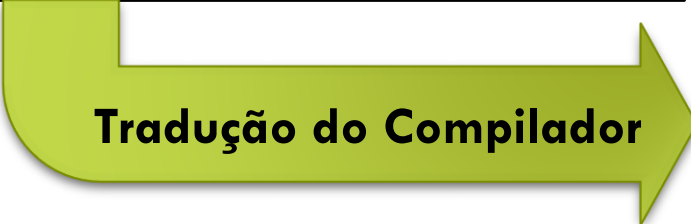
As threads esperam para que todas as demais terminem antes de prosseguir (i.e. uma barreira)

```
double A[1000];
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



# OPENMP: O QUE O COMPILADOR FAZ...

```
#pragma omp parallel num_threads(4)
{
    foobar ();
}
```



**Tradução do Compilador**

Todas as implementações de OpenMP conhecidas usam um pool de threads para que o custo de criação e destruição não ocorram para cada região paralela.

Apenas três threads serão criadas porque a última seção será invocada pela thread pai.

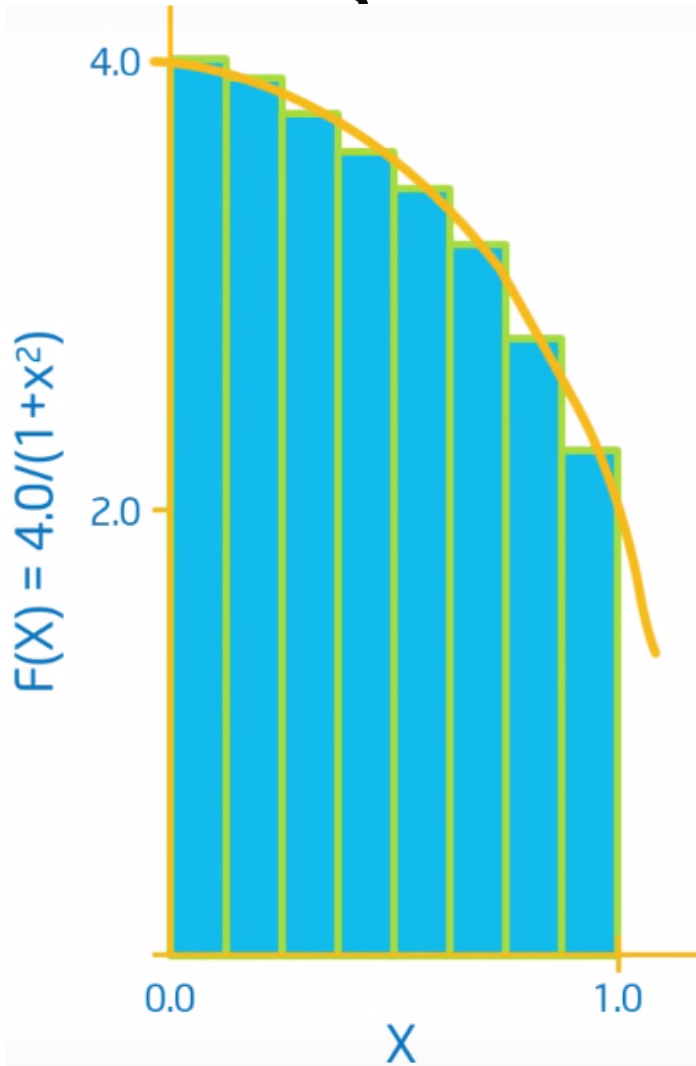
```
void thunk () {
    foobar ();
}
```

// Implementação Pthread

```
pthread_t tid[4];
for (int i = 1; i < 4; ++i)
    pthread_create(&tid[i], 0, thunk, 0);

thunk();
for (int i = 1; i < 4; ++i)
    pthread_join(tid[i]);
```

# EXERCÍCIOS 2 A 4: INTEGRAÇÃO NUMÉRICA



Matematicamente, sabemos que:

$$\int_0^1 \frac{4.0}{1+x^2} dx = \pi$$

Podemos aproximar essa integral como a soma de retângulos:

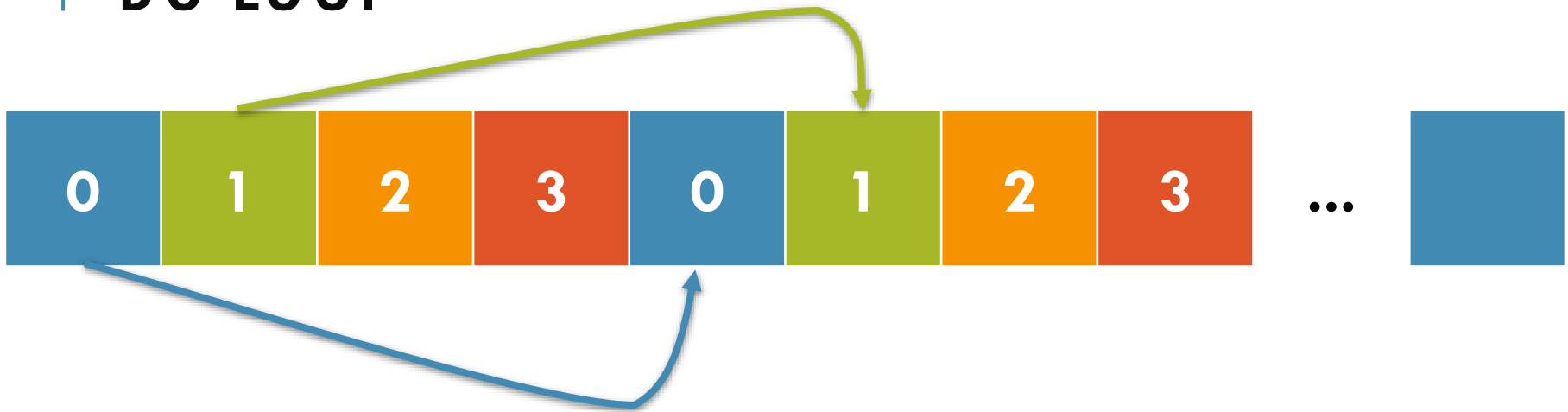
$$\sum_{i=0}^n F(x_i) \Delta x \cong \pi$$

Onde cada retângulo tem largura  $\Delta x$  e altura  $F(x_i)$  no meio do intervalo  $i$ .

## EXERCÍCIOS 2 A 4: PROGRAMA PI SERIAL

```
static long num_steps = 100000;
double step;
int main () {
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0;i< num_steps; i++){
        x = (i + 0.5) * step; // Largura do retângulo
        sum = sum + 4.0 / (1.0 + x*x); // Sum += Área do retângulo
    }
    pi = step * sum;
}
```

# DISTRIBUIÇÃO CÍCLICA DE ITERAÇÕES DO LOOP



```
// Distribuição cíclica  
for(i = id; i < num_steps; i = i + nthreads)
```

# ESTRATÉGIA DO ALGORITMO:

## PADRÃO SPMD (SINGLE PROGRAM MULTIPLE DATA)

Execute o mesmo programa no  $P$  elementos de processamento onde  $P$  pode ser definido bem grande.

Use a identificação ... ID no intervalo de 0 até  $(P-1)$  ... Para selecionar entre um conjunto de threads e gerenciar qualquer estrutura de dados compartilhada.

Esse padrão é genérico e foi usado para suportar a maior parte dos padrões de estratégia de algoritmo (se não todos).

# EXERCÍCIO 2

Crie uma versão paralela do programa pi usando a construção paralela.

Atenção para variáveis globais vs. privadas.

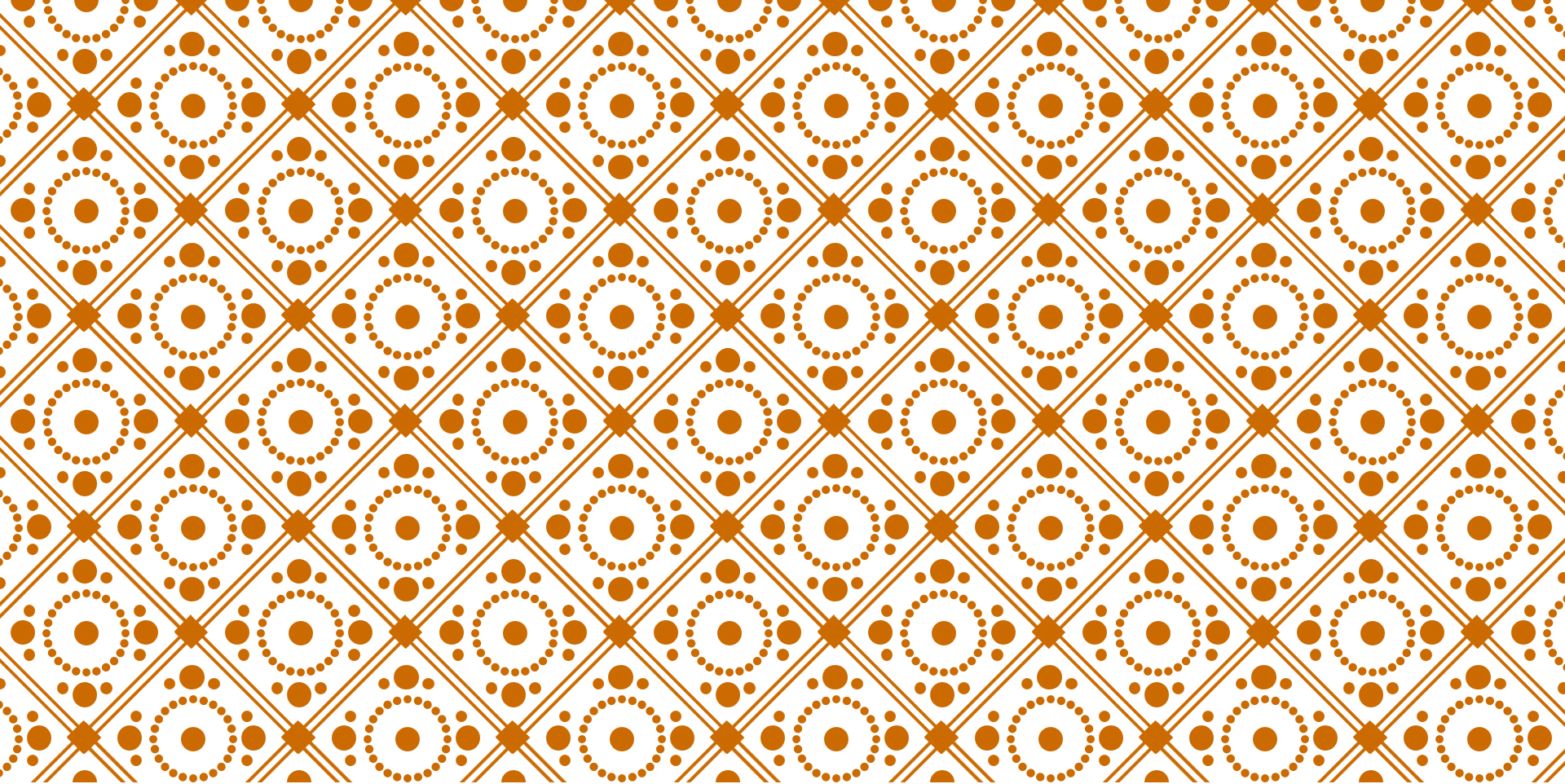
Cuidado com condições de corrida!

Além da construção paralela, iremos precisar da biblioteca de rotinas.

```
#include <omp.h>           // Biblioteca OpenMP

int omp_get_num_threads(); // Número de threads no time

int omp_get_thread_num();  // ID da thread
```



# UM SIMPLES PROGRAMA PI E PORQUE ELE NÃO PRESTA

## EXERCÍCIOS 2 A 4: PROGRAMA PI SERIAL

```
static long num_steps = 100000;
double step;
int main () {
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0;i< num_steps; i++){
        x = (i + 0.5) * step; // Largura do retângulo
        sum = sum + 4.0 / (1.0 + x*x); // Sum += Área do retângulo
    }
    pi = step * sum;
}
```

```

#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main () {
    int i, nthreads; double pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        int i, id, nthrds; double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum[id]=0.0; i<num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0; i<nthreads; i++)
        pi += sum[i] * step;
}

```

Promovemos um escalar para um vetor dimensionado pelo número de threads para prevenir condições de corrida.

Apenas uma thread pode copiar o número de thread para a variável global para certificar que múltiplas threads gravando no mesmo endereço não gerem conflito

Sempre verifique o # de threads

Este é um truque comum em programas SPMD para criar uma distribuição cíclica das iterações do loop

Usamos uma variável global para evitar perder dados

# RESULTADOS\*

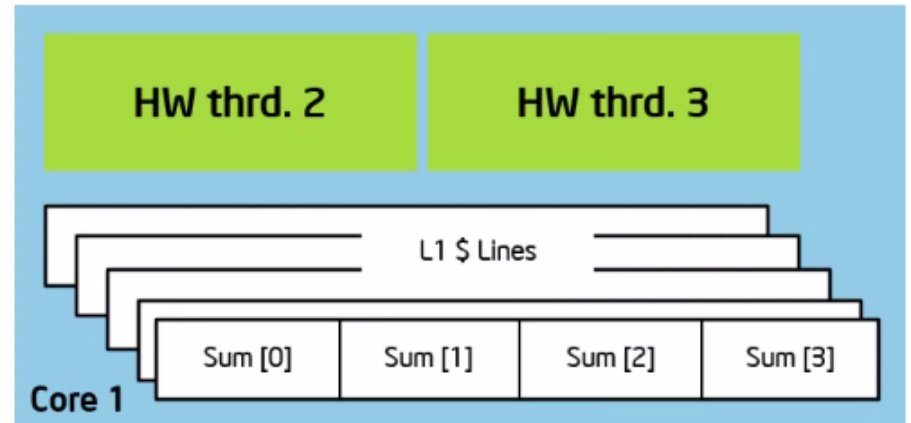
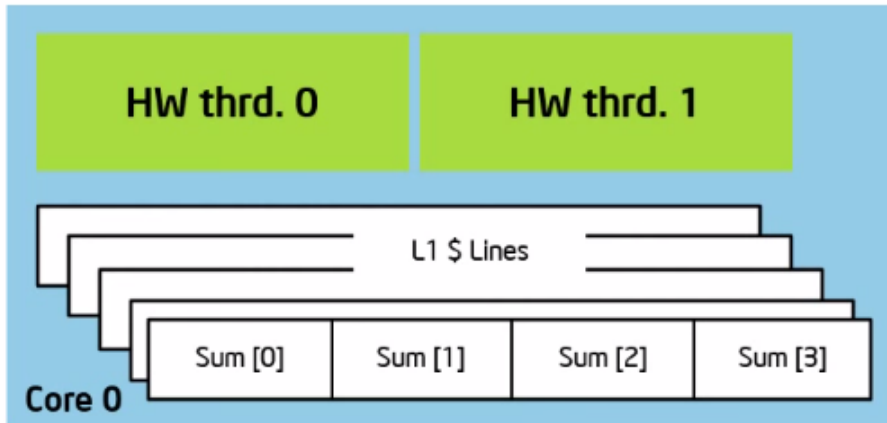
O Pi original sequencial com 100mi passos, executou em **1.83** seg.

\*Compilador Intel (icpc) sem otimizações em um Apple OS X 10.7.3 com dual core (4 HW threads) processador Intel® Core TM i5 1.7Ghz e 4 Gbyte de memória DDR3 1.333 Ghz.

| Threads | 1. SPMD |
|---------|---------|
| 1       | 1.86    |
| 2       | 1.03    |
| 3       | 1.08    |
| 4       | 0.97    |

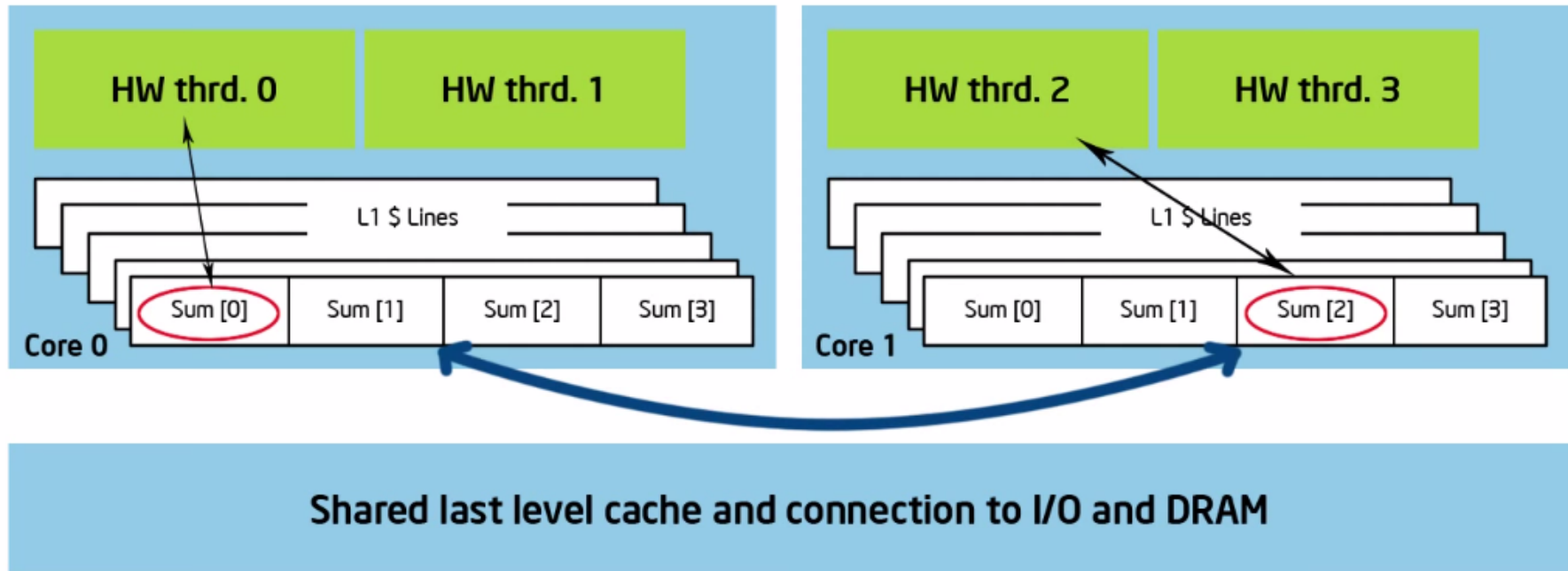
# O MOTIVO DESSA FALTA DE DESEMPENHO? FALSO COMPARTILHAMENTO

Se acontecer de elementos de dados independentes serem alocados em uma mesma linha de cache, cada atualização irá causar que a linha de cache fique em ping-pong entre as threads... Isso é chamado de “falso compartilhamento”.



**Shared last level cache and connection to I/O and DRAM**

# O MOTIVO DESSA FALTA DE DESEMPENHO? FALSO COMPARTILHAMENTO



Se promovermos escalares para vetores para suportar programas SPMD, os elementos do vetor serão contíguos na memória, compartilhando a mesma linha de cache... Resultando em uma baixa escalabilidade.

**Solução:** Colocar espaçadores “Pad” para que os elementos usem linhas distintas de cache.

## EXEMPLO: ELIMINANDO FALSO COMPARTILHAMENTO COM PADDING NO VETOR DE SOMAS

```
#include <omp.h>
static long num_steps = 100000; double step;
#define PAD 8 // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main () {
int i, nthreads; double pi, sum[NUM_THREADS][PAD];
step = 1.0/(double) num_steps;
#pragma omp parallel num_threads(NUM_THREADS)
{
int i, id,nthrds; double x;
id = omp_get_thread_num();
nthrds = omp_get_num_threads();
if (id == 0) nthreads = nthrds;
for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
x = (i+0.5)*step;
sum[id][0] += 4.0/(1.0+x*x);
}
}
for(i=0, pi=0.0;i<nthreads;i++) pi += sum[i][0] * step;
}
```

Espaça o vetor para que o valor de cada soma fique em uma linha diferente de cache

# DETECÇÃO DE FALSO COMPARTILHAMENTO

Detectar falso compartilhamento é um **problema difícil**, mesmo com código pequeno.

Depende do:

- **Compilador** (gcc, icc, ...),
- **Opções do compilador** (alinhamento),
- **Arquitetura** (tamanho de int, tamanho de estruturas, alinhamento, ...),
- entre outros.

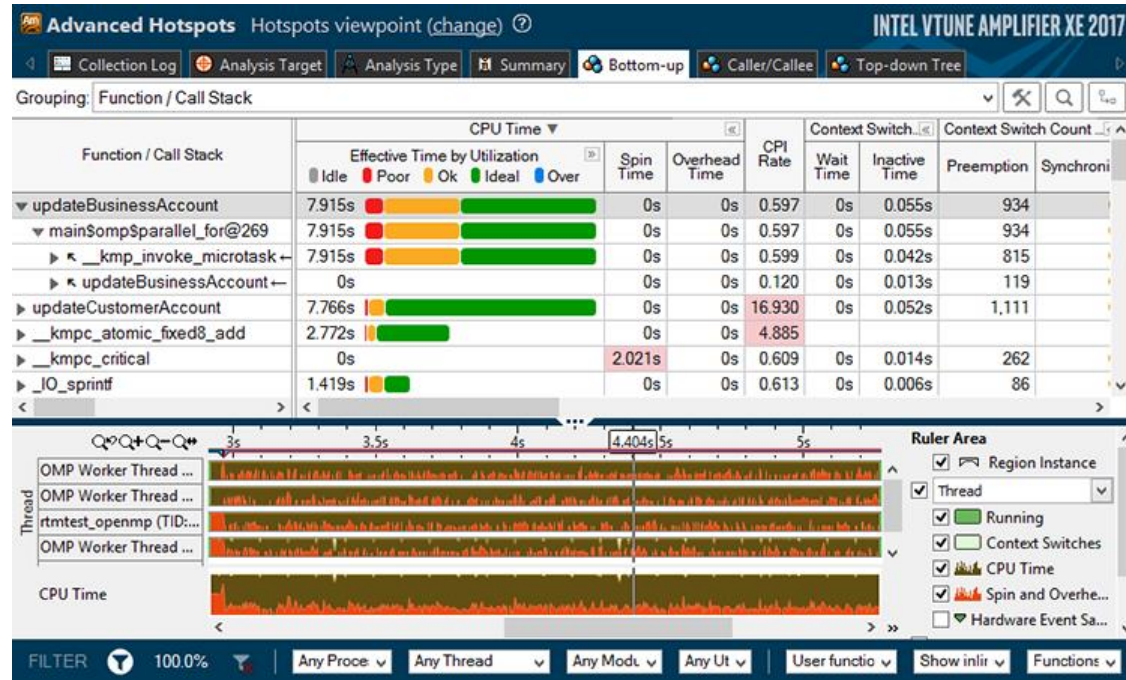
Precisa-se de ferramentas para ajudar na detecção de falso compartilhamento.

Vamos ver uma ferramenta da Intel, chamada **VTune**.

# INTEL VTUNE

Ferramenta da Intel para análise de aplicações.

1. Execução da aplicação.
2. VTune pega informações de *hardware counters* da máquina.
3. Vtune visualize os resultados.



<https://software.intel.com/en-us/intel-vtune-amplifier-xe>

Nosso caso: estatísticas da memória.  
Código Pi deveria ser *cpu-bound*.

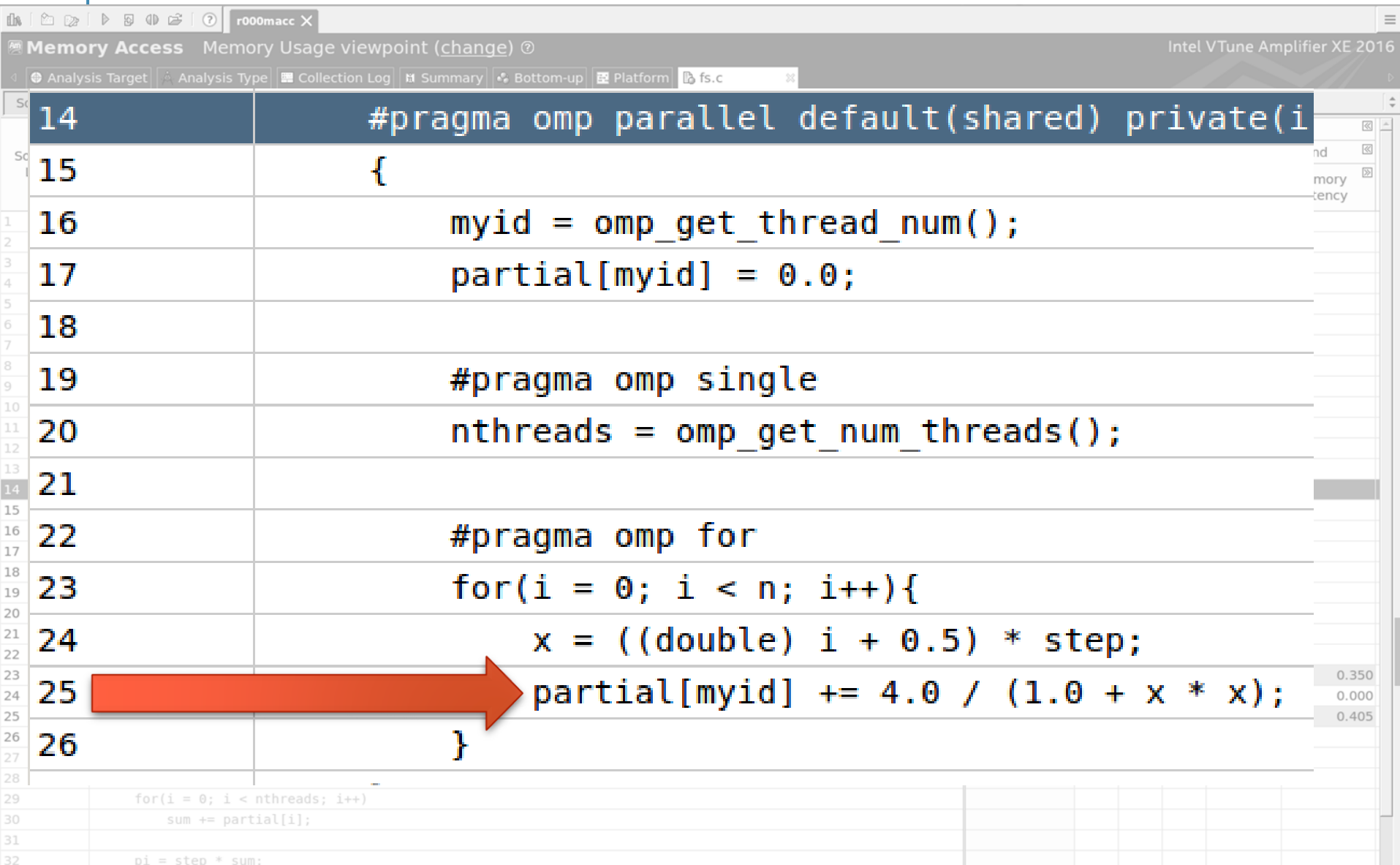
# EXEMPLO DE VTUNE – COM FALSO COMP.

| Memory Access Memory Usage viewpoint (change) ?                              |                                                          |          |              |          |          |                  | Intel VTune Amplifier XE 2016 |  |
|------------------------------------------------------------------------------|----------------------------------------------------------|----------|--------------|----------|----------|------------------|-------------------------------|--|
| Analysis Target Analysis Type Collection Log Summary Bottom-up Platform fs.c |                                                          |          |              |          |          |                  |                               |  |
| Source Assembly Assembly grouping: Address                                   |                                                          |          |              |          |          |                  |                               |  |
| Source Line                                                                  | Source                                                   | CPU Time | Memory Bound |          |          |                  |                               |  |
|                                                                              |                                                          |          | L1 Bou..     | L2 Bou.. | L3 Bou.. | DRAM Bound       |                               |  |
|                                                                              |                                                          |          |              |          |          | Memory Bandwidth | Memory Latency                |  |
| 1                                                                            | #include <omp.h>                                         |          |              |          |          |                  |                               |  |
| 2                                                                            |                                                          |          |              |          |          |                  |                               |  |
| 3                                                                            | #define MAX_THREADS 228                                  |          |              |          |          |                  |                               |  |
| 4                                                                            |                                                          |          |              |          |          |                  |                               |  |
| 5                                                                            | double partial[MAX_THREADS];                             |          |              |          |          |                  |                               |  |
| 6                                                                            |                                                          |          |              |          |          |                  |                               |  |
| 7                                                                            | double piCalc(long long int n){                          |          |              |          |          |                  |                               |  |
| 8                                                                            | long long int i;                                         |          |              |          |          |                  |                               |  |
| 9                                                                            | double x, pi, sum = 0.0;                                 |          |              |          |          |                  |                               |  |
| 10                                                                           | const double step = 1.0 / (double) n;                    |          |              |          |          |                  |                               |  |
| 11                                                                           |                                                          |          |              |          |          |                  |                               |  |
| 12                                                                           | int nthreads, myid;                                      |          |              |          |          |                  |                               |  |
| 13                                                                           |                                                          |          |              |          |          |                  |                               |  |
| 14                                                                           | #pragma omp parallel default(shared) private(i, x, myid) |          |              |          |          |                  |                               |  |
| 15                                                                           | {                                                        |          |              |          |          |                  |                               |  |
| 16                                                                           | myid = omp_get_thread_num();                             |          |              |          |          |                  |                               |  |
| 17                                                                           | partial[myid] = 0.0;                                     |          |              |          |          |                  |                               |  |
| 18                                                                           |                                                          |          |              |          |          |                  |                               |  |
| 19                                                                           | #pragma omp single                                       |          |              |          |          |                  |                               |  |
| 20                                                                           | nthreads = omp_get_num_threads();                        |          |              |          |          |                  |                               |  |
| 21                                                                           |                                                          |          |              |          |          |                  |                               |  |
| 22                                                                           | #pragma omp for                                          |          |              |          |          |                  |                               |  |
| 23                                                                           | for(i = 0; i < n; i++){                                  | 41.630s  | 0.004        | 0.000    | 0.000    | 0.403            | 0.350                         |  |
| 24                                                                           | x = ((double) i + 0.5) * step;                           | 2.270s   | 0.200        | 0.000    | 0.000    | 0.571            | 0.000                         |  |
| 25                                                                           | partial[myid] += 4.0 / (1.0 + x * x);                    | 37.580s  | 0.287        | 0.000    | 0.000    | 0.493            | 0.405                         |  |
| 26                                                                           | }                                                        |          |              |          |          |                  |                               |  |
| 27                                                                           | }                                                        |          |              |          |          |                  |                               |  |
| 28                                                                           |                                                          |          |              |          |          |                  |                               |  |
| 29                                                                           | for(i = 0; i < nthreads; i++){                           |          |              |          |          |                  |                               |  |
| 30                                                                           | sum += partial[i];                                       |          |              |          |          |                  |                               |  |
| 31                                                                           |                                                          |          |              |          |          |                  |                               |  |
| 32                                                                           | pi = step * sum;                                         |          |              |          |          |                  |                               |  |

Essa linha de código é (main) memory-bound.

Essa linha de código é (main) *memory-bound*.

# EXEMPLO DE VTUNE – COM FALSO COMP.



The screenshot displays the Intel VTune Amplifier XE 2016 interface. The top bar shows the application name and version. The main window is titled "Memory Access" and "Memory Usage viewpoint (change)". The left sidebar contains navigation tabs: Analysis Target, Analysis Type, Collection Log, Summary, Bottom-up, Platform, and fs.c. The main area shows a code snippet with line numbers 14 through 26. A red arrow points to line 25, which contains the code: `partial[myid] += 4.0 / (1.0 + x * x);`. The code snippet is as follows:

```
14 #pragma omp parallel default(shared) private(i)
15 {
16     myid = omp_get_thread_num();
17     partial[myid] = 0.0;
18
19     #pragma omp single
20     nthreads = omp_get_num_threads();
21
22     #pragma omp for
23     for(i = 0; i < n; i++){
24         x = ((double) i + 0.5) * step;
25         partial[myid] += 4.0 / (1.0 + x * x);
26     }
```

The right sidebar shows a table with memory usage data:

| Memory Usage |
|--------------|
| 0.350        |
| 0.000        |
| 0.405        |

# EXEMPLO DE VTUNE – SEM FALSO COMP.

Intel VTune Amplifier XE 2016

Memory Access Memory Usage viewpoint (change) @

Collection Log Analysis Target Analysis Type Summary Bottom-up Platform padding\_6...

Source Assembly Assembly grouping: Address

| Source Line | Source                                                                            | CPU Time | Memory Bound |          |          |                  |                |
|-------------|-----------------------------------------------------------------------------------|----------|--------------|----------|----------|------------------|----------------|
|             |                                                                                   |          | L1 Bou..     | L2 Bou.. | L3 Bou.. | DRAM Bound       | Memory Latency |
|             |                                                                                   |          |              |          |          | Memory Bandwidth | Memory Latency |
| 9           | struct element_t {                                                                |          |              |          |          |                  |                |
| 10          | double sum;                                                                       |          |              |          |          |                  |                |
| 11          | } __attribute__((aligned (CACHE_LINE_SIZE)));                                     |          |              |          |          |                  |                |
| 12          |                                                                                   |          |              |          |          |                  |                |
| 13          | struct element_t partial[MAX_THREADS] __attribute__((aligned (CACHE_LINE_SIZE))); |          |              |          |          |                  |                |
| 14          |                                                                                   |          |              |          |          |                  |                |
| 15          | double piCalc(long long int n){                                                   |          |              |          |          |                  |                |
| 16          | long long int i;                                                                  |          |              |          |          |                  |                |
| 17          | double x, pi, sum = 0.0;                                                          |          |              |          |          |                  |                |
| 18          | const double step = 1.0 / (double) n;                                             |          |              |          |          |                  |                |
| 19          |                                                                                   |          |              |          |          |                  |                |
| 20          | int nthreads, myid;                                                               |          |              |          |          |                  |                |
| 21          |                                                                                   |          |              |          |          |                  |                |
| 22          | #pragma omp parallel default(shared) private(i, x, myid)                          |          |              |          |          |                  |                |
| 23          | {                                                                                 |          |              |          |          |                  |                |
| 24          | myid = omp_get_thread_num();                                                      |          |              |          |          |                  |                |
| 25          | partial[myid].sum = 0.0;                                                          |          |              |          |          |                  |                |
| 26          |                                                                                   |          |              |          |          |                  |                |
| 27          | #pragma omp single                                                                |          |              |          |          |                  |                |
| 28          | nthreads = omp_get_num_threads();                                                 |          |              |          |          |                  |                |
| 29          |                                                                                   |          |              |          |          |                  |                |
| 30          | #pragma omp for                                                                   |          |              |          |          |                  |                |
| 31          | for(i = 0; i < n; i++){                                                           | 0.352s   | 0.530        | 0.000    | 0.000    | 0.016            | 0.670          |
| 32          | x = ((double) i + 0.5) * step;                                                    | 1.648s   | 0.272        | 0.000    | 0.000    | 0.057            | 0.684          |
| 33          | partial[myid].sum += 4.0 / (1.0 + x * x);                                         | 10.949s  | 0.468        | 0.000    | 0.000    | 0.039            | 0.599          |
| 34          | }                                                                                 |          |              |          |          |                  |                |
| 35          | }                                                                                 |          |              |          |          |                  |                |
| 36          |                                                                                   |          |              |          |          |                  |                |
| 37          | for(i = 0; i < nthreads; i++)                                                     |          |              |          |          |                  |                |
| 38          | sum += partial[i].sum;                                                            |          |              |          |          |                  |                |
| 39          |                                                                                   |          |              |          |          |                  |                |
| 40          | pi = step * sum;                                                                  |          |              |          |          |                  |                |

Não é mais *memory-bound*.

INTEL MODERN CODE PARTNER 72

# RESULTADOS\*

O Pi original sequencial com 100mi passos, executou em **1.83** seg.

\*Compilador Intel (icpc) sem otimizações em um Apple OS X 10.7.3 com dual core (4 HW threads) processador Intel® Core TM i5 1.7Ghz e 4 Gbyte de memória DDR3 1.333 Ghz.

| Threads | 1 SPMD | 1 SPMD padding |
|---------|--------|----------------|
| 1       | 1.86   | 1.86           |
| 2       | 1.03   | 1.01           |
| 3       | 1.08   | 0.69           |
| 4       | 0.97   | 0.53           |

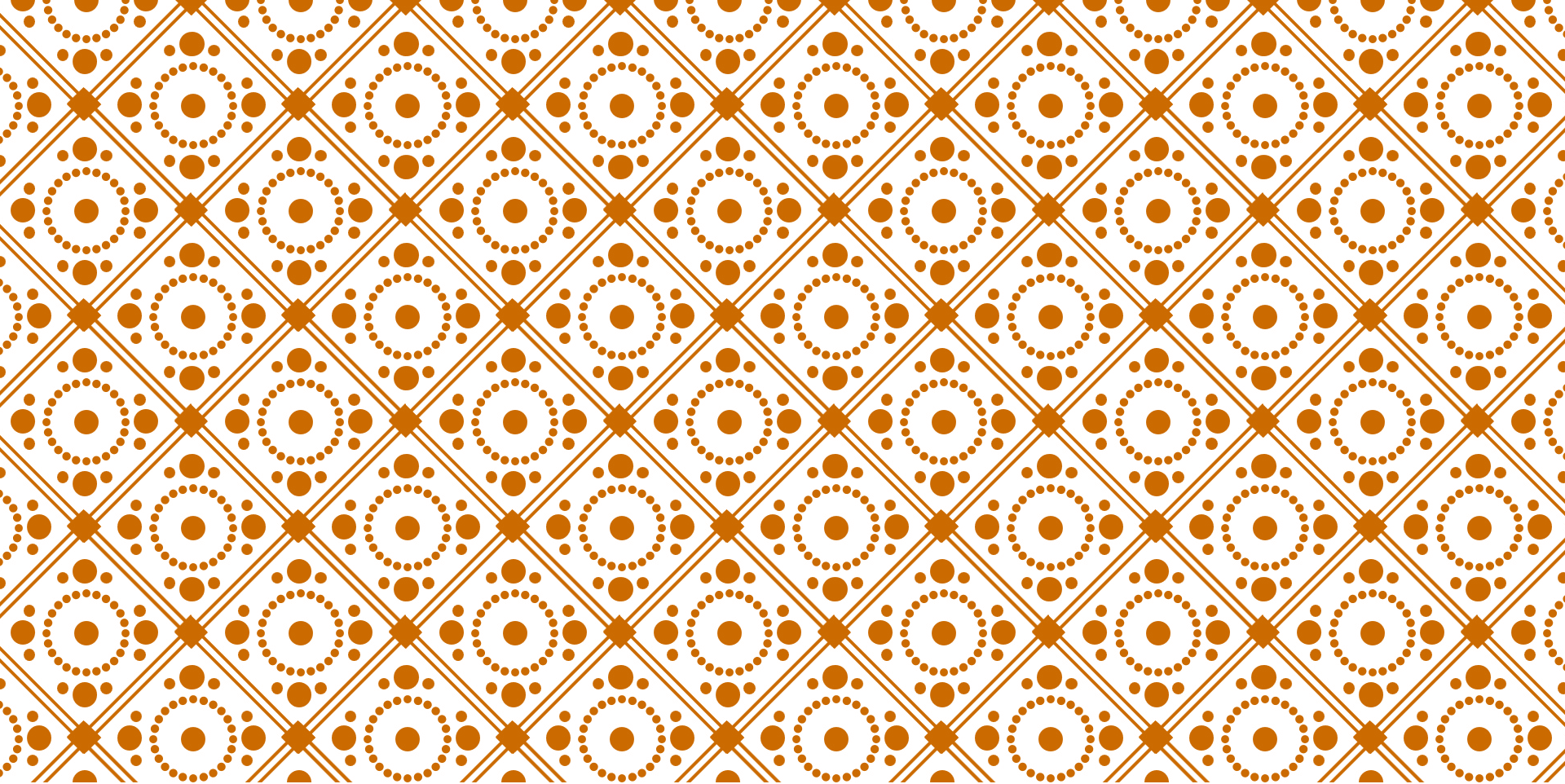
# REALMENTE PRECISAMOS ESPAÇAR NOSSOS VETORES?

**Aquilo foi feio!**

Espaçar vetores requer conhecimento profundo da arquitetura de cache.

Mova seu programa para uma máquina com tamanho diferente de linhas de cache, e o desempenho desaparece.

**Deve existir uma forma melhor para lidar com falso compartilhamento.**



# SINCRONIZAÇÃO (REVISITANDO O PROGRAMA PI)

# VISÃO GERAL DO OPENMP: COMO AS THREADS INTERAGEM?

Threads comunicam-se através de variáveis compartilhadas.

Compartilhamento de dados não intencional causa condições de corrida: quando a saída do programa muda conforme as threads são escalonadas de forma diferente.

Para controlar condições de corrida: Use sincronização para proteger os conflitos de dados.

Mude como os dados serão acessados para minimizar a necessidade de sincronizações.

# SINCRONIZAÇÃO

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

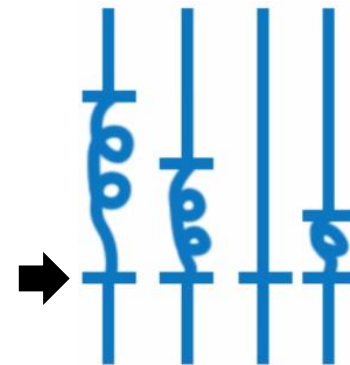
As duas formas mais comuns de sincronização são:

# SINCRONIZAÇÃO

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



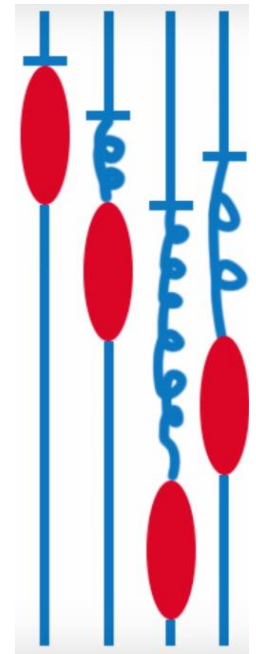
# SINCRONIZAÇÃO

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



# SINCRONIZAÇÃO

## Sincronização de alto nível:

- critical
- atomic
- barrier
- ordered

## Sincronização de baixo nível:

- flush
- locks (both simple and nested)

Sincronização é usada para impor regras de ordem e para proteger acessos a dados compartilhados

Vamos falar sobre esses mais tarde!

# SINCRONIZAÇÃO: BARRIER

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

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

    #pragma omp barrier

    B[id] = big_calc2(id, A); // Vamos usar o valor A computado
}
```

# SINCRONIZAÇÃO: CRITICAL

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

```
float res;  
#pragma omp parallel  
{ float B; int i, id, nthrds;  
  id = omp_get_thread_num();  
  nthrds = omp_get_num_threads();  
  for(i=id; i<niters; i+=nthrds){  
    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 (FORMA BÁSICA)

Formas adicionais foram incluídas no OpenMP 3.1.

Atomic prove exclusão mútua para apenas para atualizações na memória (a atualização de X no exemplo a seguir)

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

Use uma  
instrução  
especial se  
disponível

A declaração dentro de atomic deve ser uma das seguintes:

**x Op= expr**

**x++**

**++x**

**x--**

**--x**

**x** é um valor escalar

**Op** é um operador não sobrecarregado.

# EXERCÍCIO 3

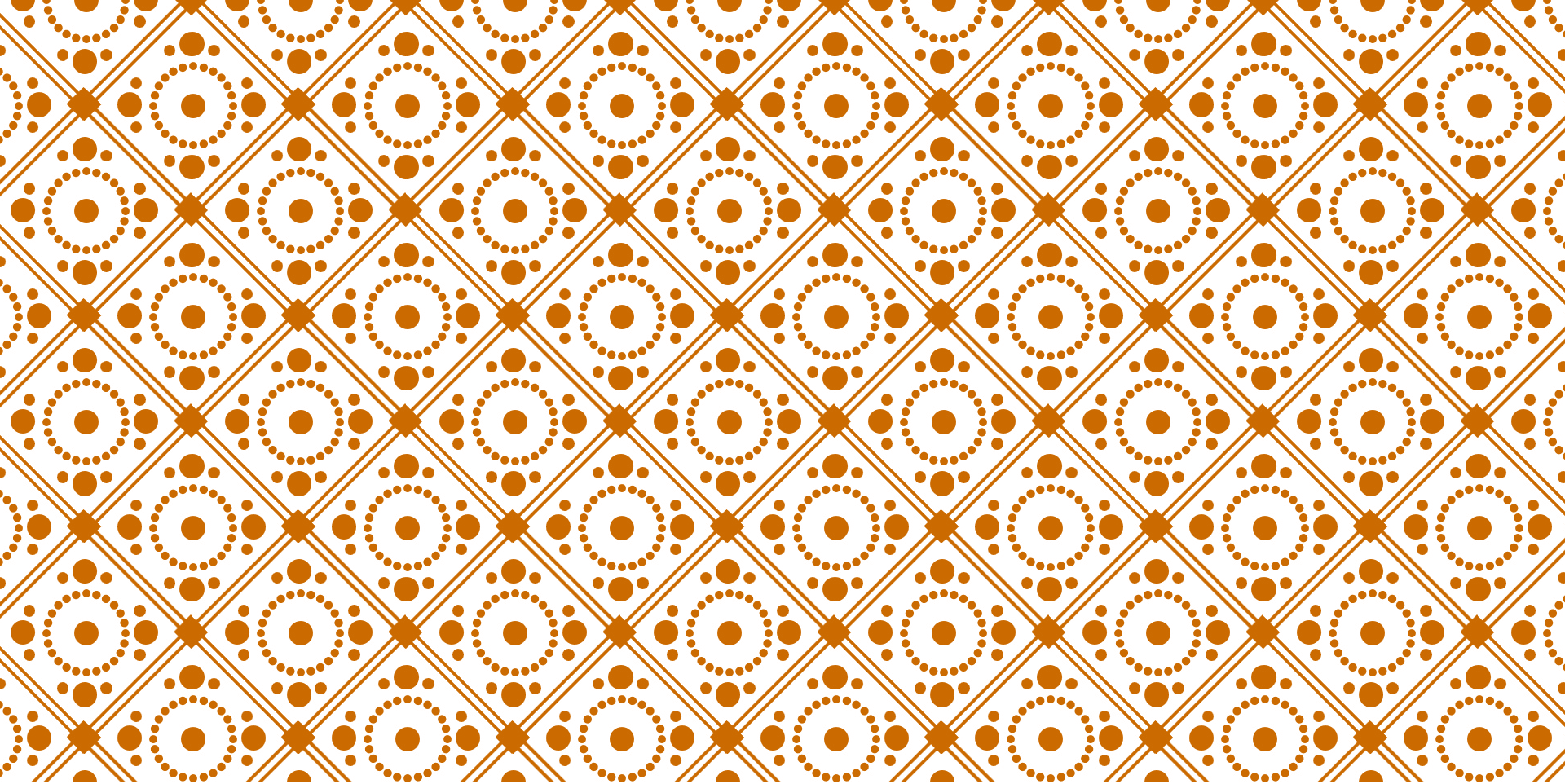
No exercício 2, provavelmente foi usado um vetor para cada thread armazenar o valor de sua soma parcial.

Se elementos do vetor estiverem compartilhando a mesma linha de cache, teremos falso compartilhamento.

- Dados não compartilhados que compartilham a mesma linha de cache, fazendo com que cada atualização invalide a linha de cache... em essência ping-pong de dados entre as threads.

**Modifique seu programa pi do exercício 2 para evitar falso compartilhamento** devido ao vetor de soma.

Lembre-se que ao promover a soma a um vetor fez a codificação ser fácil, mas levou a falso compartilhamento e baixo desempenho.



# OVERHEAD DE SINCRONIZAÇÃO E ELIMINAÇÃO DE FALSO COMPARTILHAMENTO

# EXEMPLO: USANDO SEÇÃO CRÍTICA PARA REMOVER A CONDIÇÃO DE CORRIDA

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main () {
    double pi = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    { int i, id, nthrds;
      double x, sum = 0.0;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
      for (i=id, sum=0.0; i< num_steps; i=i+nthrds) {
          x = (i + 0.5) * step;
          sum += 4.0 / (1.0 + x*x);
      }
      #pragma omp critical
      pi += sum * step;
    }
}
```

Cria um escalar local para cada thread acumular a soma parcial

Sem vetor, logo sem falso compartilhamento

A soma estará fora de escopo além da região paralela.... Assim devemos somar aqui. Devemos proteger a soma para pi em uma região critica para que as atualizações não conflitem

# RESULTADOS\*

O Pi original sequencial com 100mi passos, executou em **1.83** seg.

\*Compilador Intel (icpc) sem otimizações em um Apple OS X 10.7.3 com dual core (4 HW threads) processador Intel® Core TM i5 1.7Ghz e 4 Gbyte de memória DDR3 1.333 Ghz.

| Threads | 1. SPMD | 1. SPMD padding | SPMD critical |
|---------|---------|-----------------|---------------|
| 1       | 1.86    | 1.86            | 1.87          |
| 2       | 1.03    | 1.01            | 1.00          |
| 3       | 1.08    | 0.69            | 0.68          |
| 4       | 0.97    | 0.53            | 0.53          |

# EXEMPLO: USANDO SEÇÃO CRÍTICA PARA REMOVER A CONDIÇÃO DE CORRIDA

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main () {
    double pi = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    { int i, id, nthrds; double x;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
      for (i=id; i < num_steps; i = i+nthrds) {
          x = (i+0.5)*step;
          #pragma omp critical
          pi += 4.0/(1.0+x*x);
      }
    }
    pi *= step;
}
```

Atenção onde você irá colocar a seção crítica

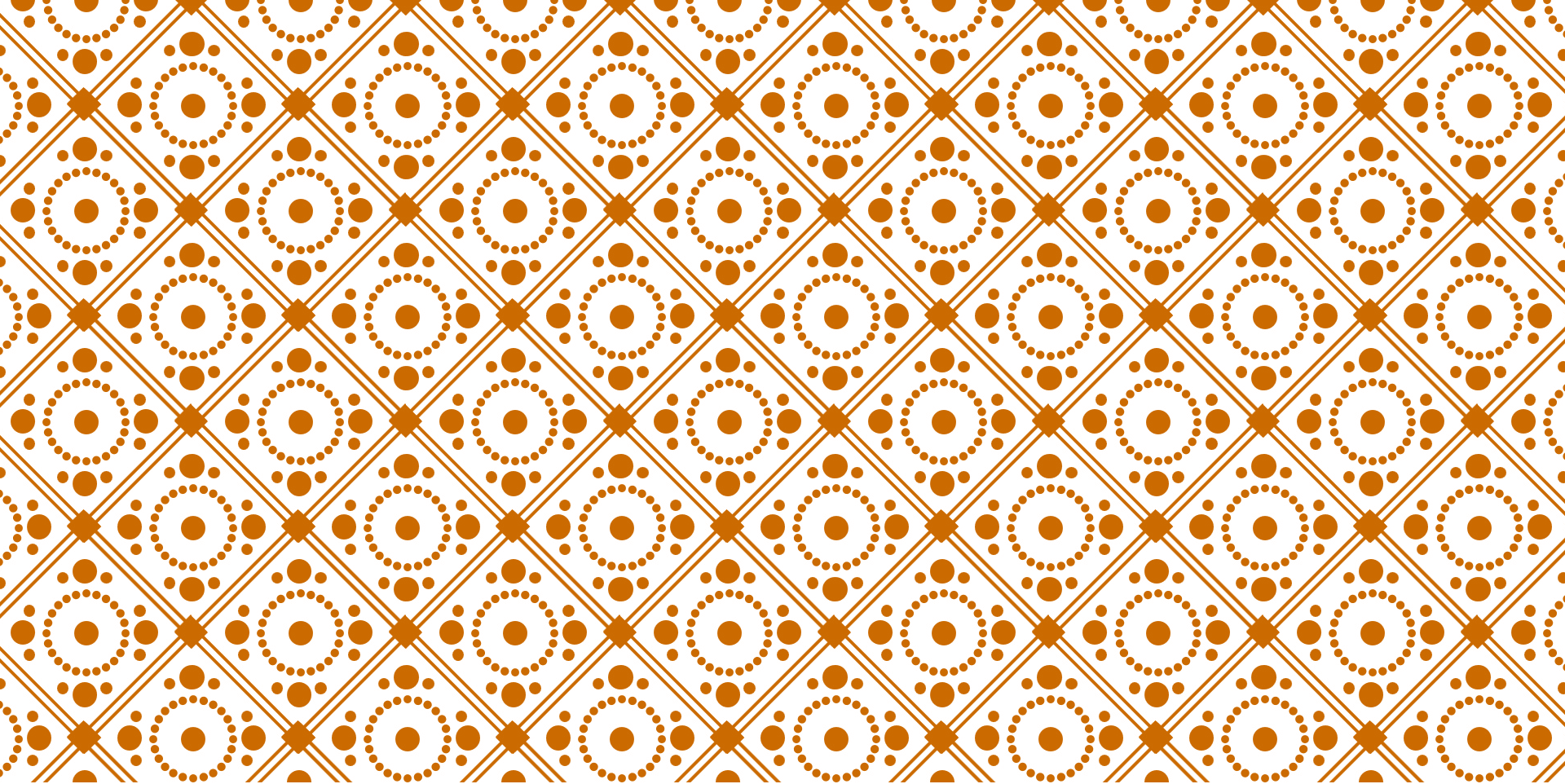
O que acontece se colocarmos a seção crítica dentro do loop?

Tempo execução sequencial + overhead

# EXEMPLO: USANDO UM ATOMIC PARA REMOVER A CONDIÇÃO DE CORRIDA

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main () {
    double pi = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    { int i, id, nthrds;
      double x, sum = 0.0;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
      for (i=id, sum=0.0; i< num_steps; i=i+nthrds) {
          x = (i + 0.5) * step;
          sum += 4.0 / (1.0 + x*x);
      }
      #pragma omp atomic
      pi += sum * step;
    }
}
```

Se o hardware possuir uma instrução de soma atômica, o compilador irá usar aqui, reduzindo o custo da operação



# LAÇOS PARALELOS (SIMPLIFICANDO O PROGRAMA PI)

# SPMD VS. WORKSHARING

A construção *parallel* por si só cria um programa SPMD (Single Program Multiple Data)... i.e., cada thread executa de forma redundante o mesmo código.

Como dividir os caminhos dentro do código entre as threads?

Isso é chamado de worksharing (divisão de trabalho)

- Loop construct
- Sections/section constructs
- Single construct
- Task construct

Veremos mais tarde

# CONSTRUÇÕES DE DIVISÃO DE LAÇOS

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

Nome da construção:  
C/C++: **for**  
Fortran: **do**

```
#pragma omp parallel
{
    #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];}
```

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 trabalho**

```
#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

Atalho OpenMP: Coloque o “**parallel**” e a diretiva de divisão de trabalho na mesma linha

```
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();
    }
```

# 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

`schedule(dynamic[,chunk])`

- Cada thread pega um “chunk” de iterações da fila até que todas as iterações sejam executadas.

`schedule(guided[,chunk])`

- As threads pegam blocos de iterações dinamicamente, iniciando de blocos grandes reduzindo até o tamanho “chunk”.

`schedule(runtime)`

- O modelo de distribuição e o tamanho serão pegos da variável de ambiente `OMP_SCHEDULE`.

`schedule(auto)`  **Novo**

- 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                                        |
| DYNAMIC          | Imprevisível, quantidade de trabalho por iteração altamente variável                 |
| 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 |

Menos trabalho durante a execução (mais durante a compilação)

Mais trabalho durante a execução (lógica complexa de controle)

# TRABALHANDO COM LAÇOS

## Abordagem básica:

- Encontre laços com computação intensiva
- Transforme as iterações em operações independentes (assim as iterações podem ser executadas em qualquer ordem sem problemas)
- Adicione a diretiva OpenMP apropriada e teste

```
int i, j, A[MAX];  
j = 5;  
for (i=0; i< MAX; i++) {  
    j +=2;  
    A[i] = big(j);  
}
```

Onde está a  
dependência  
aqui?

# TRABALHANDO COM LAÇOS

## Abordagem básica:

- Encontre laços com computação intensiva
- Transforme as iterações em operações independentes (assim as iterações podem ser executadas em qualquer ordem sem problemas)
- Adicione a diretiva OpenMP apropriada e teste

```
int i, j, A[MAX];  
j = 5;  
for (i=0; i< MAX; i++) {  
    j +=2;  
    A[i] = big(j);  
}
```

Note que o índice  
“i” será privado  
por padrão

Remove a  
dependência  
dentro do laço

```
int i, A[MAX];  
#pragma omp parallel for  
for (i=0; i< MAX; i++) {  
    int j = 5 + 2*(i+1);  
    A[i] = big(j);  
}
```

# LAÇOS ANINHADOS

Pode ser útil em casos onde o laço interno possua desbalanceamento  
Irá gerar um laço de tamanho  $N \times M$  e torna-lo paralelo

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
    for (int j=0; j<M; j++) {
        .....
    }
}
```

Número de laços a serem  
paralelizados, contando de  
fora para dentro

# EXERCÍCIO 4: PI COM LAÇOS

Retorne ao programa Pi sequencial e **paralelize com as construções de laço**

Nosso objetivo é minimizar o número de modificações feitas no programa original.

# REDUÇÃO

Como podemos proceder nesse caso?

```
double media=0.0, A[MAX]; int i;
for (i=0;i< MAX; i++) {
    media += A[i];
}
media = media / MAX;
```

Devemos combinar os valores em uma variável acumulação única (*media*) ... existe uma condição de corrida na variável compartilhada

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

# REDUÇÃO

A diretiva OpenMP reduction:                      reduction (op : list)

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 “+”).
- Atualizações acontecem na cópia local.
- Cópias locais são “reduzidas” para uma única variável original (global).

A variável na “lista” deve ser compartilhada entre as threads.

```
double ave=0.0, A[MAX]; int i;
#pragma omp parallel for reduction (+:ave)
    for (i=0;i< MAX; i++) {
        ave + = A[i];
    }
ave = ave/MAX;
```

# REDUÇÃO OPERANDOS E VALORES INICIAIS

Vários operandos associativos podem ser utilizados com **reduction**:

Valores iniciais são os que fazer sentido (elemento nulo)

| Operador | Valor Inicial         |
|----------|-----------------------|
| +        | 0                     |
| *        | 1                     |
| -        | 0                     |
| Min      | Maior número possível |
| Max      | Menor número possível |

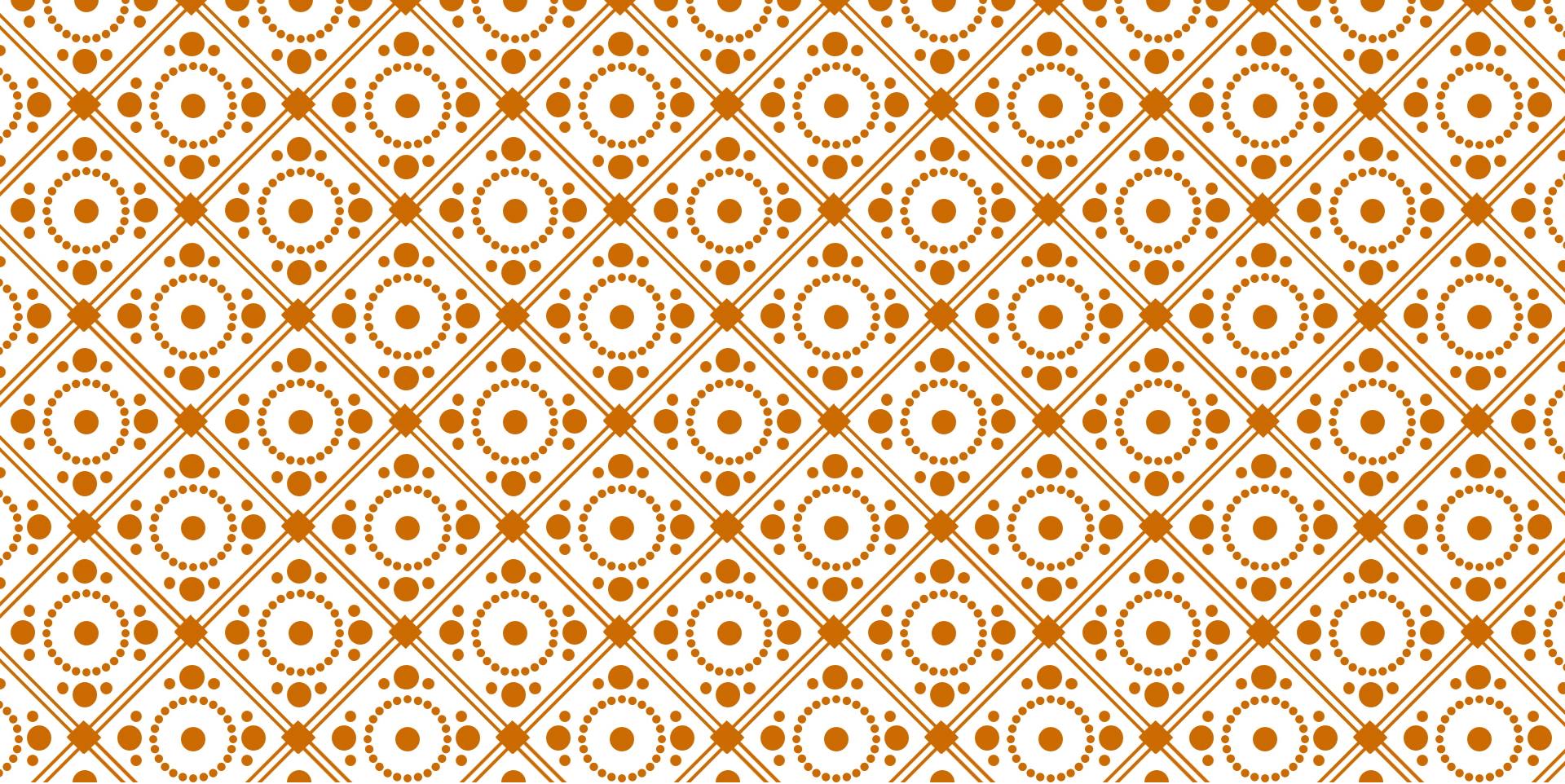
| Operador | Valor Inicial |
|----------|---------------|
| &        | $\sim 0$      |
|          | 0             |
| ^        | 0             |
| &&       | 1             |
|          | 0             |

Apenas para  
C e C++

# EXERCÍCIO 5: PI COM LAÇOS

Retorne ao programa Pi e paralelize com as construções de laço e **use redução para retornar o resultado**

Nosso objetivo é minimizar o número de modificações feitas no programa original.



# PROGRAMA PI COMPLETO

# SERIAL PI PROGRAM

```
static long num_steps = 100000;
double step;
int main ()
{ int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  for (i=0;i< num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

# EXEMPLO: PI COM UM LAÇO E REDUÇÃO

```
#include <omp.h>
static long num_steps = 100000; double step;
void main ()
{ int i; double pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  #pragma omp parallel
  {
    double x;
    #pragma omp for reduction(+:sum)
    for (i=0;i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
    }
  }
  pi = step * sum;
}
```

Cria um time de threads ...  
sem a construção paralela, nunca  
veremos mais que uma thread

Cria um escalar local para cada  
thread para armazenar o valor de  
x de cada iteração/thread

Quebra o laço e distribui as  
iterações entre as threads...  
Fazendo a redução dentro de sum.  
Note que o índice do laço será  
privado por padrão

# EXEMPLO: PI COM UM LAÇO E REDUÇÃO

```
#include <omp.h>
static long num_steps = 100000; double step;
void main ()
{ int i; double pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  #pragma omp parallel for reduction(+:sum)
  for (i=0;i< num_steps; i++){
      double x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

# RESULTADOS\*

O Pi original sequencial com 100mi passos, executou em **1.83** seg.

\*Compilador Intel (icpc) sem otimizações em um Apple OS X 10.7.3 com dual core (4 HW threads) processador Intel® Core TM i5 1.7Ghz e 4 Gbyte de memória DDR3 1.333 Ghz.

| Threads | 1. SPMD | SPMD padding | SPMD critical | Pi Loop |
|---------|---------|--------------|---------------|---------|
| 1       | 1.86    | 1.86         | 1.87          | 1.91    |
| 2       | 1.03    | 1.01         | 1.00          | 1.02    |
| 3       | 1.08    | 0.69         | 0.68          | 0.80    |
| 4       | 0.97    | 0.53         | 0.53          | 0.68    |

# LAÇOS (CONTINUAÇÃO)

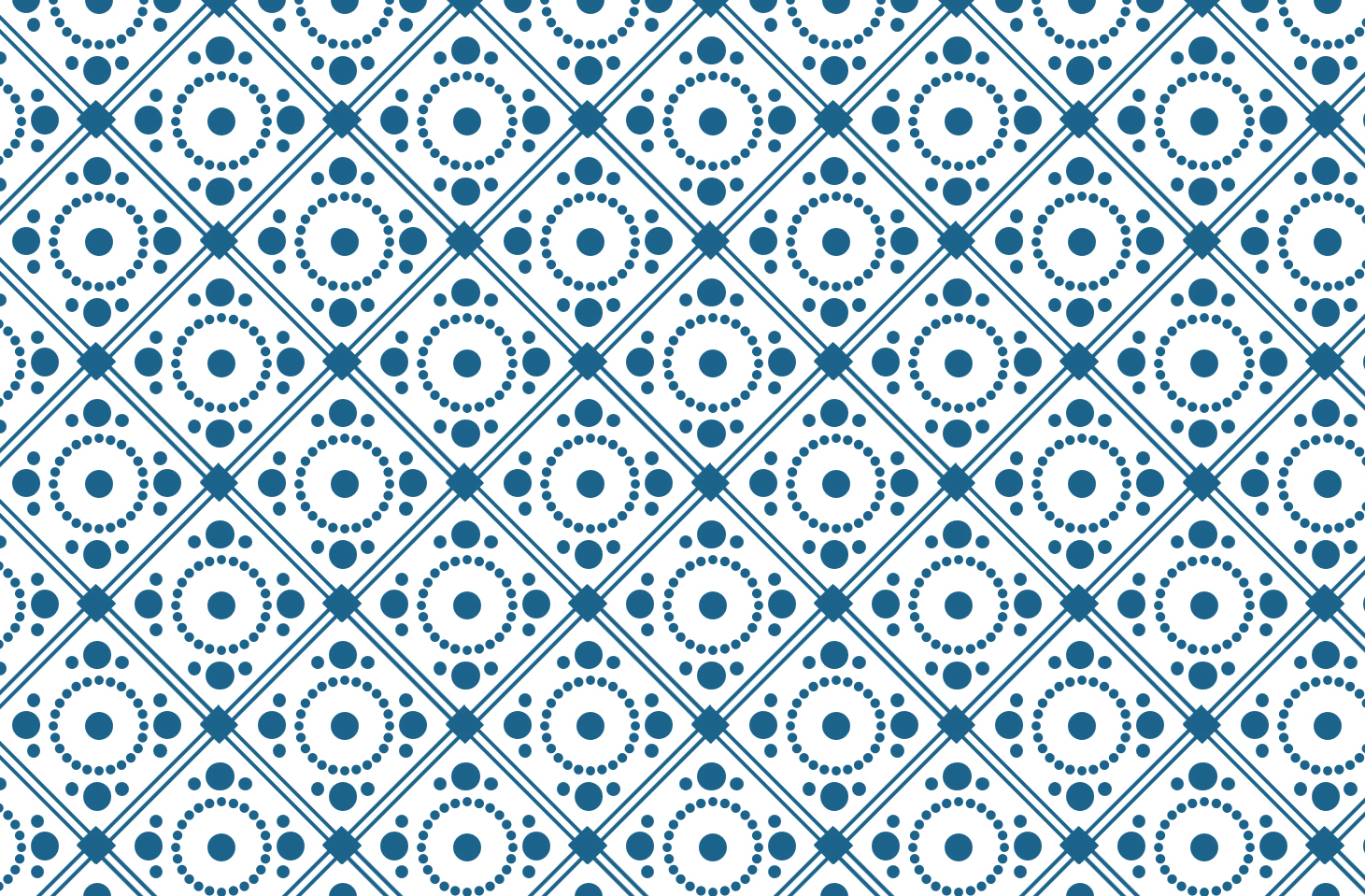
## Novidades do OpenMP 3.0

### Tornou o `schedule(runtime)` mais útil

- Pode obter/definir o escalonamento dentro de bibliotecas
- `omp_set_schedule()`
- `omp_get_schedule()`
- Permite que as implementações escolham suas formas de escalonar

Adicionado também o tipo de escalonamento **AUTO** que provê liberdade para o ambiente de execução determinar a forma de distribuir as iterações.

Permite iterators de acesso aleatório de C++ como variáveis de controle de laços paralelos



# INTEL MODERN CODE PARTNER

## OPENMP — AULA 03

# AGENDA GERAL

## Unit 1: Getting started with OpenMP

- Mod 1: Introduction to parallel programming
- Mod 2: The boring bits: Using an OpenMP compiler (hello world)
- Disc 1: Hello world and how threads work

## Unit 2: The core features of OpenMP

- Mod 3: Creating Threads (the Pi program)
- Disc 2: The simple Pi program and why it sucks
- Mod 4: Synchronization (Pi program revisited)
- Disc 3: Synchronization overhead and eliminating false sharing
- Mod 5: Parallel Loops (making the Pi program simple)
- Disc 4: Pi program wrap-up

## Unit 3: Working with OpenMP

- Mod 6: Synchronize single masters and stuff
- Mod 7: Data environment
- Disc 5: Debugging OpenMP programs

## Unit 4: a few advanced OpenMP topics

- Mod 8: Skills practice ... linked lists and OpenMP
- Disc 6: Different ways to traverse linked lists
- Mod 8: Tasks (linked lists the easy way)
- Disc 7: Understanding Tasks

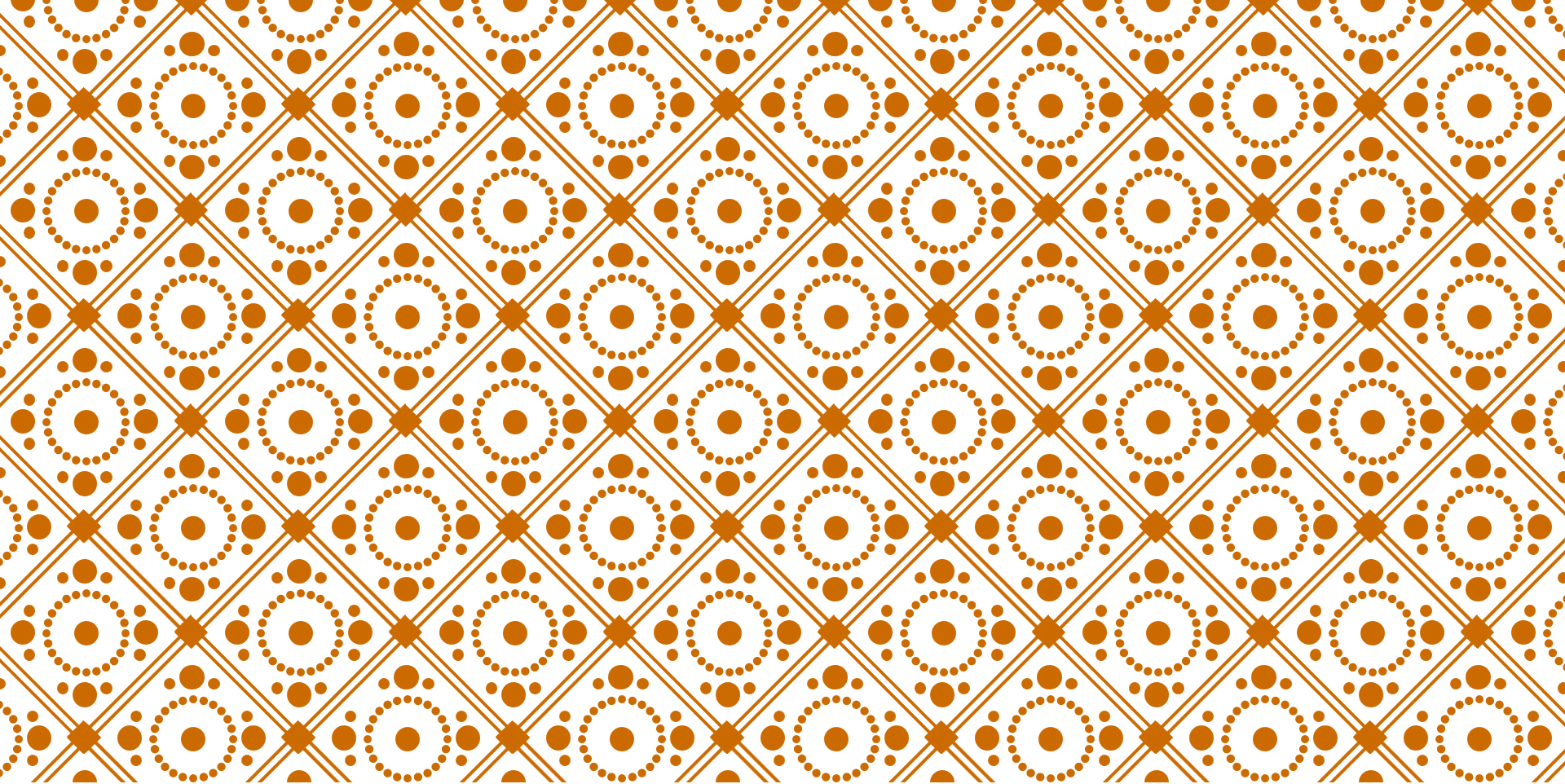
## Unit 5: Recapitulation

- Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
- Disc 8: The pitfalls of pairwise synchronization
- Mod 9: Threadprivate Data and how to support libraries (Pi again)
- Disc 9: Random number generators

# AGENDA DESSA AULA

## **Unit 3: Working with OpenMP**

- Mod 6: Synchronize single masters and stuff
- Mod 7: Data environment
- Disc 5: Debugging OpenMP programs



# SINCRONIZAÇÃO: SINGLE, MASTER, ETC.

# SINCRONIZAÇÃO: BARRIER E NOWAIT

**Barrier:** Cada thread aguarda até que todas as demais cheguem

```
#pragma omp parallel shared (A, B, C) private(id)
{
    id = omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
        for(i=0; i<N; i++){
            C[i] = big_calc3(i, A);
        }
    #pragma omp for nowait
        for(i=0; i<N; i++){
            B[i] = big_calc2(C, i);
        }
    A[id] = big_calc4(id);
}
```

**Barreira implícita** no final da construção FOR

**Sem barreira implícita** devido ao nowait (use com cuidado)

**Barreira implícita** ao final na região paralela (não podemos desligar essa)

# CONSTRUÇÃO MASTER

A construção master denota um bloco estruturado que será executado apenas pela thread master (id=0).

As outras threads apenas ignoram (**sem barreira implícita**)

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    {
        exchange_boundaries();
    }

    #pragma omp barrier
    do_many_other_things();
}
```

Sem barreira implícita

Barreira explícita

# CONSTRUÇÃO SINGLE

A construção single denota um bloco de código que deverá ser executado apenas por uma thread (não precisar ser a thread master).

Uma **barreira implícita** estará no final do bloco single (podemos remover essa barreira com a diretiva **nowait**).

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp single
    {
        exchange_boundaries();
    }
    do_many_other_things();
}
```

**Barreira implícita**

Nesse caso, podemos usar “**nowait**”

# CONSTRUÇÃO **SECTIONS** PARA DIVISÃO DE TRABALHO

A construção de divisão de trabalho com **sections** prove um bloco estruturado diferente para cada thread.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        x_calculation();
        #pragma omp section
        y_calculation();
        #pragma omp section
        z_calculation();
    }
}
```

Por padrão, existe uma barreira implícita no final do “omp sections”.

Use a diretiva “nowait” para desligar essa barreira.

# SINCRONIZAÇÃO: ROTINAS LOCK

**Rotinas simples de Lock:** Um lock está disponível caso esteja “não setado” (unset).

- `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`, `omp_destroy_lock()`

**Locks aninhados:** Um lock aninhado está disponível se estiver “unset” ou se está “set” e o dono for a thread que estiver executando a função com lock aninhado.

- `omp_init_nest_lock()`, `omp_set_nest_lock()`, `omp_unset_nest_lock()`, `omp_test_nest_lock()`, `omp_destroy_nest_lock()`

**Note:** uma thread sempre irá acessar a cópia mais recente do lock, logo, não precisamos fazer o flush na variável do lock.

Um **lock** implica em um **memory fence** (um “flush”) de todas variáveis visíveis as threads

# EXEMPLO DO HISTOGRAMA

Vamos considerar o pedaço de texto abaixo:

“Nobody feels any pain. Tonight as I stand inside the rain”

Vamos supor que queremos contar quantas vezes cada letra aparece no texto.

| A | B | C | D | E | F | G | H | ... | Z |
|---|---|---|---|---|---|---|---|-----|---|
| 5 | 1 | 0 | 3 | 4 | 1 | 1 | 1 |     | 0 |

Para fazer isso em um texto grande, poderíamos dividir o texto entre múltiplas threads. Mas como evitar conflitos durante as atualizações?

# SINCRONIZAÇÃO: LOCKS SIMPLES

Exemplo: são raros os casos de conflito, mas para estarmos seguros, vamos assegurar exclusão mutual para atualizar os elementos do histograma

```
#pragma omp parallel for
for(i=0;i<NBUCKETS; i++){
    omp_init_lock(&hist_locks[i]);
    hist[i] = 0;
}
#pragma omp parallel for
for(i=0;i<NVALS;i++){
    ival = (int) sample(arr[i]);
    omp_set_lock(&hist_locks[ival]);
    hist[ival]++;
    omp_unset_lock(&hist_locks[ival]);
}
for(i=0;i<NBUCKETS; i++)
    omp_destroy_lock(&hist_locks[i]);
```

Um lock por elemento do histograma

Garante exclusão mútua ao atualizar o vetor do histograma

Libera a memória quando termina

# ROTINAS DA BIBLIOTECA DE EXECUÇÃO

Modifica/verifica o número de threads

- `omp_set_num_threads()`, `omp_get_num_threads()`, `omp_get_thread_num()`,  
`omp_get_max_threads()`

Número máximo de threads que eu posso requisitar

Estamos em uma região paralela ativa?

- `omp_in_parallel()`

Queremos que o sistema varie dinamicamente o número de threads de uma construção paralela para outra?

- `omp_set_dynamic()`, `omp_get_dynamic()`

Entra em modo dinâmico

Quantos processadores no sistema?

- `omp_num_procs()`

...mais algumas rotinas menos frequentemente utilizadas.

# ROTINAS DA BIBLIOTECA DE EXECUÇÃO

Como usar um número conhecido de threads em um programa:

- (1) diga ao sistema que não queremos ajuste dinâmico de threads;
- (2) configure o número de threads;
- (3) salve o número de threads que conseguiu;

```
#include <omp.h>
void main()
{
    int num_threads;
    omp_set_dynamic( 0 );
    omp_set_num_threads( omp_num_procs() );
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        #pragma omp single
            num_threads = omp_get_num_threads();
        do_lots_of_stuff(id);
    }
}
```

Desligue o ajuste dinâmico de threads

Peça o número de threads igual ao número de processadores

Proteja essa operação para evitar condições de corrida

Mesmo neste caso, o sistema **poderá te dar menos threads do que requerido**. Se o número exato de threads importa, então teste sempre que necessário.

# ROTINAS DA BIBLIOTECA DE EXECUÇÃO

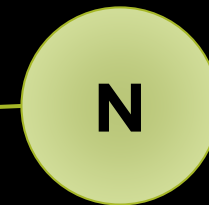
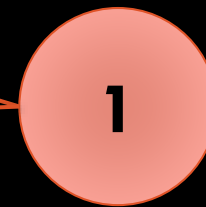
```
#include <omp.h>
void main() {
    int num_threads;
    omp_set_dynamic( 0 );
    omp_set_num_threads( omp_num_procs() );
    num_threads = omp_get_num_threads();
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        #pragma omp single
        num_threads = omp_get_num_threads();
        do_lots_of_stuff(id);
    }
}
```

Quantas threads existem aqui?



# ROTINAS DA BIBLIOTECA DE EXECUÇÃO

```
#include <omp.h>
void main() {
    int num_threads;
    omp_set_dynamic( 0 );
    omp_set_num_threads( omp_num_procs() );
    num_threads = omp_get_num_threads();
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        #pragma omp single
        num_threads = omp_get_num_threads();
        do_lots_of_stuff(id);
    }
}
```



# VARIÁVEIS DE AMBIENTE

Define o número padrão de threads.

- **OMP\_NUM\_THREADS** int\_literal

Controle do tamanho da pilha das threads filhas

- **OMP\_STACKSIZE**

Fornece dicas de como tratar as threads durante espera

- **OMP\_WAIT\_POLICY**

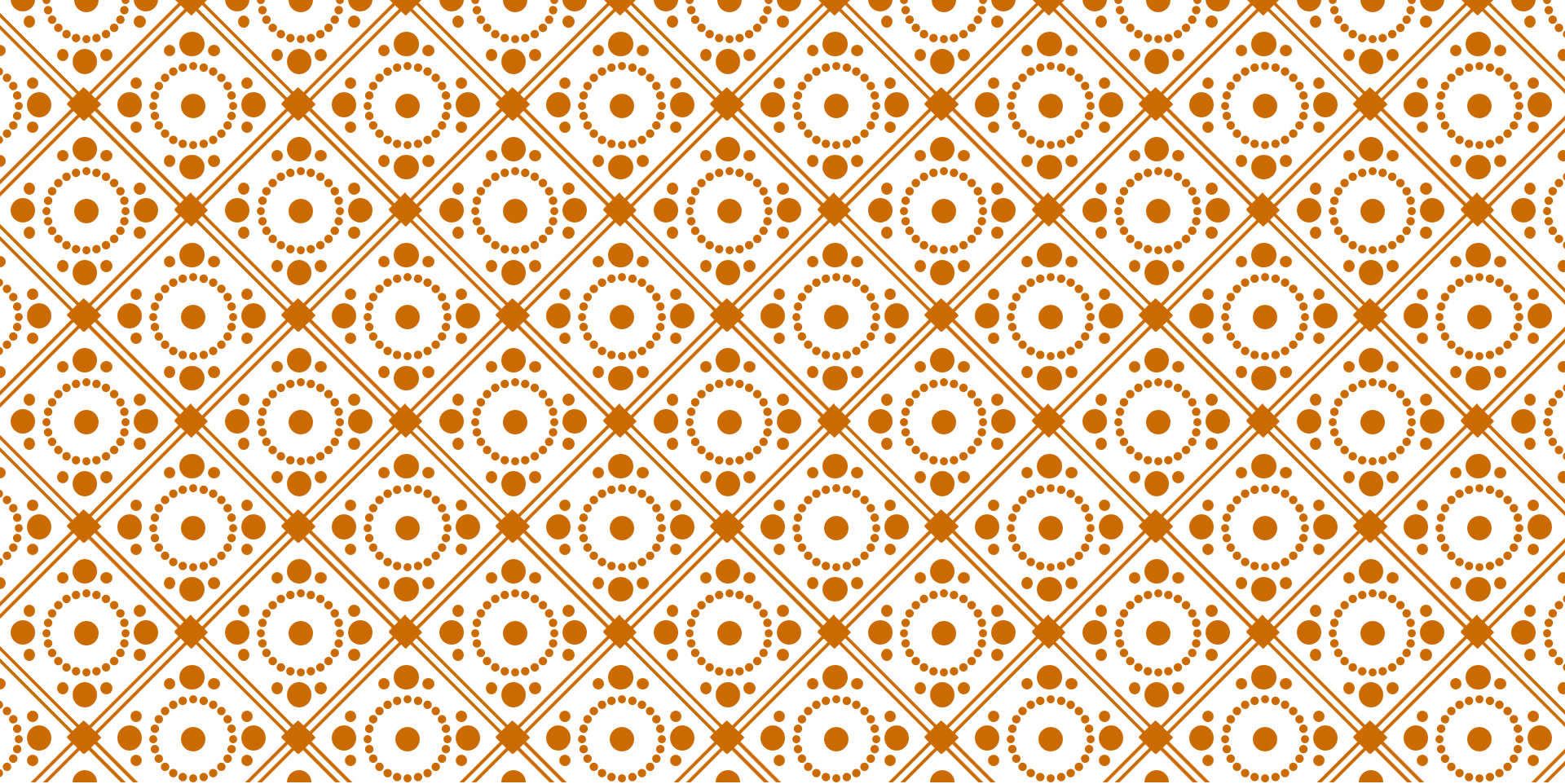
**ACTIVE** mantenha as threads vivas nos barriers/locks (**spin lock**)

**PASSIVE** try to release processor at barriers/locks

Amarra a thread aos processadores. Se estiver TRUE, as threads não irão pular entre processadores.

- **OMP\_PROC\_BIND** true | false

Essas  
variáveis são  
interessantes  
pois não  
precisamos  
recompilar o  
código



# ESCOPO DAS VARIÁVEIS

# ESCOPO PADRÃO DE DADOS

A maioria das variáveis são compartilhadas por padrão

## **Região paralela**

Outside → Global

Inside → Privado

Heap → Global

Stack → Privado

Variáveis globais são compartilhadas entre as threads:

- Variáveis de escopo de arquivo e estáticas
- Variáveis alocadas dinamicamente na memória (malloc, new)

Mas nem tudo é compartilhado:

- Variáveis da pilha de funções chamadas de regiões paralelas são privadas
- Variáveis declaradas dentro de blocos paralelos são privadas

# COMPARTILHAMENTO DE DADOS

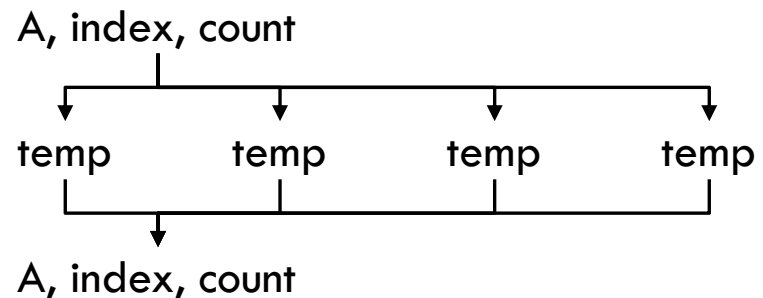
```
double A[10];  
int main() {  
    int index[10];  
    #pragma omp parallel  
    work(index);  
    printf("%d\n", index[0]);  
}
```

```
extern double A[10];  
void work(int *index) {  
    double temp[10];  
    static int count;  
    ...  
}
```

**A**, **index** são compartilhadas entre todas threads.

**temp** é local (privado) para cada thread

**count** também é compartilhada entre as threads



# COMPARTILHAMENTO DE DADOS: MUDANDO OS ATRIBUTOS DE ESCRITA

Podemos mudar seletivamente o compartilhamento de dados usando as devidas diretivas\*

- SHARED
- PRIVATE
- FIRSTPRIVATE

**Todas as diretivas neste slide se aplicam a construção OpenMP e não a região toda.**

O valor final de dentro do laço paralelo pode ser transmitido para uma variável compartilhada fora do laço:

- LASTPRIVATE

Os modos padrão podem ser sobrescritos:

- DEFAULT (SHARED | **NONE**)
- DEFAULT(PRIVATE) is Fortran only

**\*Todas diretivas se aplicam a construções com parallel e de divisão de tarefa, exceto “share” que se aplica apenas a construções parallel.**

# COMPARTILHAMENTO DE DADOS: PRIVATE

`private(var)` cria uma nova variável local para cada thread.

- O valor das variáveis locais novas não são inicializadas
- O valor da variável original não é alterada ao final da região

```
void wrong() {  
    int tmp = 0;  
  
    #pragma omp parallel for private(tmp)  
    for (int j = 0; j < 1000; ++j)  
        tmp += j;  
  
    printf("%d\n", tmp);  
}
```

tmp não foi inicializada

tmp vale 0 aqui

# COMPARTILHAMENTO DE DADOS: PRIVATE ONDE O VALOR ORIGINAL É VALIDO?

O valor da variável original não é especificado se for referenciado fora da construção

As implementações pode referenciar a variável original ou a cópia privada... uma prática de programação perigosa!

- Por exemplo, considere o que poderia acontecer se a função fosse inline?

```
int tmp;  
void danger() {  
    tmp = 0;  
    #pragma omp parallel private(tmp)  
        work();  
  
    printf("%d\n", tmp);  
}
```

```
extern int tmp;  
void work() {  
    tmp = 5;  
}
```

tmp tem um valor não especificado

Não está especificado  
que cópia de tmp  
Privada? Global?

# DIRETIVA FIRSTPRIVATE

As variáveis serão inicializadas com o valor da variável compartilhada

Objetos C++ são construídos por cópia

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
    if ((i%2)==0) incr++;
    A[i] = incr;
}
```

Cada thread obtém sua própria cópia de **incr** com o valor inicial em 0

# DIRETIVA LASTPRIVATE

As variáveis compartilhadas serão atualizadas com o valor da variável que executar a última iteração

Objetos C++ serão atualizado por cópia por padrão

```
void sq2(int n, double *lastterm)
{
    double x; int i;
    #pragma omp parallel for lastprivate(x)
    for (i = 0; i < n; i++){
        x = a[i]*a[i] + b[i]*b[i];
        b[i] = sqrt(x);
    }
    *lastterm = x;
}
```

“x” tem o valor que era mantido nele na última iteração do laço (i.e., for  $i=(n-1)$ )

# COMPARTILHAMENTO DE DADOS: TESTE DE AMBIENTE DAS VARIÁVEIS

Considere esse exemplo de PRIVATE e FIRSTPRIVATE

```
variables: A = 1, B = 1, C = 1  
#pragma omp parallel private(B) firstprivate(C)
```

As variáveis A,B,C são privadas ou compartilhadas dentro da região paralela?

Quais os seus valores iniciais dentro e após a região paralela?

# DATA SHARING: A DATA ENVIRONMENT TEST

Considere esse exemplo de PRIVATE e FIRSTPRIVATE

```
variables: A = 1, B = 1, C = 1  
#pragma omp parallel private(B) firstprivate(C)
```

## Dentro da região paralela...

“A” é compartilhada entre as threads; igual a 1

“B” e “C” são locais para cada thread.

B tem valor inicial não definido

C tem valor inicial igual a 1

## Após a região paralela ...

B e C são revertidos ao seu valor inicial igual a 1

A ou é igual a 1 ou ao valor que foi definido dentro da região paralela

# COMPARTILHAMENTO DE DADOS: A DIRETIVA DEFAULT

Note que o atributo padrão é DEFAULT(SHARED) (logo, não precisamos usar isso)

- Exceção: `#pragma omp task`

Para mudar o padrão: DEFAULT(PRIVATE)

- Cada variável na construção será feita privada como se estivesse sido declaradas como `private(vars)`

DEFAULT(NONE): nenhum padrão será assumido. Deverá ser fornecida uma lista de variáveis privadas e compartilhadas. Boa prática de programação!

**Apenas Fortran suporta `default(private)`.**

**C/C++ possuem apenas `default(shared)` ou `default(none)`.**

# EXERCÍCIO 6: ÁREA DO CONJUNTO MANDELBROT

O programa fornecido (mandel.c) computa uma área do conjunto Mandelbrot.

O programa foi paralelizado com OpenMP, mas fomos preguiçosos e não fizemos isso certo.

**Mas sua versão sequencial funciona corretamente.**

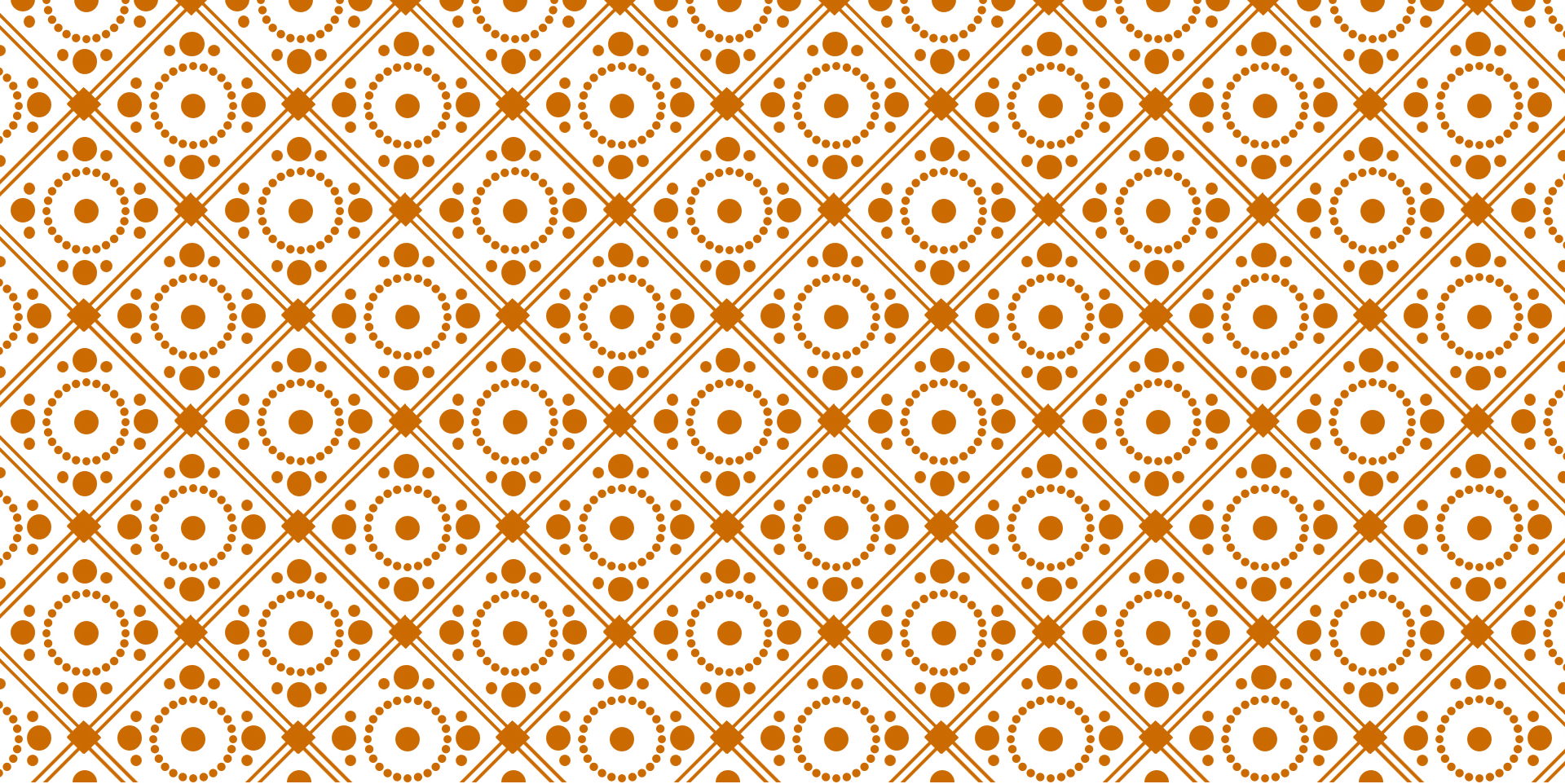
Procure e conserte os erros (dica... o problema está no compartilhamento das variáveis).

<http://www.inf.ufpr.br/mazalves/mandel.c>

# EXERCÍCIO 6 (CONT.)

Ao terminar de consertar, tente otimizar o programa

- Tente diferentes modos de schedule no laço paralelo.
- Tente diferentes mecanismos para suporta exclusão mutua.



# DEBUGANDO PROGRAMAS OPENMP

```

#include <omp.h>
#define NPOINTS 1000
#define MXITR 1000
void testpoint(void);
struct d_complex{
    double r; double i;
};
struct d_complex c;
int numoutside = 0;
int main(){
    int i, j;
    double area, error, eps = 1.0e-5;
    #pragma omp parallel for default(shared) private(c,eps)
    for (i=0; i<NPOINTS; i++) {
        for (j=0; j<NPOINTS; j++) {
            c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
            c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
            testpoint();
        }
    }
    area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-numoutside)/
        (double)(NPOINTS*NPOINTS);
    error=area/(double)NPOINTS;
}

```

```

void testpoint(void){
    struct d_complex z;
    int iter; double temp;
    z=c;
    for (iter=0; iter<MXITR; iter++){
        temp = (z.r*z.r)-(z.i*z.i)+c.r;
        z.i = z.r*z.i*2+c.i;
        z.r = temp;
        if ((z.r*z.r+z.i*z.i)>4.0) {
            numoutside++;
            break;
        }
    }
}

```

Quando executamos esse programa, obtemos uma resposta errada a cada execução ... **existe uma condição de corrida!!!**

# DEBUGANDO PROGRAMAS PARALELOS

Encontre ferramentas que funcionam com seu ambiente e entenda como usa-las. Um bom debugador paralelo pode fazer grande diferença.

Porém, debugadores paralelos não são portáteis, e em algum ponto teremos que debugar “na mão”.

Existem truques que podem nos ajudar. O mais importante é utilizar a diretiva **default(none)**

```
#pragma omp parallel for default(none) private(c, eps)
for (i=0; i<NPOINTS; i++) { // Implicit private
    for (j=0; j<NPOINTS; j++) {
        c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
        c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
        testpoint();
    }
}
```

Ao usar o `default(none)` o compilador gera um erro que J não foi especificado!

# PROGRAMA MANDELBROT

```
#include <omp.h>
#define NPOINTS 1000
#define MXITR 1000
void testpoint(void);
struct d_complex{
    double r; double i;
};
struct d_complex c;
int numoutside = 0;
int main(){
    int i, j;
    double area, error, eps = 1.0e-5;
    #pragma omp parallel for default(shared) private(c,eps)
    for (i=0; i<NPOINTS; i++) {
        for (j=0; j<NPOINTS; j++) {
            c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
            c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
            testpoint();
        }
    }
    area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTSnumoutside)/
        (double)(NPOINTS*NPOINTS);
    error=area/(double)NPOINTS;
}
```

```
void testpoint(void){
    struct d_complex z;
    int iter; double temp;
    z=c;
    for (iter=0; iter<MXITR; iter++){
        temp = (z.r*z.r)-(z.i*z.i)+c.r;
        z.i = z.r*z.i*2+c.i;
        z.r = temp;
        if ((z.r*z.r+z.i*z.i)>4.0) {
            numoutside++;
            break;
        }
    }
}
```

Outros erros achados usando debugador ou por inspeção:

- **eps** não foi inicializado
- Proteja as atualizações ao **numoutside**
- Qual o valor de **c** a função testpoint() vê? Global ou privado?

# SERIAL PI PROGRAM

```
static long num_steps = 100000;
double step;
int main () {
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

Agora que entendemos como mudar o ambiente das variáveis, vamos dar uma última olhada no nosso programa pi.

Qual a menor mudança que podemos fazer para paralelizar esse código?

# EXEMPLO: PROGRAMA PI ...

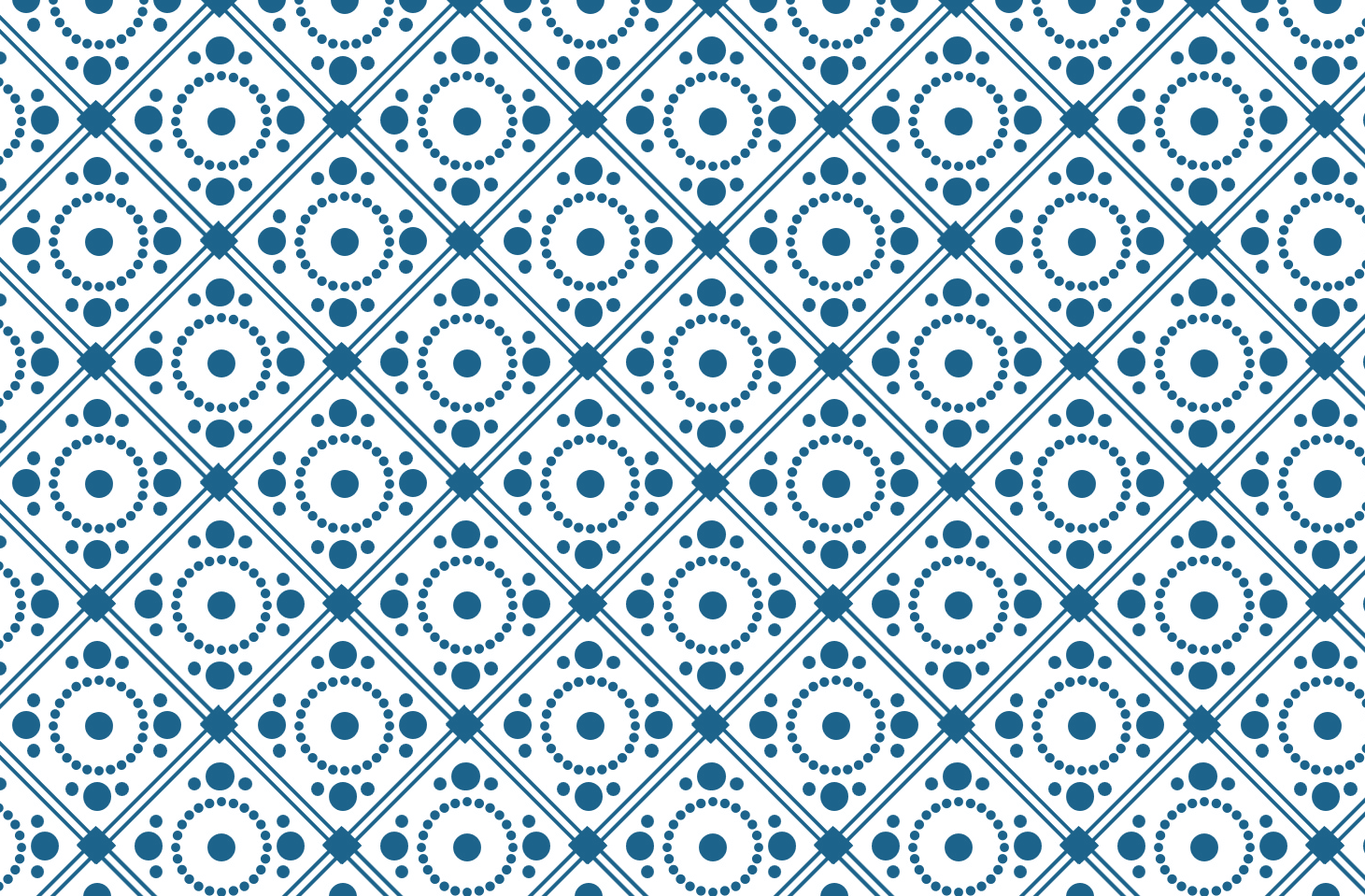
## MUDANÇAS MÍNIMAS

Para boas implementações OpenMP, reduction é mais escalável que o critical.

```
#include <omp.h>
static long num_steps = 100000; double step;
void main (){
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

i é privada por padrão

Note: criamos um programa paralelo sem mudar nenhum código sequencial e apenas incluindo duas linhas de texto!



# INTEL MODERN CODE PARTNER

## OPENMP — AULA 04

# AGENDA GERAL

## Unit 1: Getting started with OpenMP

- Mod1: Introduction to parallel programming
- Mod 2: The boring bits: Using an OpenMP compiler (hello world)
- Disc 1: Hello world and how threads work

## Unit 2: The core features of OpenMP

- Mod 3: Creating Threads (the Pi program)
- Disc 2: The simple Pi program and why it sucks
- Mod 4: Synchronization (Pi program revisited)
- Disc 3: Synchronization overhead and eliminating false sharing
- Mod 5: Parallel Loops (making the Pi program simple)
- Disc 4: Pi program wrap-up

## Unit 3: Working with OpenMP

- Mod 6: Synchronize single masters and stuff
- Mod 7: Data environment
- Disc 5: Debugging OpenMP programs

## Unit 4: a few advanced OpenMP topics

- Mod 8: Skills practice ... linked lists and OpenMP
- Disc 6: Different ways to traverse linked lists
- Mod 8: Tasks (linked lists the easy way)
- Disc 7: Understanding Tasks

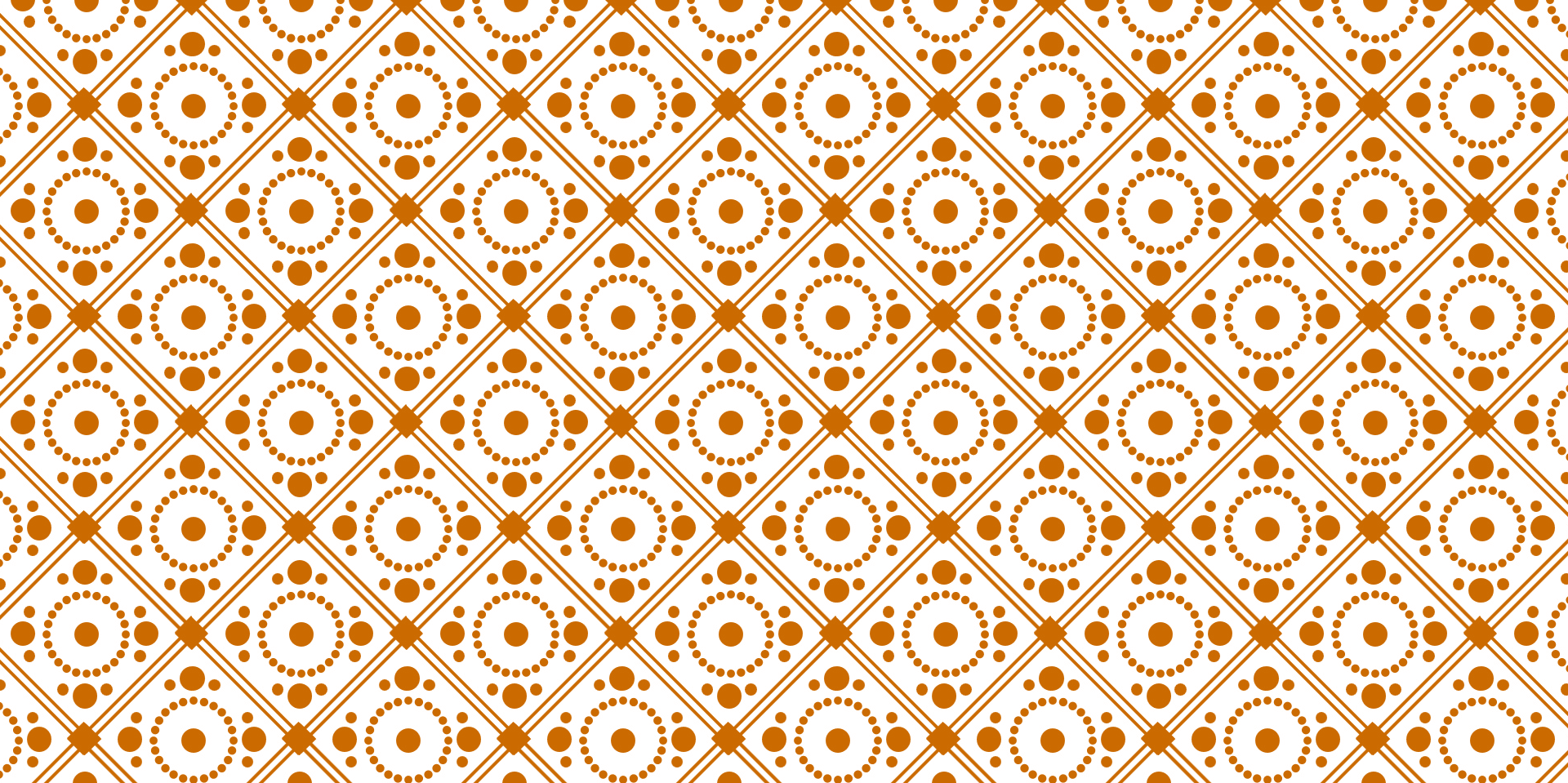
## Unit 5: Recapitulation

- Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
- Disc 8: The pitfalls of pairwise synchronization
- Mod 9: Threadprivate Data and how to support libraries (Pi again)
- Disc 9: Random number generators

# AGENDA DESSA AULA

## **Unit 4: a few advanced OpenMP topics**

- Mod 8: Skills practice ... linked lists and OpenMP
- Disc 6: Different ways to traverse linked lists
- Mod 8: Tasks (linked lists the easy way)
- Disc 7: Understanding Tasks



# PRÁTICA DE CONHECIMENTOS...

## LISTA ENCADEADA E OPENMP

# AS PRINCIPAIS CONSTRUÇÕES OPENMP VISTAS ATÉ AGORA

Para criar um time de threads

- `#pragma omp parallel`

Ao imprimir o valor da macro `_OPENMP`  
Teremos um valor `yyyymm` (ano e mês) da  
implementação OpenMP usada

Para compartilhar o trabalho entre as threads

- `#pragma omp for`
- `#pragma omp single`

Para prevenir conflitos (previne corridas)

- `#pragma omp critical`
- `#pragma omp atomic`
- `#pragma omp barrier`
- `#pragma omp master`

Diretivas de ambiente de variáveis

- `private (variable_list)`
- `firstprivate (variable_list)`
- `lastprivate (variable_list)`
- `reduction(+:variable_list)`

Onde **variable\_list** é uma  
lista de variáveis separadas  
por vírgula

# CONSIDERE UMA SIMPLES LISTA ENCADEADA

Considerando o que vimos até agora em OpenMP, como podemos processar esse laço em paralelo?

```
p=head;
while (p) {
    process(p);
    p = p->next;
}
```

Lembre-se, a construção de divisão de trabalho funciona apenas para laços onde o número de repetições do laço possa ser representado de uma forma fechada pelo compilador.

Além disso, laços do tipo **while** não são cobertos.

# EXERCÍCIO 7: LISTA ENCADEADA (MODO DIFÍCIL)

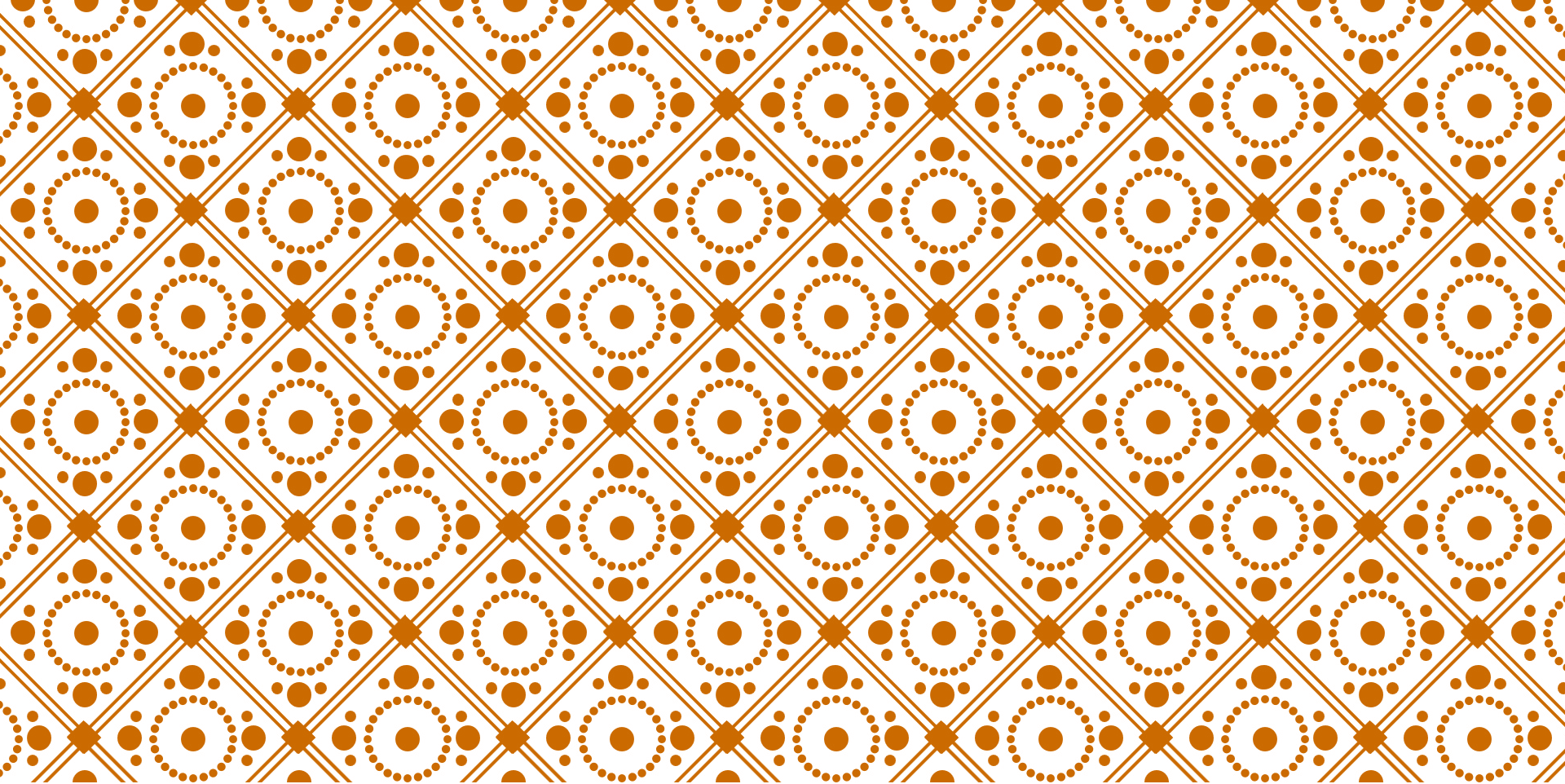
Considere o programa `linked.c`

- Atravessa uma lista encadeada computando uma sequência de números Fibonacci para cada nó.

Paralelize esse programa usando as construções vistas até agora (ou seja, mesmo que saiba, **não use tasks**).

Quando tiver um programa correto, otimize ele.

<http://www.inf.ufpr.br/mazalves/linked.c>



# DIFERENTES MANEIRAS DE PERCORRER LISTAS ENCADEADAS

# PERCORRENDO UMA LISTA

Quando OpenMP foi criado, o foco principal eram os casos frequentes em HPC ... vetores processados com laços “regulares”.

**Recursão e “pointer chasing” foram removidos do foco de OpenMP.**

Assim, mesmo um simples passeio por uma lista encadeada é bastante difícil nas versões originais de OpenMP

```
p=head;
while (p) {
    process(p);
    p = p->next;
}
```

# LISTA ENCADEADA SEM TASKS (HORRÍVEL)

```
while (p != NULL) {  
    p = p->next;  
    count++;  
}  
p = head;  
for(i=0; i<count; i++) {  
    parr[i] = p;  
    p = p->next;  
}  
#pragma omp parallel  
{  
    #pragma omp for schedule(static,1)  
    for(i=0; i<count; i++)  
        processwork(parr[i]);  
}
```

Conta o número de itens na lista encadeada

Copia o ponteiro para cada nó em um vetor

Processa os nós em paralelo

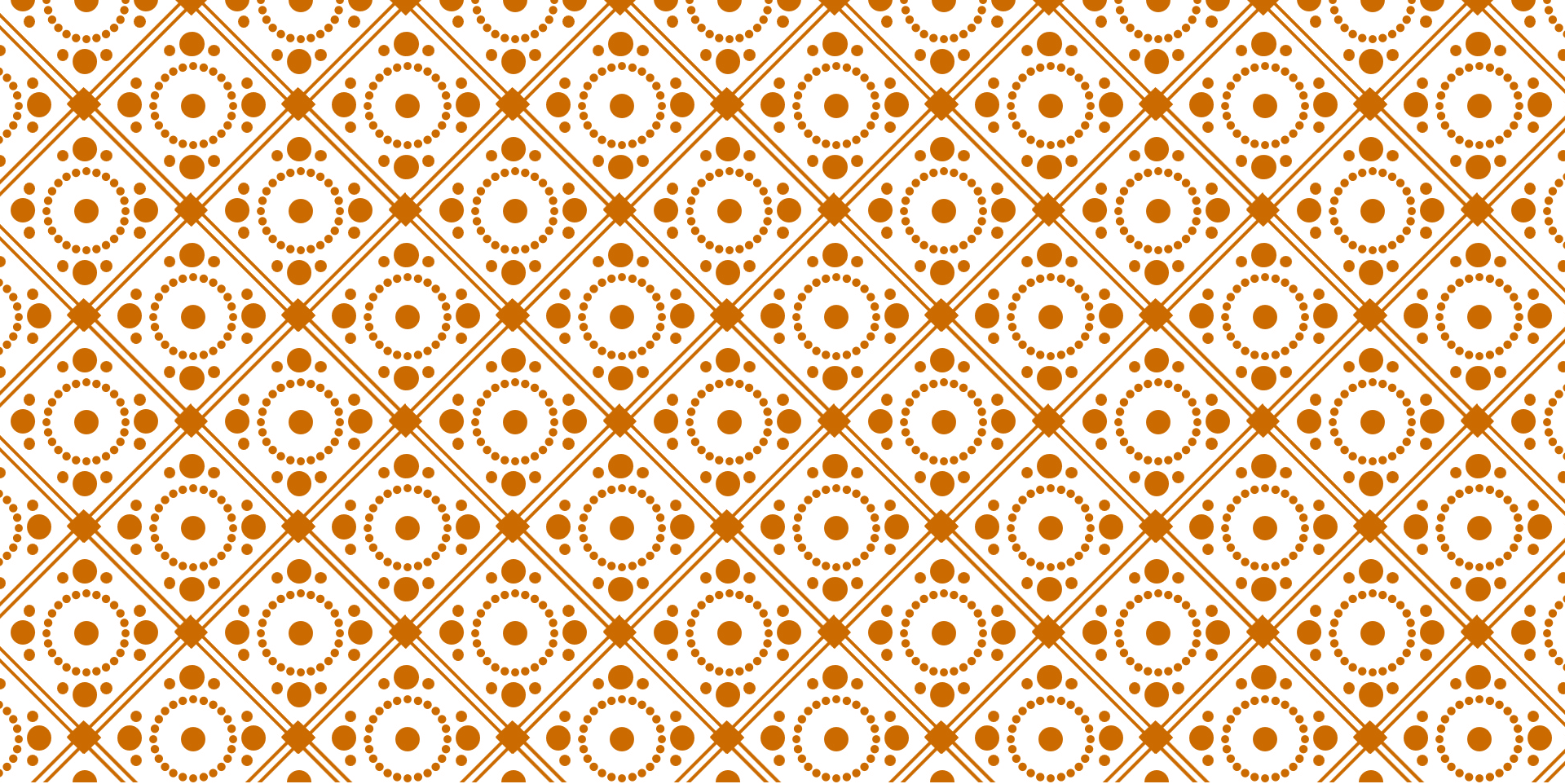
|           | Schedule padrão | Static, 1 |
|-----------|-----------------|-----------|
| 1 Thread  | 48 sec          | 45 sec    |
| 2 Threads | 39 sec          | 28 sec    |

# CONCLUSÕES

Somos capazes de paralelizar listas encadeadas ... mas isso foi feio, e requereu múltiplas passadas sobre os dados.

Para ir além do mundo baseado em vetores, precisamos suportar estruturas de dados e laços além dos tipos básicos.

Por isso, foram adicionadas **tasks** no **OpenMP 3.0**



# TASKS (SIMPLIFICANDO AS LISTAS ENCADEADAS)

# OPENMP TASKS

Tasks são unidades de trabalho independentes.

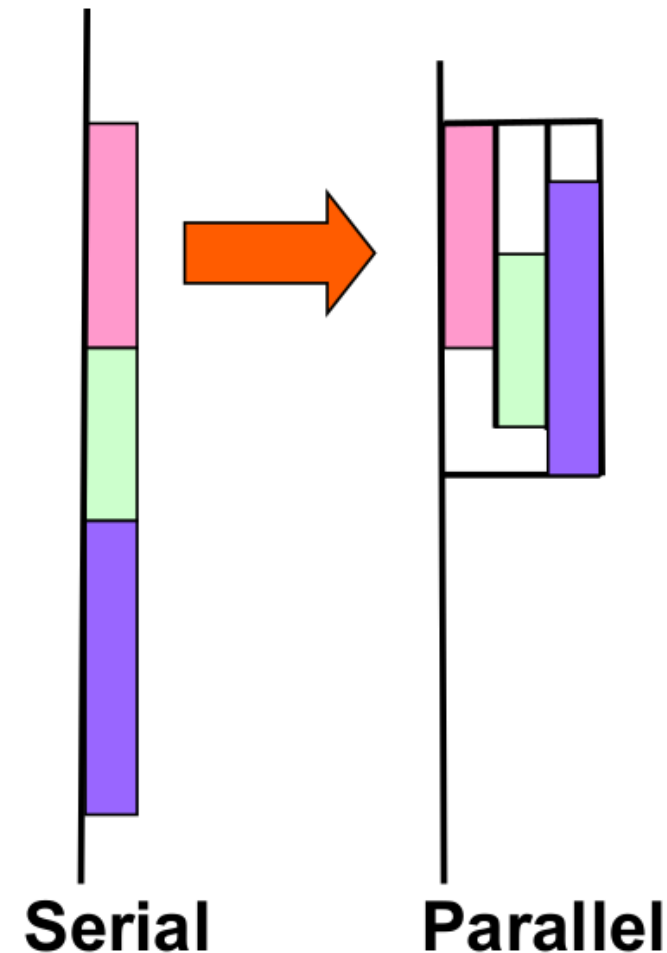
Tasks são compostas de:

- Código para executar
- Dados do ambiente
- Variáveis de controle interno (ICV)

As threads executam o trabalho de cada task.

O sistema de execução decide quando as tasks serão executadas

- As tasks podem ser atrasadas
- As Tasks podem ser executadas imediatamente



# QUANDO PODEMOS GARANTIR QUE AS TASKS ESTARÃO PRONTAS?

As tasks estarão completadas na barreira das threads:

- `#pragma omp barrier`

Ou barreira de tasks

- `#pragma omp taskwait`

```
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

Múltiplas **tasks foo** são criadas aqui. Uma por thread.

Todas **tasks foo** estarão completadas aqui

Uma **task bar** foi criada aqui

A **task bar** estará completa aqui (barreira implícita)

# ESCOPO DE VARIÁVEIS COM TASKS: EXEMPLO FIBONACCI.

Exemplo de divisão e conquista

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

n é privada para ambas tasks

x é uma variável privada da thread  
y é uma variável privada da thread

○ que está errado aqui?

# ESCOPO DE VARIÁVEIS COM TASKS: EXEMPLO FIBONACCI.

Exemplo de divisão e conquista

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

n é privada para ambas tasks

x é uma variável privada da thread  
y é uma variável privada da thread

O que está errado aqui?

As variáveis se tornaram privadas das tasks e não vão estar disponíveis fora das tasks

# ESCOPO DE VARIÁVEIS COM TASKS: EXEMPLO FIBONACCI.

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

n é privada para ambas tasks

x & y serão compartilhados  
**Boa solução**  
pois precisamos de ambos para  
computar a soma

# ESCOPO DE VARIÁVEIS COM TASKS: EXEMPLO LISTA ENCADEADA.

```
List m1; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
    for(e=m1->first;e;e=e->next)
        #pragma omp task
        process(e);
}
```

○ que está errado aqui?

# ESCOPO DE VARIÁVEIS COM TASKS: EXEMPLO LISTA ENCADEADA.

```
List m1; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
    for(e=m1->first;e;e=e->next)
        #pragma omp task
        process(e);
}
```

O que está errado aqui?

Possível condição de corrida!  
A variável compartilhada “e”  
poderá ser atualizada por múltiplas  
tasks

# ESCOPO DE VARIÁVEIS COM TASKS: EXEMPLO LISTA ENCADEADA.

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
for(e=ml->first;e;e=e->next)
    #pragma omp task firstprivate(e)
    process(e);
}
```

Boa solução  
e será first private

# REGRAS DE ESCOPO DE VARIÁVEIS (OPENMP 3.0 SPECS.)

Variáveis **static** declarada na rotina chamada na task serão **compartilhadas**, a menos que sejam utilizadas as primitivas de *private* da thread.

Variáveis do tipo **const** não tendo membros mutáveis, e declarado nas rotinas chamadas, serão **compartilhadas**.

**Escopo de arquivo** ou **variáveis no escopo de namespaces** referenciadas nas rotinas chamadas são **compartilhadas**, a menos que sejam utilizadas as primitivas de *private* da thread.

Variáveis alocadas no **heap**, serão **compartilhadas**.

**Demais variáveis** declaradas nas rotinas chamadas **serão privadas**.

# REGRAS DE ESCOPO DE VARIÁVEIS

As regras de padronização de escopo são implícitas e podem nem sempre ser óbvias.

Para evitar qualquer surpresa, é sempre recomendado que o programador diga explicitamente o escopo de todas as variáveis que são referenciadas dentro da task usando as diretivas *private*, *shared*, *firstprivate*.

# EXERCÍCIO 8: TASKS EM OPENMP

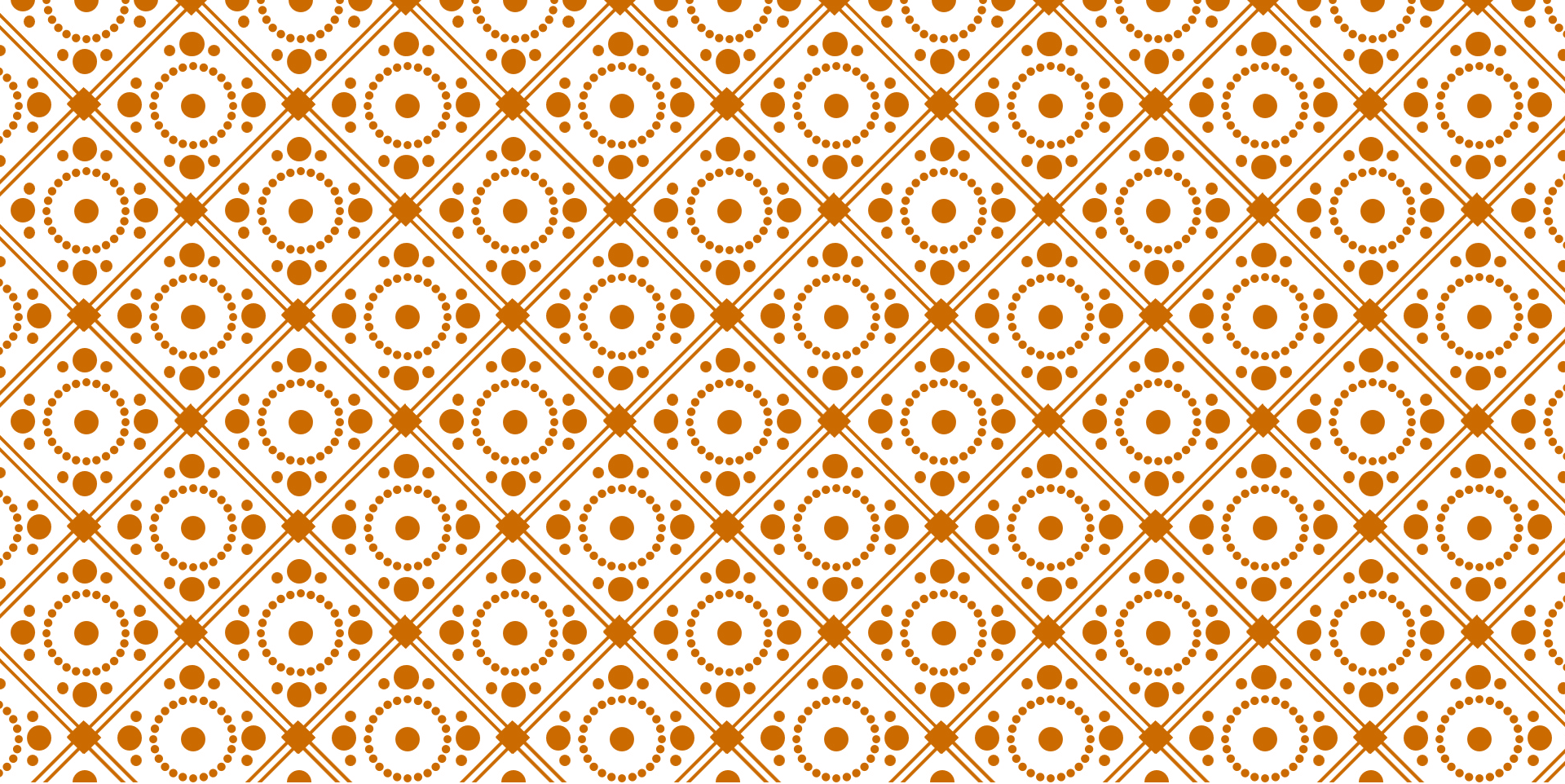
Considere o programa `linked.c`

- Atravessa uma lista encadeada computando uma sequência de números Fibonacci para cada nó.

Paralelize esse programa usando **tasks**.

Compare a solução obtida com a versão sem `tasks`.

<http://www.inf.ufpr.br/mazalves/linked.c>



# ENTENDENDO TASKS

# CONSTRUÇÕES TASK – TASKS EXPLÍCITAS

```
#pragma omp parallel
{
    #pragma omp single
    {
        node *p = head;
        while (p) {
            #pragma omp task firstprivate(p)
            process(p);
            p = p->next;
        }
    }
}
```

1. Cria um time de threads

2. Uma thread executa a construção single ... as demais threads vão aguardar na barreira implícita ao final da construção single

3. A thread “single” cria a task com o seu próprio valor de ponteiro p

4. As threads aguardando na barreira executam as tasks.

A execução move além da barreira assim que todas as tasks estão completas

# EXECUÇÃO DE TASKS

Possui potencial para paralelizar padrões irregulares e chamadas de funções recursivas.

```
#pragma omp parallel
{
    #pragma omp single
    {
        //block 1
        node * p = head;
        while (p) {
            // block 2
            #pragma omp task
            process(p);
        }
        //block 3
        p = p->next;
    }
}
```

Threads

Única

block1

Block2  
task1

block3

Block2  
task2

block3

Block2  
task3

Th 1

block1

block3

block3

Th 2

Block2  
task1

Th 3

Block2  
task2

Th 4

Block2  
task3

Tempo  
economizado

<http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf>

INTEL MODERN CODE PARTNER 178