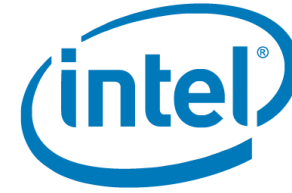


# INTEL MODERN CODE PARTNER UFPEL



# EQUIPE



Matheus S. Serpa

Marco A. Zanata Alves

Vinícius Garcia Pinto

**Demais membros do GPPD.**



# TESTANDO O AMBIENTE REMOTO

Login remoto a sistemas de computadores

```
ssh ufpelintel@term1
```

```
senha: ufpelintel
```

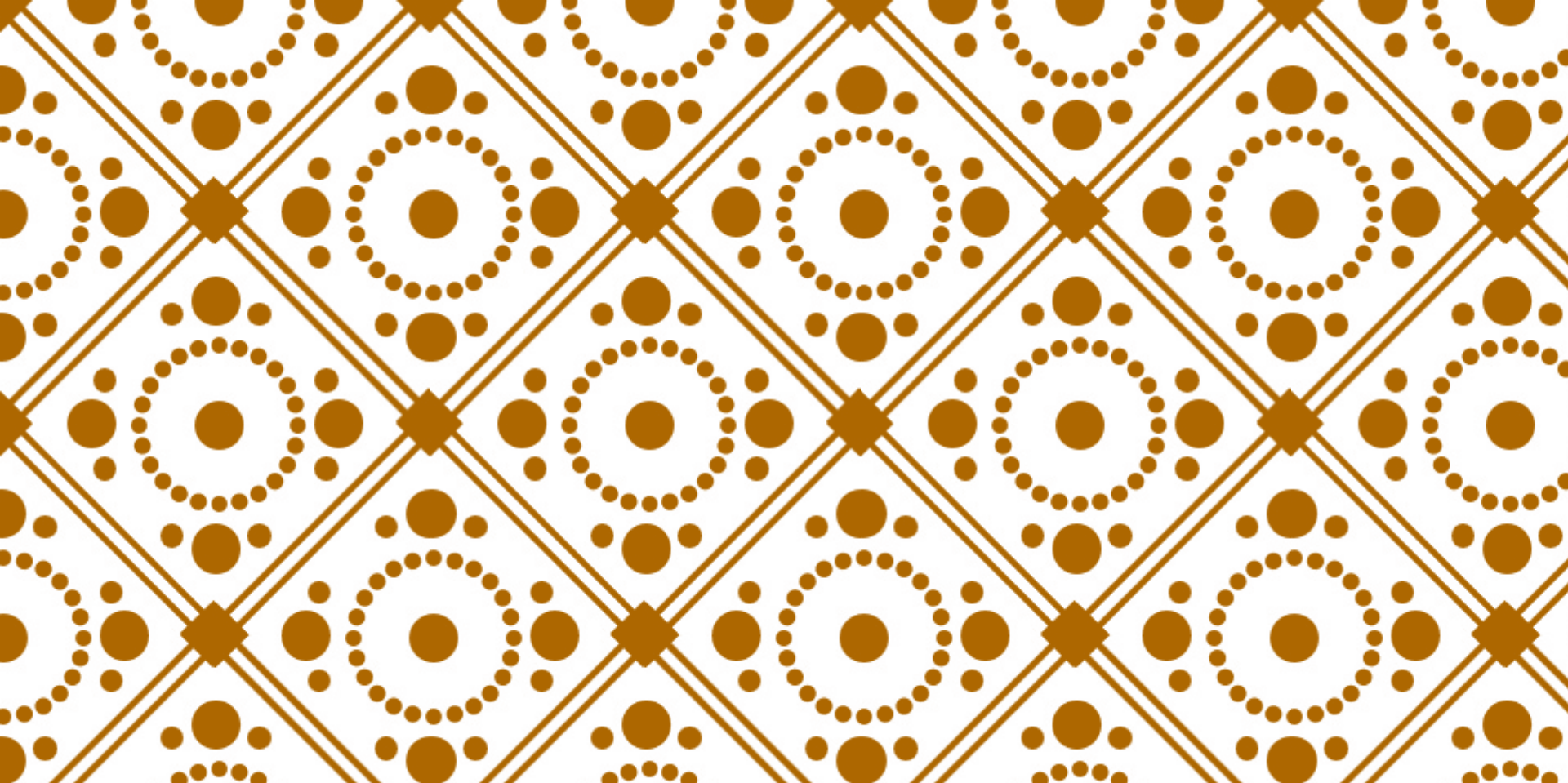
**conecta**

Copie os exercícios

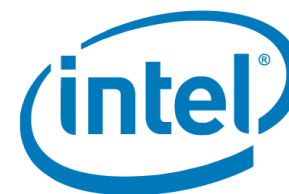
```
cp -r ufrgs-intel-modern-code/ seu-nome/
```

```
cd seu-nome/
```

Trabalhe dentro de seu diretório



# PROGRAMAÇÃO PARALELA

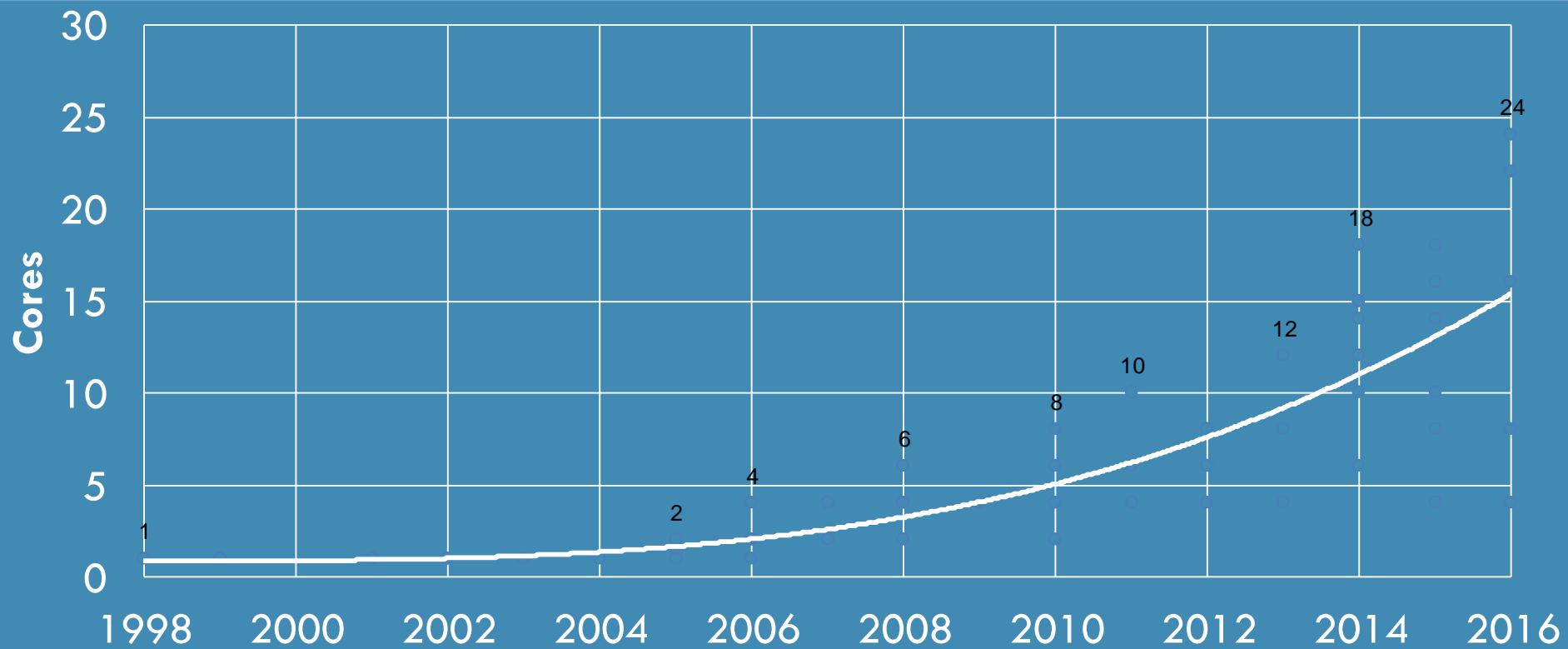


# POR QUE ESTUDAR PROGRAMAÇÃO PARALELA?

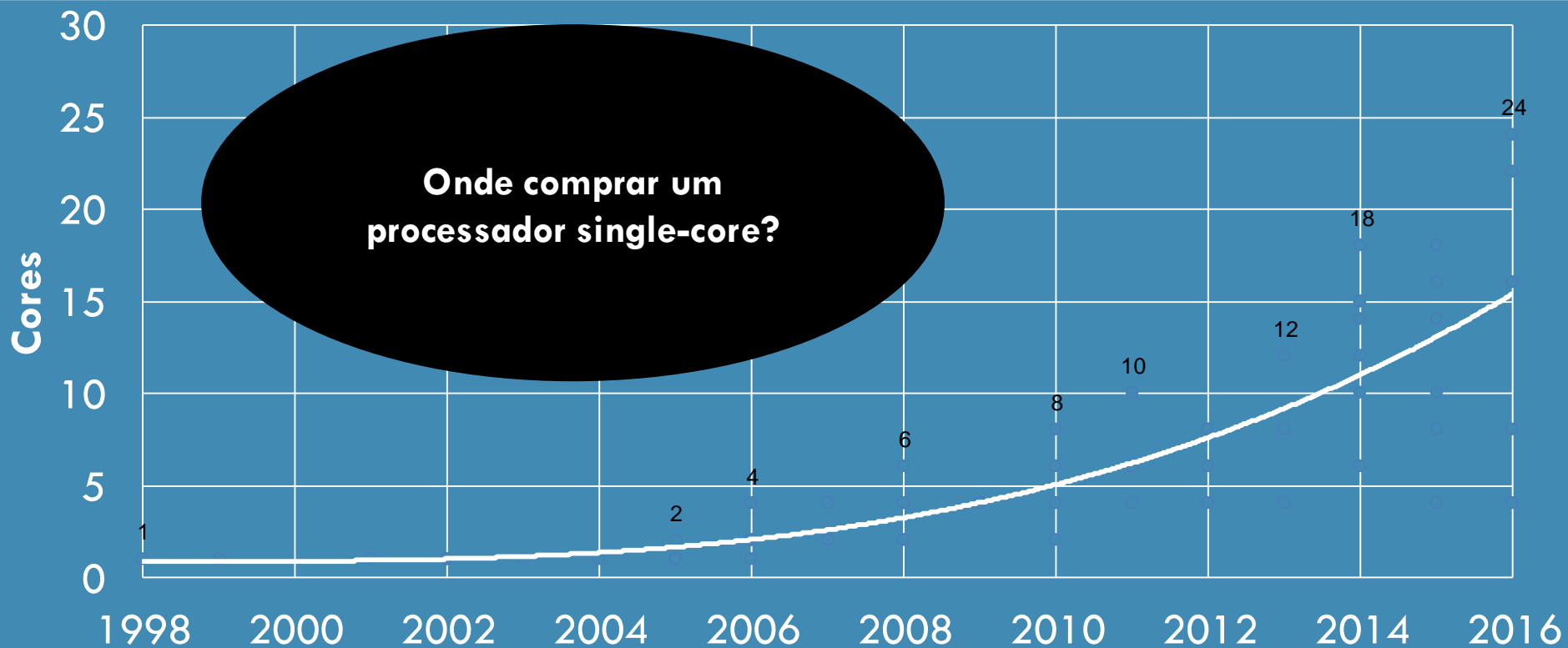
Os programas já não são rápidos o suficiente?

As máquinas já não são rápidas o suficiente?

# EVOLUÇÃO DO INTEL XEON



# EVOLUÇÃO DO INTEL XEON



# PORQUÊ PROGRAMAÇÃO PARALELA?

Dois dos principais motivos para utilizar programação paralela são:

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

Outros motivos são:

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



# COMO FAZER ATIVIDADES EM PARALELO? NO MUNDO REAL

Você pode enviar cartas/mensagens aos amigos e pedir ajuda

Mais amigos,  
mais tempo  
para eles  
chegarem/ se  
acomodarem

Você pode criar uma lista de tarefas (pool)

Tarefas  
curtas ou  
longas?

Quando um amigo ficar atoa, pegue uma nova tarefa da lista

Muitos  
amigos  
olhando e  
riscando a  
lista?

Você pode ajudar na tarefa ou então ficar apenas gerenciando

Precisa  
gerenciar  
algo mais?

# OPÇÕES PARA CIENTISTAS DA COMPUTAÇÃO

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

# PRINCIPAIS MODELOS DE PROGRAMAÇÃO PARALELA



## Programação em Memória Compartilhada (OpenMP, Cilk)

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

## Programação em Memória Distribuída (MPI)

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

# COMO IREMOS PARALELIZAR? **PENSANDO!**

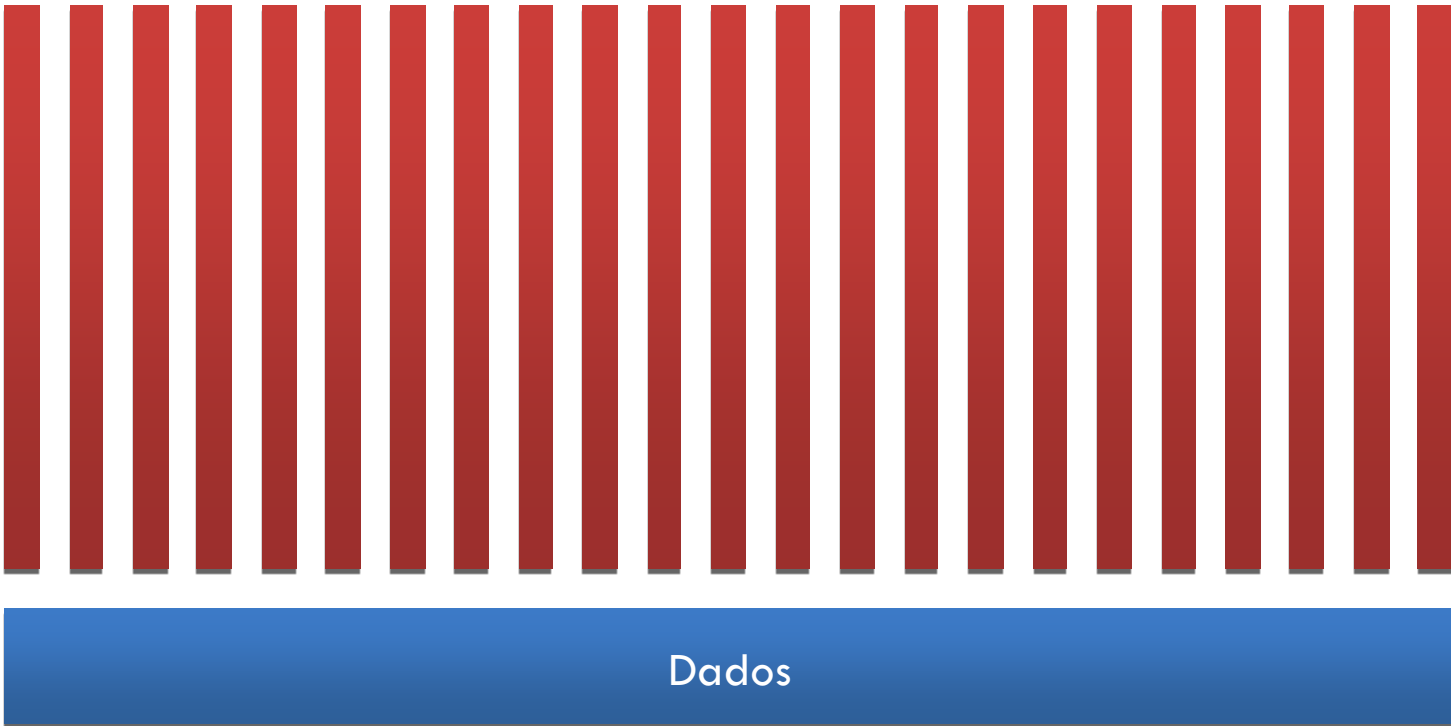


Trabalho

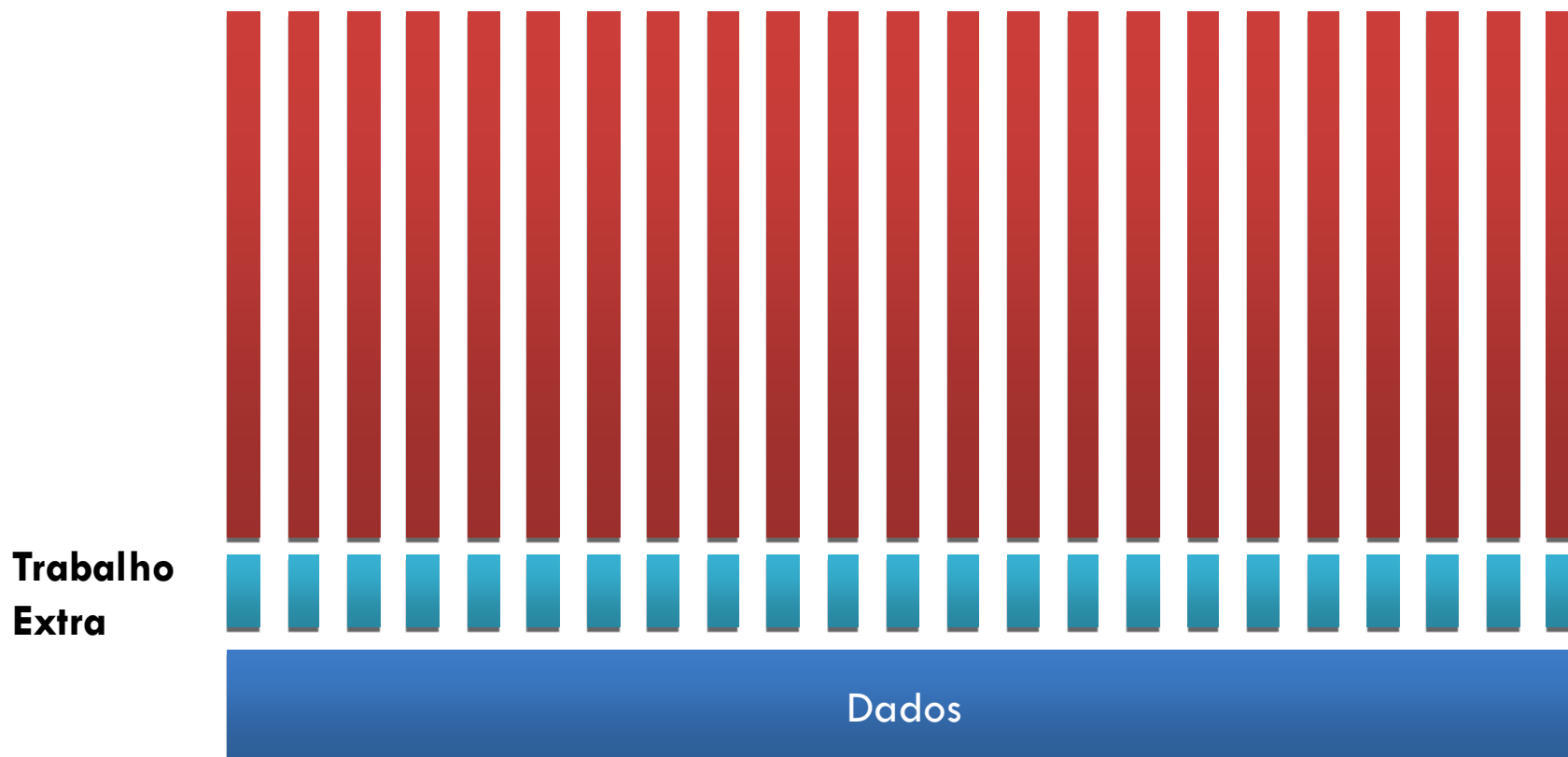
The diagram consists of two stacked rectangular blocks. The top block is red and contains the word 'Trabalho' in white text. The bottom block is blue and contains the word 'Dados' in white text.

Dados

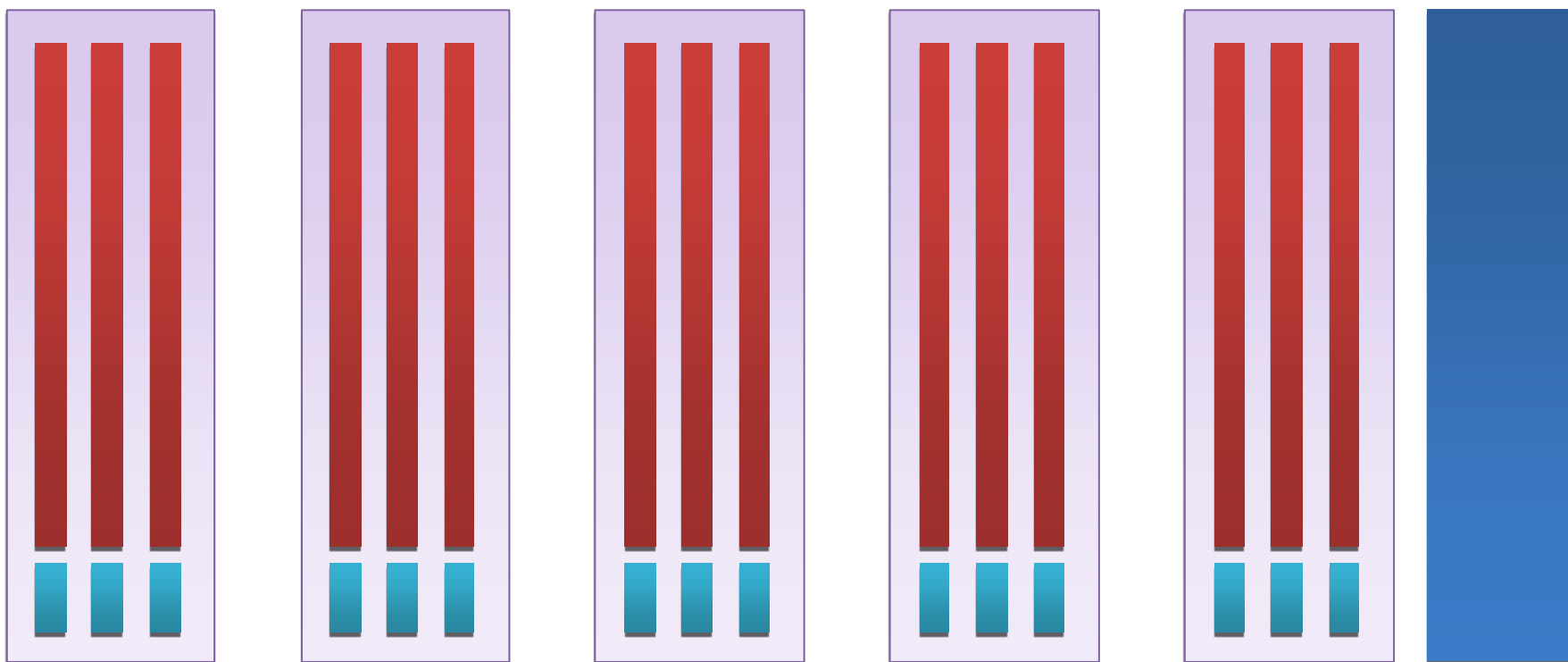
# COMO IREMOS PARALELIZAR? PENSANDO!



# COMO IREMOS PARALELIZAR? PENSANDO!



# COMO IREMOS PARALELIZAR? PENSANDO!



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

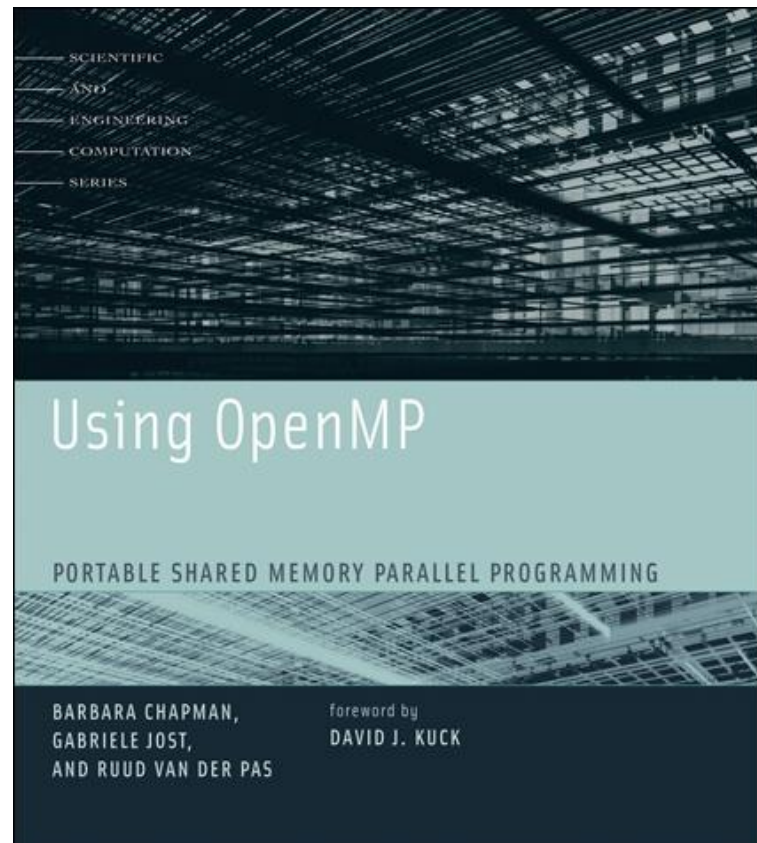
# BIBLIOGRAFIA BÁSICA

## **Using OpenMP - Portable Shared Memory Parallel Programming**

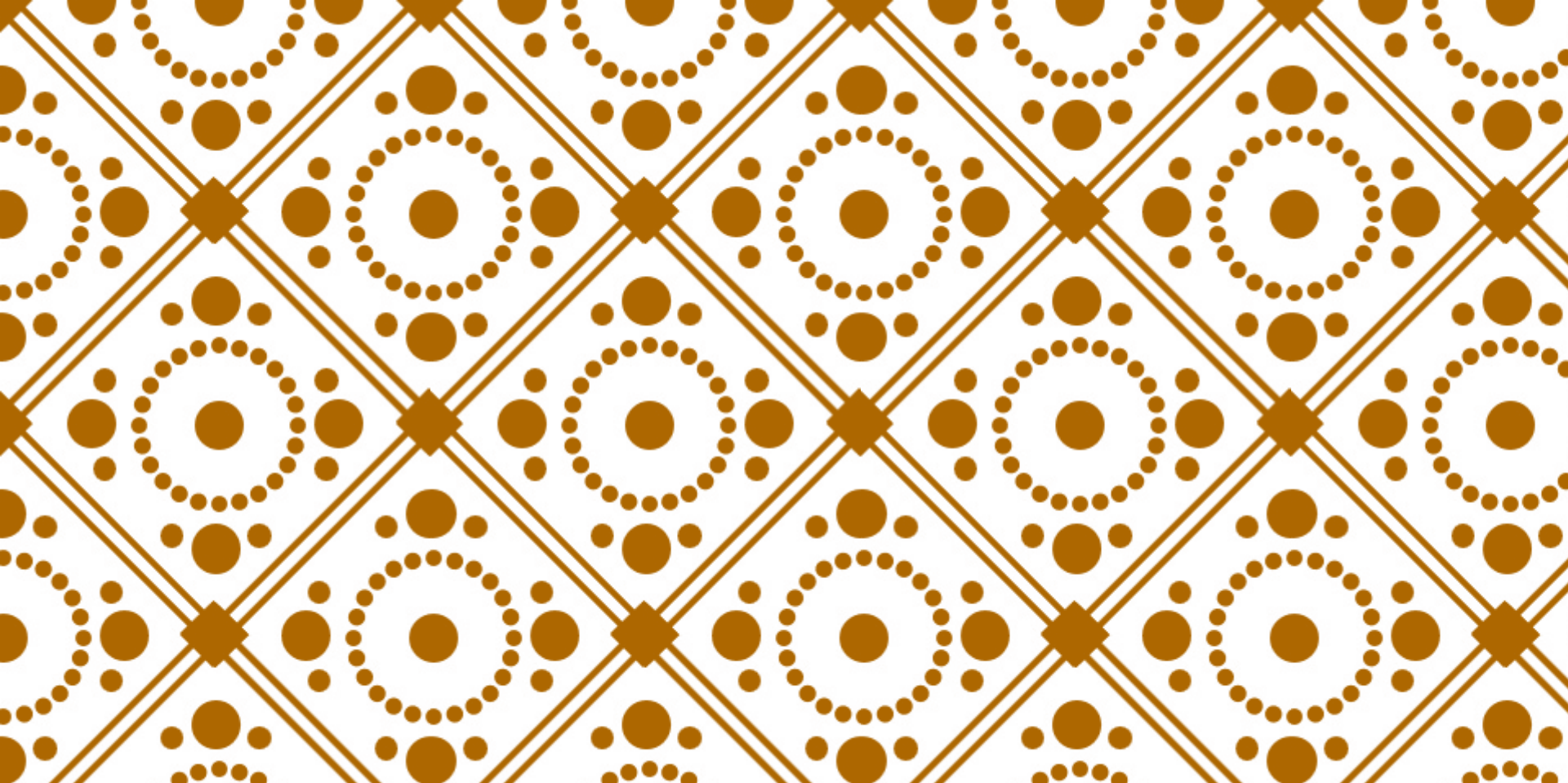
**Autores:** Barbara Chapman,  
Gabriele Jost and Ruud van der  
Pas

**Editora:** MIT Press

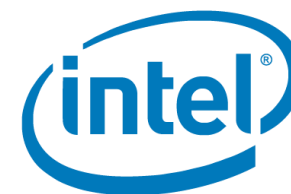
**Ano:** 2007







OPENMP



# INTRODUÇÃO

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

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

## Observações:

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

# VISÃO GERAL OPENMP:

OpenMP: Uma API para escrever aplicações Multithreaded

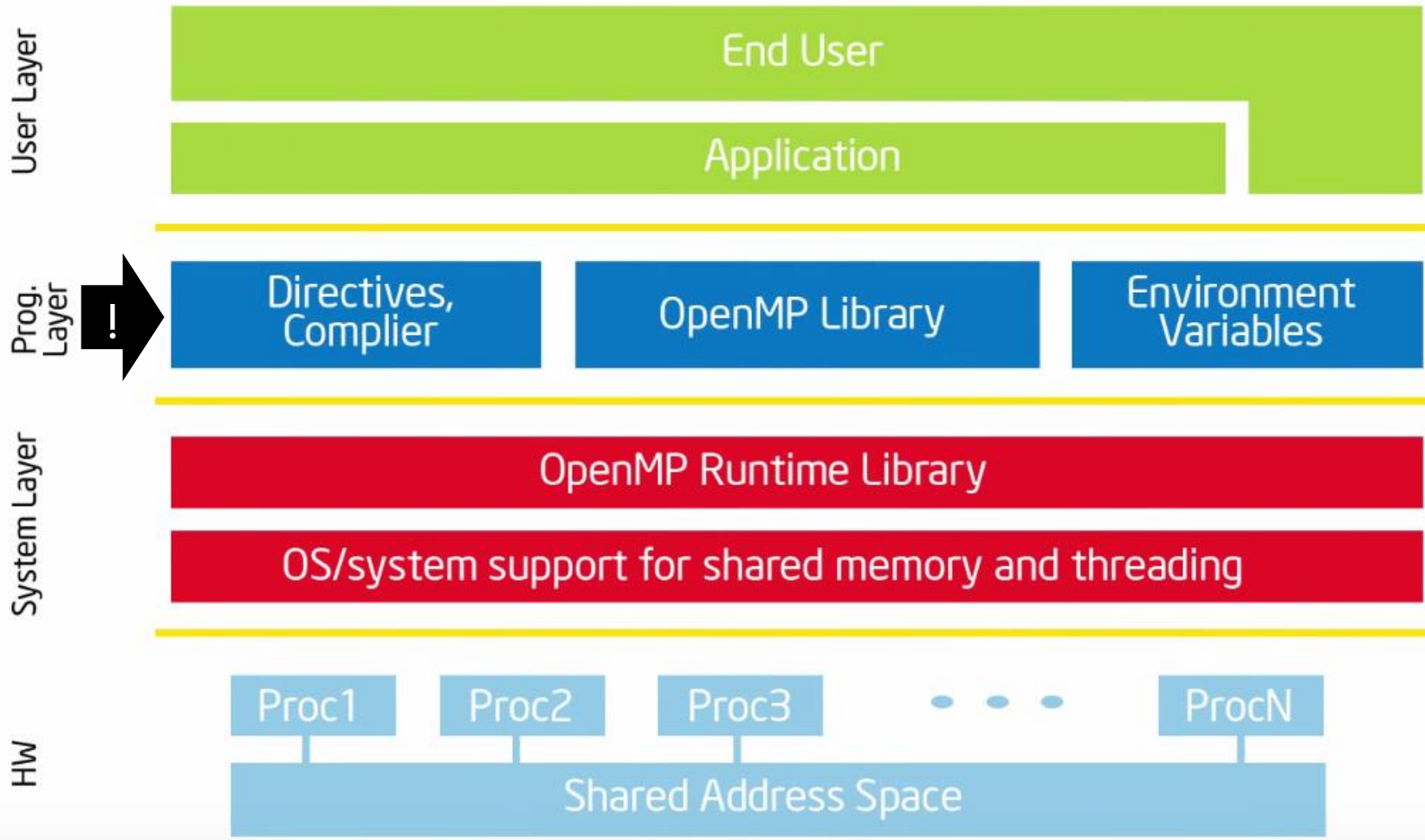
Um conjunto de **diretivas do compilador** e **biblioteca de rotinas** para programadores de aplicações paralelas

+ variáveis de ambiente

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 private(var1, var2) shared(var3,  
var4)
```

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

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

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

# NOTAS DE COMPILAÇÃO

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

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

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

```
export OMP_NUM_THREADS=40
```

```
./a.out
```

Para shell bash

Por padrão é o nº de  
proc. virtuais.

Também  
funciona no  
Windows!

Até mesmo  
no Visual  
Studio!

**Mas vamos  
usar Linux**



# FUNÇÕES

Funções da biblioteca OpenMP.

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

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

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

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

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

# DIRETIVAS

Diretivas do OpenMP.

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

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

}
```



# EXEMPLO, PARTE A

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);

}
```

# EXEMPLO, PARTE B

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);           // Queremos escrever o
        printf(" world(%d) \n", ID);         // id de cada thread
    }
}
```

# EXEMPLO, PARTE C

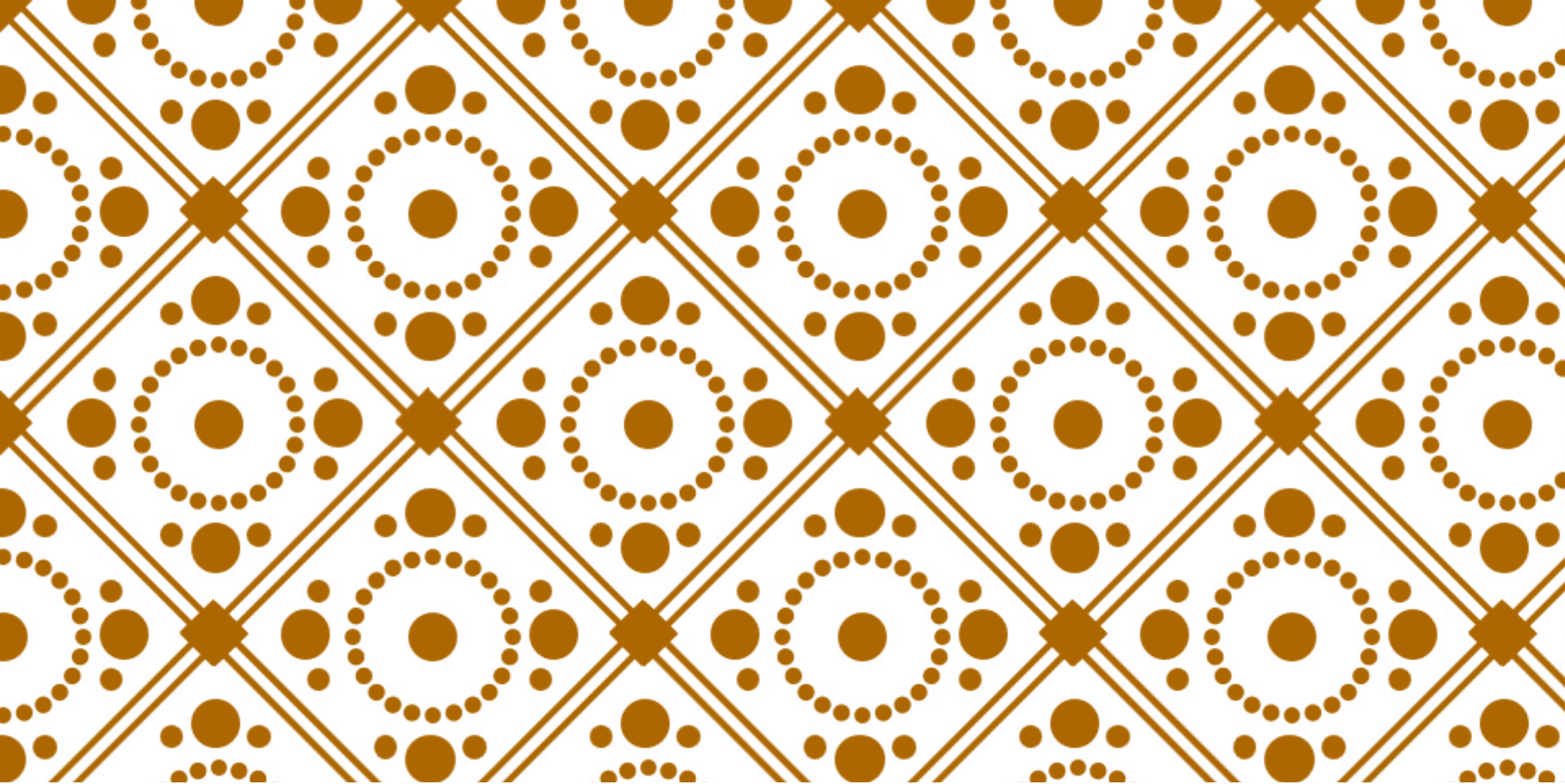
Verifique se seu ambiente funciona

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

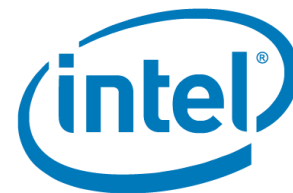
```
#include <stdio.h>
#include <omp.h>

int main() {

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



# HELLO WORLD E COMO AS THREADS FUNCIONAM



# EXEMPLO, 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

# EXEMPLO, SOLUÇÃO

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

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

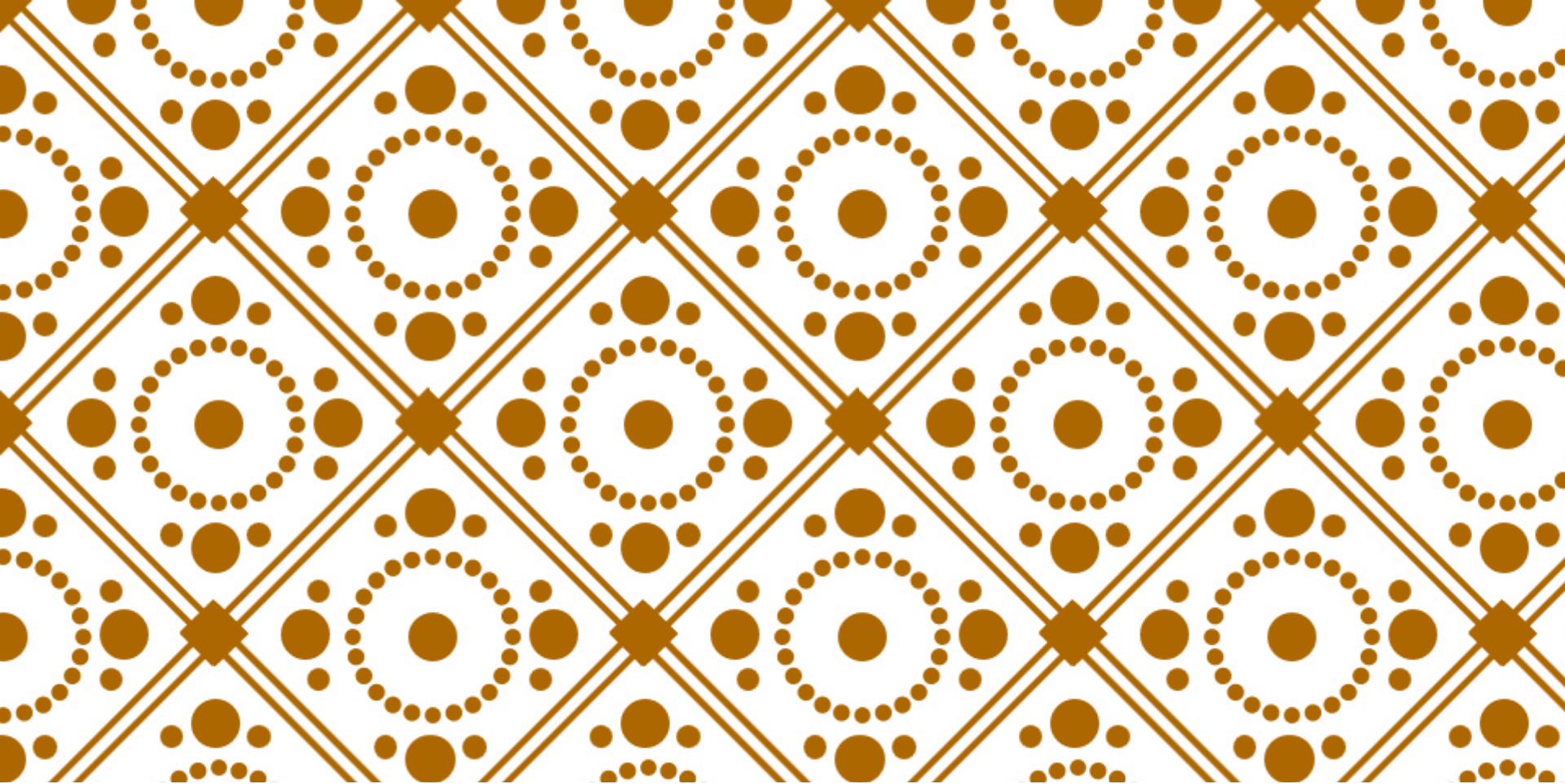
Como foi a saída  
do programa?  
Organizada?

Arquivo OpenMP

Região paralela com um  
número padrão de threads

Função da biblioteca que  
retorna o thread ID.

Fim da região paralela



# FUNCIONAMENTO DOS PROCESSOS VS. THREADS

# MULTIPROCESSADORES MEMÓRIA COMPARTILHADA

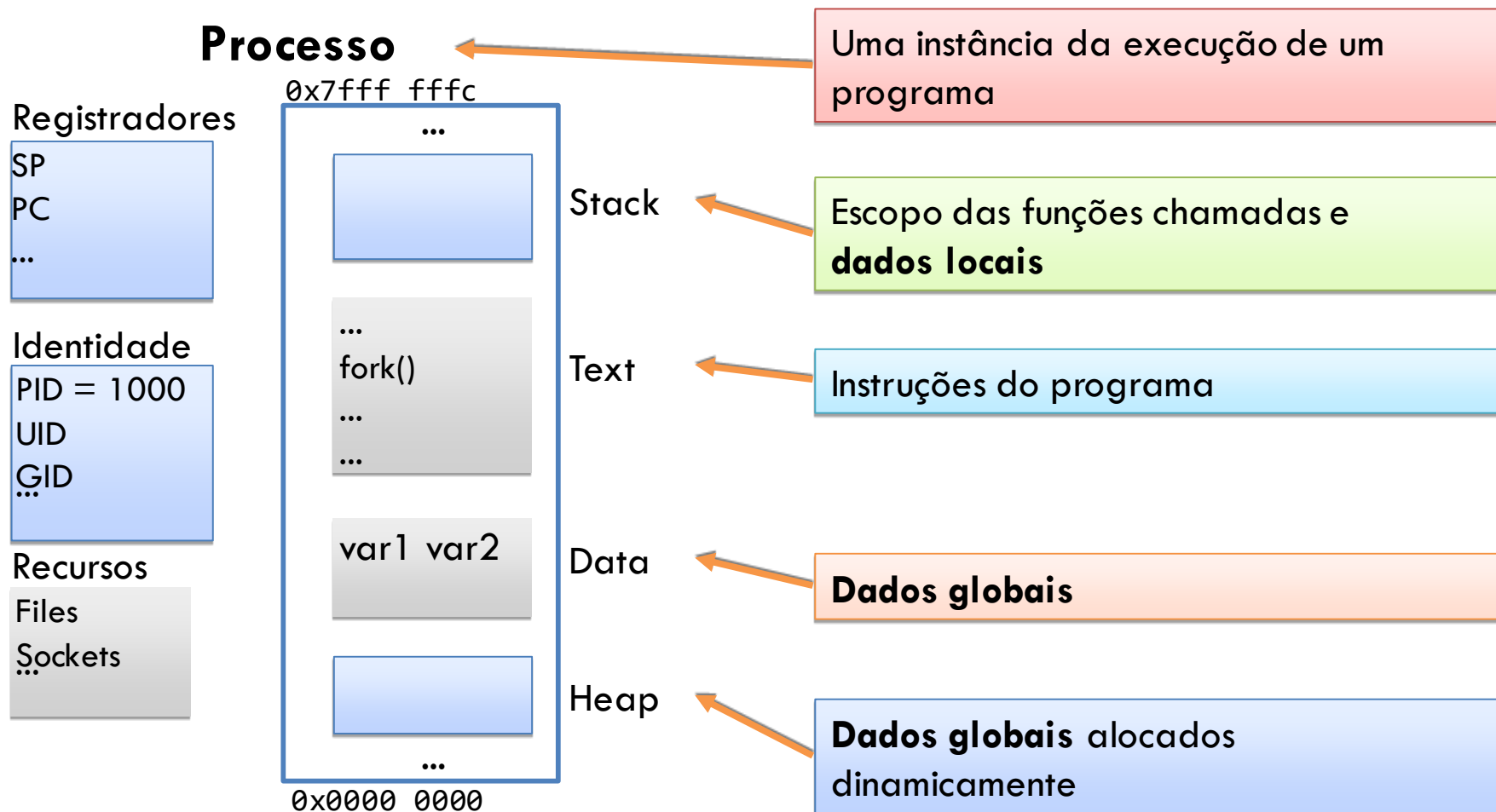
Um multiprocessadores é um computador em que todos os processadores partilham o acesso à memória física.

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

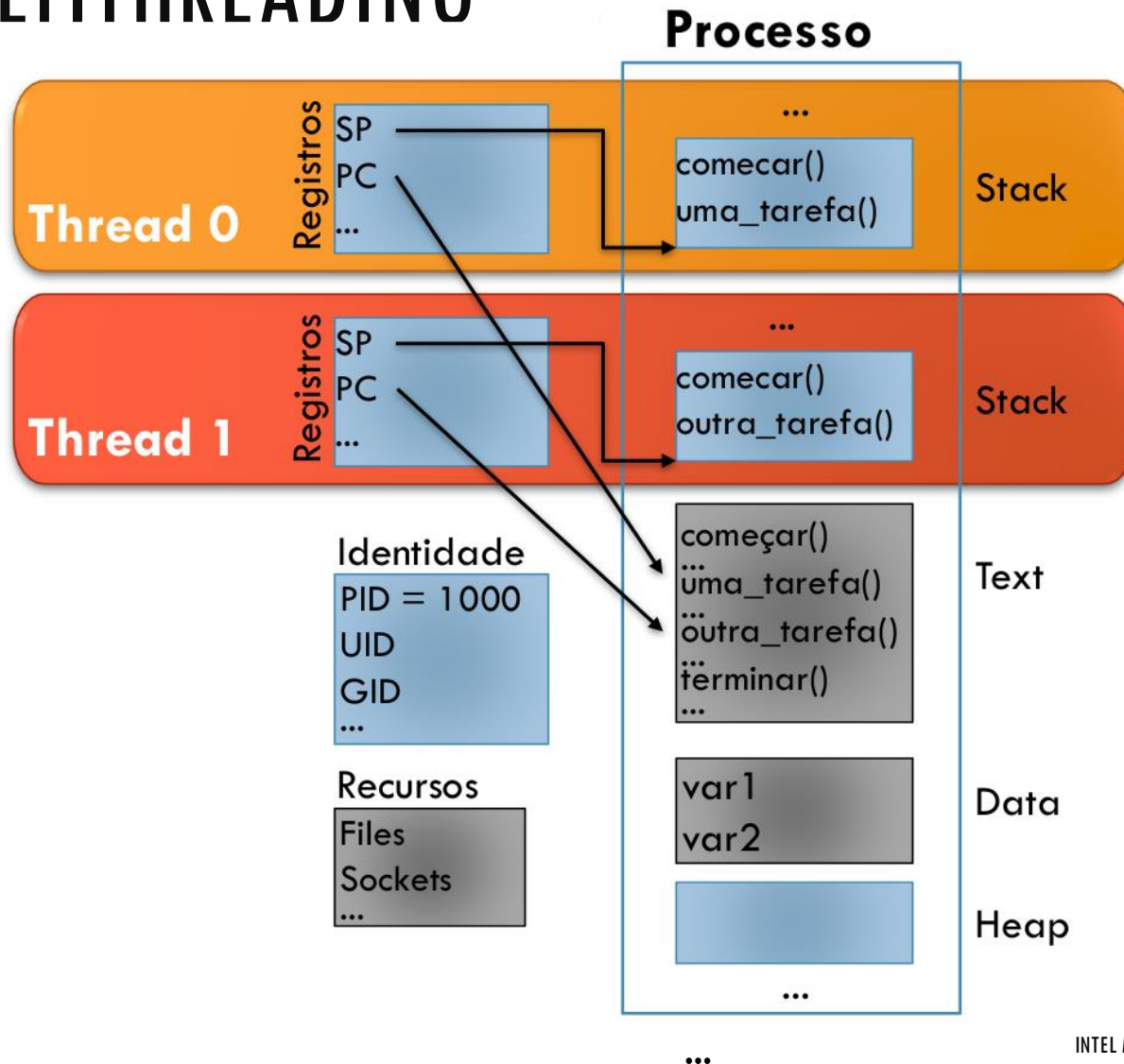
Qualquer alteração sobre uma posição de memória realizada por um determinado processador é igualmente visível por todos os restantes processadores.



# PROCESSOS



# MULTITHREADING



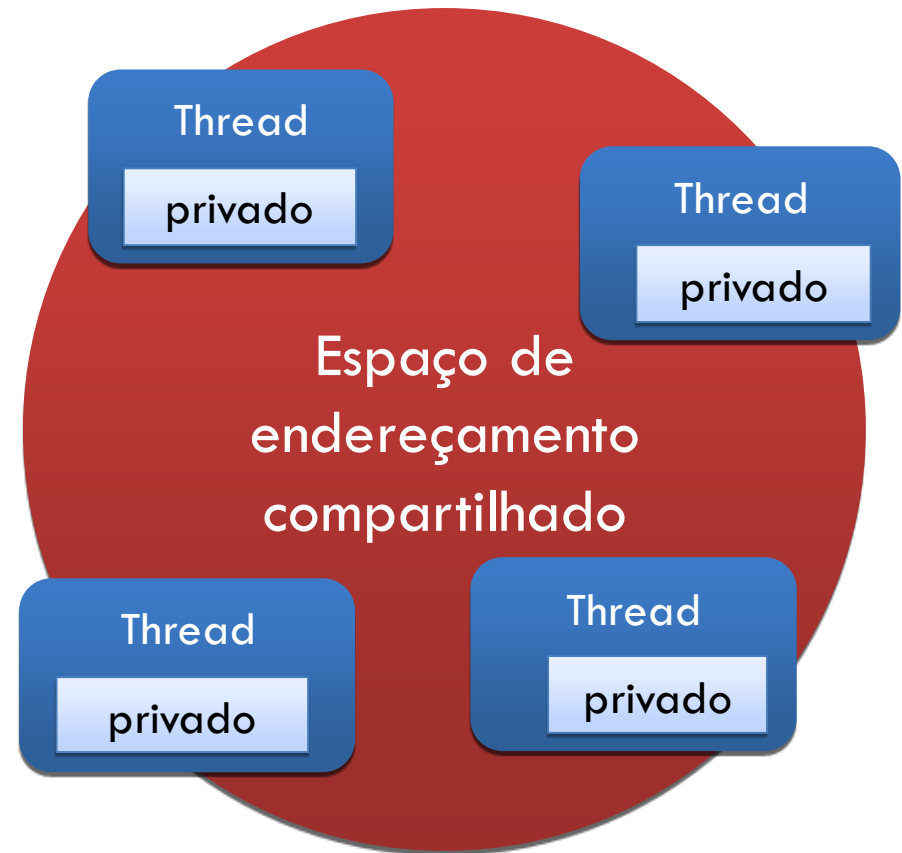
# UM PROGRAMA DE MEMÓRIA COMPARTILHADA

Uma instância do programa:

Um processo e muitas threads.

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

Sincronização garante a ordem correta dos resultados.



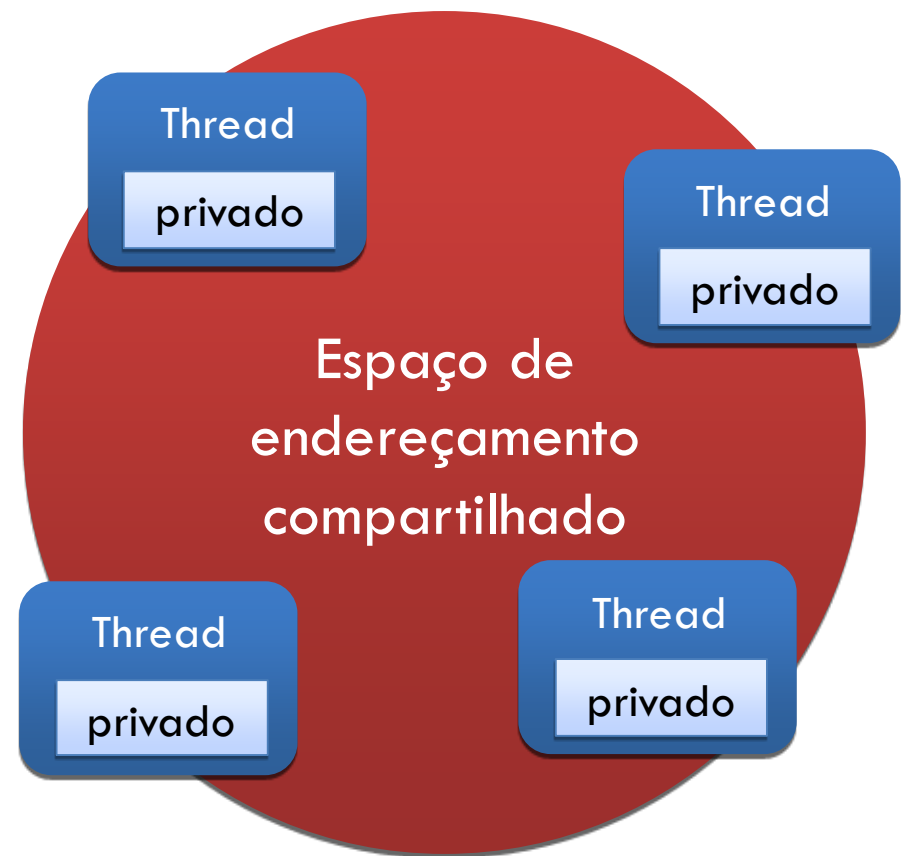
# UM PROGRAMA DE MEMÓRIA COMPARTILHADA

Uma instância do programa:

Um processo e muitas threads.

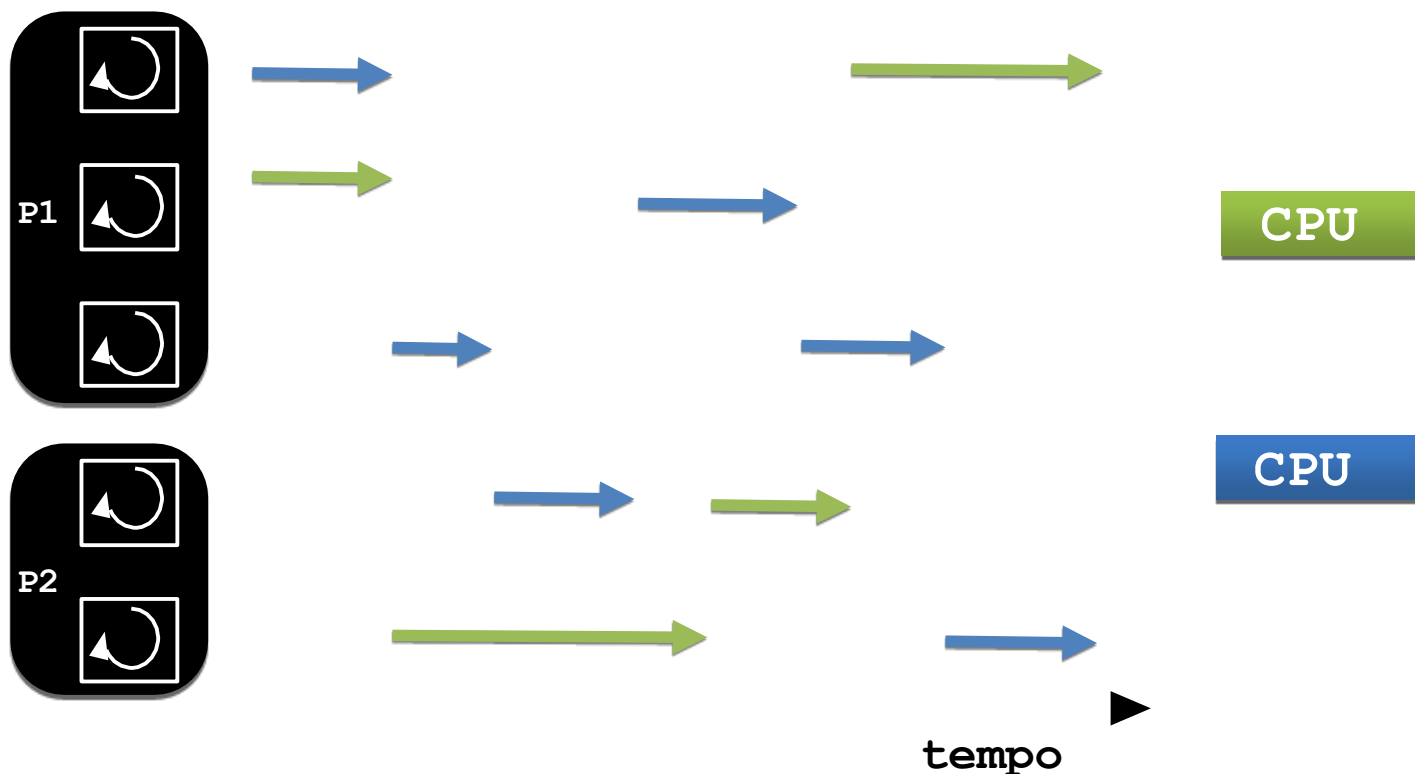
Threads interagem através de leituras/escrita com o espaço de endereçamento compartilhado.

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



# EXECUÇÃO DE PROCESSOS MULTITHREADED

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




# EXEMPLO, SOLUÇÃO

```
#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);
    }
}
```

Agora é possível  
entender esse  
comportamento!



Sample Output:

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

# EXERCÍCIOS

Login remoto a sistemas de computadores

```
ssh ufpelintel@term1
```

```
senha: ufpelintel
```

**conecta**

**Copie os exercícios**

```
1) cp -r ufrgs-intel-modern-code/ seu-nome/
```

```
2) cd seu-nome/
```

**Entre no diretório/pasta de cada exercício**

Exemplo (helloWorld)

```
1) cd helloWorld/
```

```
2) make
```

```
3) ./helloWorld.exec
```

# EXERCÍCIO 1: HELLO WORLD

`cd helloWorld/`    `make`    `./helloWorld.exec`

```
#include <stdio.h>

int main(){
    int myid, nthreads;

    myid = 0;

    nthreads = 1;
    printf("%d of %d - hello world!\n", myid, nthreads);

    return 0;
}
```



# FUNÇÕES

Funções da biblioteca OpenMP.

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

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

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

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

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

# SOLUÇÃO 1.1: HELLO WORLD

Variáveis privadas.

```
#include <stdio.h>
#include <omp.h>
int main(){
    int myid, nthreads;

    #pragma omp parallel private(myid, nthreads)
    {
        myid = omp_get_thread_num();

        nthreads = omp_get_num_threads();
        printf("%d of %d - hello world!\n", myid, nthreads);
    }
    return 0;
}
```

```
icc -o hello hello.c -qopenmp
./hello
0 of 2 - hello world!
1 of 2 - hello world!
```

# SOLUÇÃO 1.1: HELLO WORLD

Variáveis privadas.

```
#include <stdio.h>
#include <omp.h>
int main(){
    int myid, nthreads;
```

Cria threads em OpenMP

```
#pragma omp parallel private(myid, nthreads)
```

```
{
```

```
myid = omp_get_thread_num();
```

Função que retorna o ID de cada thread

```
nthreads = omp_get_num_threads();
```

Função que retorna o total de threads

```
printf("%d of %d - hello world!\n", myid, nthreads);
```

```
}
```

```
return 0;
```

```
}
```

# SOLUÇÃO 1.2: HELLO WORLD

Variáveis privadas e compartilhadas.

```
#include <stdio.h>
#include <omp.h>
int main(){
    int myid, nthreads;
```

```
icc -o hello hello.c -qopenmp
./hello
0 of 2 – hello world!
1 of 2 – hello world!
```

```
#pragma omp parallel private(myid) shared(nthreads)
{
    myid = omp_get_thread_num();
    #pragma omp single
    nthreads = omp_get_num_threads();
    printf("%d of %d – hello world!\n", myid, nthreads);
}
return 0;
}
```

# SOLUÇÃO 1.3: HELLO WORLD

NUM\_THREADS fora da região paralela?

```
#include <stdio.h>
#include <omp.h>
int main(){
    int myid, nthreads;

    nthreads = omp_get_num_threads();
    #pragma omp parallel private(myid) shared(nthreads)
    {
        myid = omp_get_thread_num();
        printf("%d of %d - hello world!\n", myid, nthreads);
    }
    return 0;
}
```

```
icc -o hello hello.c -qopenmp
./hello
```

# ~~SOLUÇÃO 1.3~~: HELLO WORLD

NUM\_THREADS fora da **região paralela**.

**Não funciona.**

```
#include <stdio.h>
#include <omp.h>
int main(){
    int myid, nthreads;

    nthreads = omp_get_num_threads();
    #pragma omp parallel private(myid) shared(nthreads)
    {
        myid = omp_get_thread_num();
        printf("%d of %d - hello world!\n", myid, nthreads);
    }
    return 0;
}
```

```
icc -o hello hello.c -qopenmp
```

```
./hello
```

```
0 of 1 - hello world!
```

```
1 of 1 - hello world!
```

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

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

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

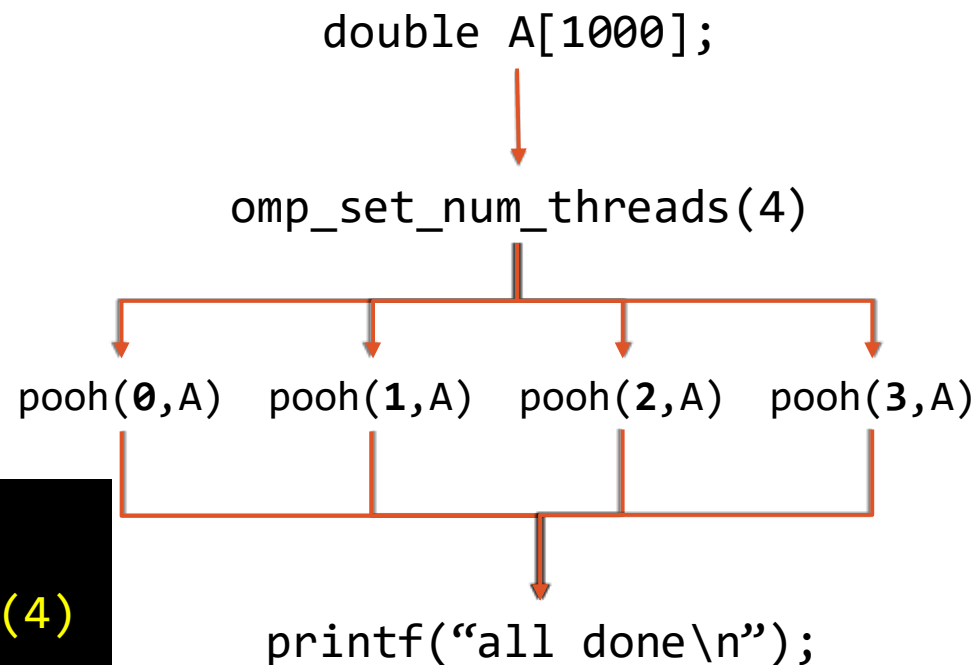


# CRIAÇÃO DE THREADS: REGIÕES PARALELAS

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**

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();

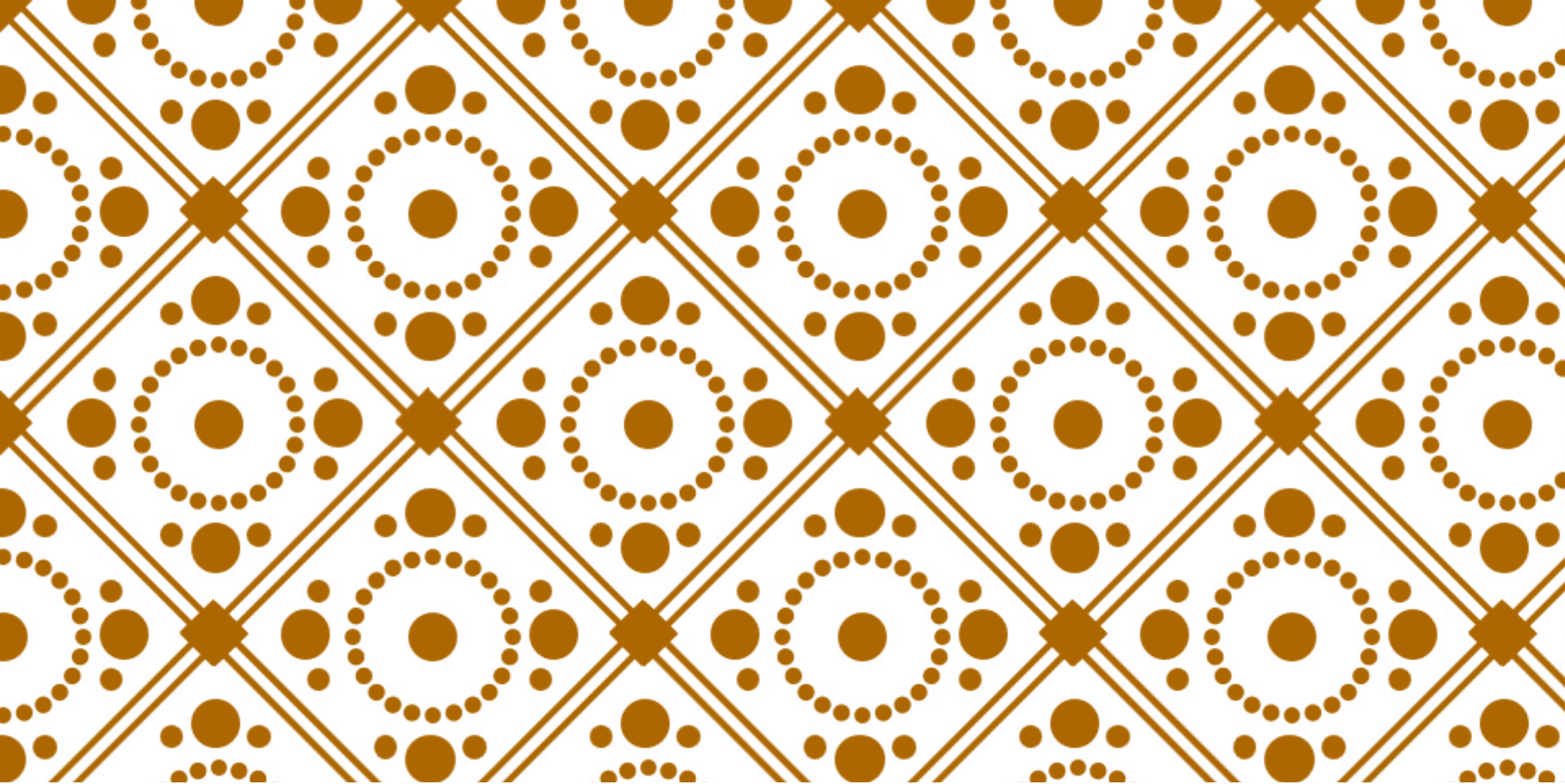
for (int i = 1; i < 4; ++i)
    pthread_join(tid[i]);
```

# SOLUÇÃO 1.2: HELLO WORLD

**Variáveis privadas e compartilhadas.**

```
#include <stdio.h>
#include <omp.h>
int main(){
    int myid, nthreads;

    #pragma omp parallel private(myid) shared(nthreads)
    {
        myid = omp_get_thread_num();
        #pragma omp single
        nthreads = omp_get_num_threads();
        printf("%d of %d - hello world!\n", myid, nthreads);
    }
    return 0;
}
```



# 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

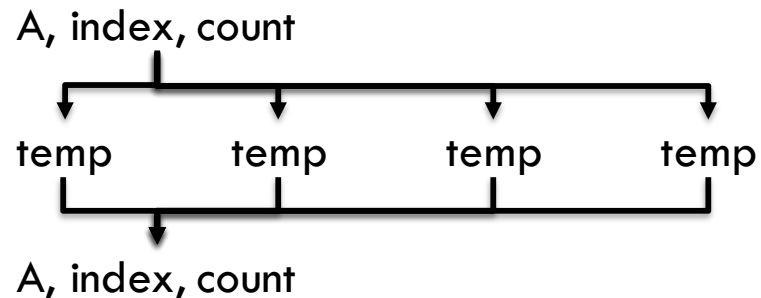
```
double A[10];  
  
int main() {  
    int index[10];  
    #pragma omp parallel  
    work(index);  
    printf("%d\n", index[0]);  
}
```

```
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 VÁLIDO?

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

As implementações podem 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);  
}
```

tmp tem um valor não especificado

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

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)

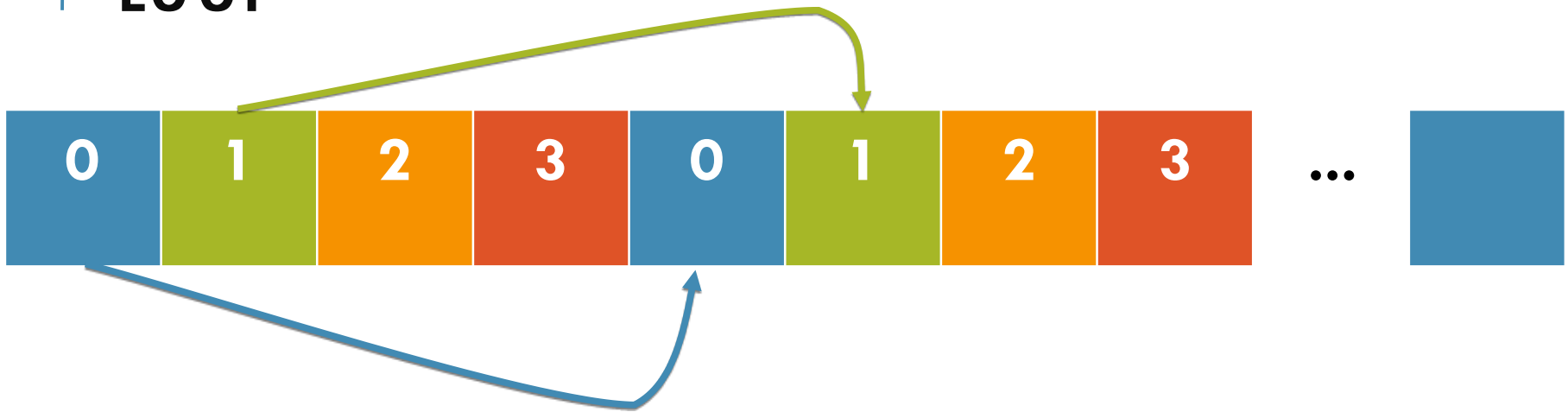
- Cada variável na construção será feita privada como se estivesse sido declaradas como `private(vars)`

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)`.**

# 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, PARTE A: VECTOR SUM

```
cd vectorSum/ make ./vectorSum.exec <elementos>
```

```
long long int sum(int *v, long long int N){  
    long long int i, sum = 0;  
  
    for(i = 0; i < N; i++)  
        sum += v[i];  
  
    return sum  
}
```

# FUNÇÕES

Funções da biblioteca OpenMP.

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

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

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

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

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

# DIRETIVAS

Diretivas do OpenMP.

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

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

}
```

# SOLUÇÃO 2.1, PARTE B: VECTOR SUM

```
cd vectorSum/ make ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, myid)

{
myid = omp_get_thread_num();
#pragma omp single
{
nthreads = omp_get_num_threads();
}

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

    sum += v[i];
} ...
```

## ~~SOLUÇÃO 2.1~~, PARTE B: VECTOR SUM

cd vectorSum/ make ./vectorSum.exec <elementos>

```
sum = 0;
#pragma omp parallel private(i, myid)

{
  myid = omp_get_thread_num();
  #pragma omp single
  {
    nthreads = omp_get_num_threads();
  }

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

    sum += v[i];
  ...
}
```

## ~~SOLUÇÃO 2.1~~, PARTE B: VECTOR SUM

```
cd vectorSum/ make ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, myid)

{
myid = omp_get_thread_num();
#pragma omp single
{
nthreads = omp_get_num_threads();
}

for(i = myid; i < N; i += nthreads)
    // RACE CONDITION
    sum += v[i]; // ler sum, v[i]; somar; escrever sum;
} ...
```

# 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

# CONDIÇÕES DE CORRIDA: EXEMPLO



Thread 0	Thread 1	sum
		0
<b>Leia</b> sum 0		0
	<b>Leia</b> sum 0	0
	<b>Some</b> 0, 5 5	0
<b>Some</b> 0, 10 10		0
	<b>Escreva</b> 5, sum 5	5
<b>Escreva</b> 10, sum 10		10
		15 !?



# CONDIÇÕES DE CORRIDA: EXEMPLO

Thread 0	Thread 1	sum
		0
Devemos garantir que <b>não importa a ordem de execução</b> , teremos sempre um resultado consistente!		
	Escreva 5, sum	5
Escreva 10, sum		10
10		15 !?

# SINCRONIZAÇÃO

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

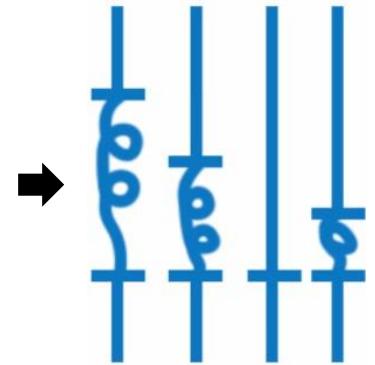
As duas formas mais comuns de sincronização são:

# 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



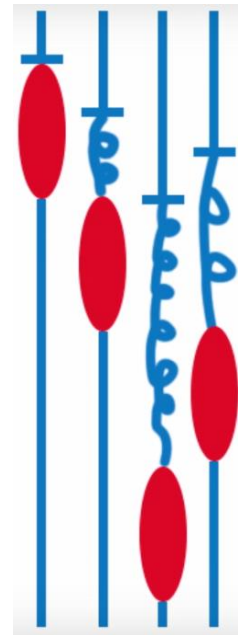
# SINCRONIZAÇÃO

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



# SINCRONIZAÇÃO: BARRIER

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

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

    #pragma omp barrier

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

# SINCRONIZAÇÃO: CRITICAL

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

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

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

# SINCRONIZAÇÃO: ATOMIC

**atomic** prove exclusão mútua para operações específicas.

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

Instruções especiais da  
arquitetura (se  
disponível)

Algumas operações aceitáveis:

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

# EXERCÍCIO 2, PARTE C: VECTOR SUM

```
cd vectorSum/ make ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, myid)

{
myid = omp_get_thread_num();
#pragma omp single
{
nthreads = omp_get_num_threads();
}

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

    sum += v[i];
} ...
```



# SOLUÇÃO 2.2, PARTE C: VECTOR SUM

```
cd vectorSum/    make    ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, myid)

{
myid = omp_get_thread_num();
#pragma omp single
{
nthreads = omp_get_num_threads();
}

for(i = myid; i < N; i += nthreads)
    #pragma omp critical
    sum += v[i];
} ...
```

## SOLUÇÃO 2.3, PARTE C: VECTOR SUM

```
cd vectorSum/ make ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, myid)

{
  myid = omp_get_thread_num();
  #pragma omp single
  {
    nthreads = omp_get_num_threads();
  }

  for(i = myid; i < N; i += nthreads)
    #pragma omp atomic
    sum += v[i];
  ...
}
```

# EXERCÍCIO 2, PARTE D: VECTOR SUM

```
cd vectorSum/ make ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, myid)

{
myid = omp_get_thread_num();
#pragma omp single
{
nthreads = omp_get_num_threads();
}

for(i = myid; i < N; i += nthreads)
    #pragma omp atomic
    sum += v[i];
} ...
```

Qual o problema da seção crítica dentro do loop?

# EXERCÍCIO 2, PARTE D: VECTOR SUM

```
cd vectorSum/ make ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, myid)

{
  myid = omp_get_thread_num();
  #pragma omp single
  {
    nthreads = omp_get_num_threads();
  }

  for(i = myid; i < N; i += nthreads)
    #pragma omp atomic
    sum += v[i];
} ...
```

Qual o problema da seção crítica dentro do loop?

Regiões atomic – n vezes. **Ex.** 1 000 000 000

# SOLUÇÃO 2.4, PARTE D: VECTOR SUM

```
cd vectorSum/    make    ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, sum_local, myid)

{
myid = omp_get_thread_num();    sum_local = 0;
#pragma omp single
{
nthreads = omp_get_num_threads();
}

for(i = myid; i < N; i += nthreads)
    sum_local += v[i];
#pragma omp atomic
sum += sum_local;
} ...
```

# SOLUÇÃO 2.4, PARTE D: VECTOR SUM

```
cd vectorSum/ make ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, sum_local, myid)

{
myid = omp_get_thread_num();    sum_local = 0;
#pragma omp single
{
nthreads = omp_get_num_threads();
}

for(i = myid; i < N; i += nthreads)
    sum_local += v[i];
#pragma omp atomic
sum += sum_local;
} ...
```

Regiões atomic – **nthreads** vezes. **Ex. 40** threads / vezes

# EXERCÍCIO 2, PARTE E: VECTOR SUM

```
cd vectorSum/ make ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, sum_local, myid)

{
myid = omp_get_thread_num();    sum_local = 0;
#pragma omp single
{
nthreads = omp_get_num_threads();
}

for(i = myid; i < N; i += nthreads)
    sum_local += v[i];
#pragma omp atomic
sum += sum_local;
} ...
```

Existe uma solução melhor?

# EXERCÍCIO 2, PARTE E: VECTOR SUM

`cd vectorSum/`    `make`    `./vectorSum.exec <elementos>`

```
sum = 0;
#pragma omp parallel private(i, sum_local, myid)

{
myid = omp_get_thread_num();      sum_local = 0;
#pragma omp single
{
nthreads = omp_get_num_threads();
}

for(i = myid; i < N; i += nthreads)
    sum_local += v[i];
#pragma omp atomic
sum += sum_local;
} ...
```

Existe uma solução melhor?

Como a cache funciona?



# PRINCÍPIO DA LOCALIDADE

Programas repetem trechos de código e acessam repetidamente dados próximos.

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

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

# ACESSOS INTERCALADOS

TEMPO

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

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

25% do conteúdo trazido para a cache é utilizado.

# ACESSOS CONSECUTIVOS

TEMPO

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

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

100% do conteúdo trazido para a cache é utilizado.

# EXERCÍCIO 2, PARTE E: VECTOR SUM

```
cd vectorSum/ make ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, sum_local, myid)
{
    myid = omp_get_thread_num();    sum_local = 0;
    #pragma omp single
    {
        nthreads = omp_get_num_threads();
    }

    for(i = myid; i < N; i += nthreads)
        sum_local += v[i];
    #pragma omp atomic
    sum += sum_local;
} ...
```

# SOLUÇÃO 2.5, PARTE E: VECTOR SUM

`cd vectorSum/`    `make`    `./vectorSum.exec <elementos>`

```
sum = 0;
#pragma omp parallel private(i, sum_local, myid, init, end)
{
    myid = omp_get_thread_num();    sum_local = 0;
    #pragma omp single
    {
        nthreads = omp_get_num_threads(); slice = N / nthreads;
    }
    init = myid * slice;
    if(myid == nthreads - 1) end = N; else end = init + slice;
    for(i = init; i < end; i++)
        sum_local += v[i];
    #pragma omp atomic
    sum += sum_local;
} ...
```

# EXERCÍCIO 2, PARTE F: VECTOR SUM

`cd vectorSum/`    `make`    `./vectorSum.exec <elementos>`

```
sum = 0;
#pragma omp parallel private(i, sum_local, myid, init, end)
{
    myid = omp_get_thread_num();    sum_local = 0;
    #pragma omp single
    {
        nthreads = omp_get_num_threads(); slice = N / nthreads;
    }
    init = myid * slice;
    if(myid == nthreads - 1) end = N; else end = init + slice;
    for(i = init; i < end; i++)
        sum_local += v[i];
    #pragma omp atomic
    sum += sum_local;
} ...
```

**OpenMP é um modelo relativamente fácil de usar**

# CONSTRUÇÕES DE DIVISÃO DE LAÇOS

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

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

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

# 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



# CONSTRUÇÕES DE DIVISÃO DE LAÇOS

## UM EXEMPLO MOTIVADOR

Código sequencial

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

# CONSTRUÇÕES DE DIVISÃO DE LAÇOS

## UM EXEMPLO MOTIVADOR

Código sequencial

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

Região OpenMP parallel

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

# CONSTRUÇÕES DE DIVISÃO DE LAÇOS

## UM EXEMPLO MOTIVADOR

Código sequencial

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

Região OpenMP parallel

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

Região paralela OpenMP  
com uma construção de  
divisão de trabalho

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

## EXERCÍCIO 2, PARTE F: VECTOR SUM

`cd vectorSum/`    `make`    `./vectorSum.exec <elementos>`

```
sum = 0;
#pragma omp parallel private(i, sum_local, myid, init, end)
{
    myid = omp_get_thread_num();    sum_local = 0;
    #pragma omp single
    {
        nthreads = omp_get_num_threads(); slice = N / nthreads;
    }
    init = myid * slice;
    if(myid == nthreads - 1) end = N; else end = init + slice;
    for(i = init; i < end; i++)
        sum_local += v[i];
    #pragma omp atomic
    sum += sum_local;
} ...
```

# SOLUÇÃO 2.6, PARTE F: VECTOR SUM

```
cd vectorSum/    make    ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, sum_local)
{
    sum_local = 0;

    #pragma omp for
    for(i = 0; i < N; i++)
        sum_local += v[i];
    #pragma omp atomic
    sum += sum_local;
} ...
```

# EXERCÍCIO 2, PARTE G: VECTOR SUM

```
cd vectorSum/    make    ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, sum_local)
{
    sum_local = 0;
```

```
#pragma omp for
for(i = 0; i < N; i++)
    sum_local += v[i];
#pragma omp atomic
sum += sum_local;
} ...
```

OpenMP é um modelo relativamente **fácil** de usar

# REDUÇÃO

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

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

# DIRETIVA REDUCTION

`reduction(op : list_vars)`

Dentro de uma região paralela ou de divisão de trabalho:

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

**`#pragma omp for reduction(* : var_mult)`**



# EXERCÍCIO 2, PARTE G: VECTOR SUM

```
cd vectorSum/ make ./vectorSum.exec <elementos>
```

```
sum = 0;
#pragma omp parallel private(i, sum_local)
{
    sum_local = 0;
```

```
#pragma omp for
for(i = 0; i < N; i++)
    sum_local += v[i];
#pragma omp atomic
sum += sum_local;
} ...
```

OpenMP é um modelo relativamente **fácil** de usar

# SOLUÇÃO 2.7, PARTE G: VECTOR SUM

```
cd vectorSum/    make    ./vectorSum.exec <elementos>
```

```
sum = 0;
```

```
#pragma omp parallel for private(i) reduction(+ : sum)
```

```
for(i = 0; i < N; i++)
```

```
    sum += v[i];
```

```
...
```

# VECTOR SUM

## Sequential vs. Paralelo

```
sum = 0;  
  
for(i = 0; i < N; i++)  
    sum += v[i];
```

```
sum = 0;  
#pragma omp parallel for default(shared) reduction(+ : sum)  
for(i = 0; i < N; i++)  
    sum += v[i];
```

# RESULTADOS\*

Entrada 1 000 000 000 (1B), executou em 0.69 seg.

\* 2 x Xeon E5-2640 v2, 8 cores, 2 SMT-cores

**2 proc. x 8 cores** x 2 SMT = 32 HW Threads

▪ 16 cores físicos

#	2.2 critical	2.3 atomic	2.4 atomic improved	2.5 atomic improved cache	2.6 for	2.7 for Reduction
1	<b>0.69</b>					
8	346.42	36.85	0.67	0.17	0.17	0.17
16	371.17	37.13	0.59	<b>0.12</b>	<b>0.12</b>	<b>0.12</b>
32	406.41	38.40	1.38	0.19	0.19	0.18

# SPEEDUP

Entrada 1 000 000 000 (1B), executou em 0.69 seg.

\* 2 x Xeon E5-2640 v2, 8 cores, 2 SMT-cores

**2 proc. x 8 cores** x 2 SMT = 32 HW Threads

▪ 16 cores físicos

#	2.2 critical	2.3 atomic	2.4 atomic improved	2.5 atomic improved cache	2.6 for	2.7 for Reduction
1	1.00					
8	0.0019	0.019	1.03	4.06	4.06	4.06
16	0.0018	0.018	1.17	<b>5.75</b>	<b>5.75</b>	<b>5.75</b>
32	0.0017	0.017	0.49	3.63	3.63	3.63

# EXERCÍCIO 3: SELECTION SORT

```
cd selectionSort/    make    ./selectionSort.exec
```

```
void selection_sort(int *v, int n){
    int i, j, min, tmp;

    for(i = 0; i < n - 1; i++){
        min = i;

        for(j = i + 1; j < n; j++){
            if(v[j] < v[min])
                min = j;

        tmp = v[i];
        v[i] = v[min];
        v[min] = tmp;
    }
}
```

# SOLUÇÃO 3.1: SELECTION SORT

`cd selectionSort/`      `make`      `./selectionSort.exec <elementos>`

```
for(i = 0; i < n - 1; i++){
    #pragma omp parallel default(shared) private(j, min_local)
    {
        min_local = i;
        #pragma omp single
        min = i
        #pragma omp for
        for(j = i + 1; j < n; j++)
            if(v[j] < v[min_local])
                min_local = j;
        #pragma omp critical
        if(v[min_local] < v[min])
            min = min_local;
    }
    tmp = v[i];
    v[i] = v[min];
    v[min] = tmp;
}
```

# RESULTADOS\*

Entrada 1 000 000, executou em 301.53 seg.

\* 2 x Xeon E5-2640 v2, 8 cores, 2 SMT-cores

2 proc. x 8 cores x 2 SMT = 32 HW Threads


□ 16 cores físicos

#	Tempo	Speedup
1	301.53	1.00
8	136.17	2.21
16	81.49	3.70
32	61.69	4.89



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

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



Como o laço  
será mapeado  
para as  
threads?

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

`schedule(static [,chunk])`

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

`schedule(dynamic[,chunk])`

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

`schedule(guided[,chunk])`

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

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

`schedule(runtime)`

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

`schedule(auto)`  **Novo**

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

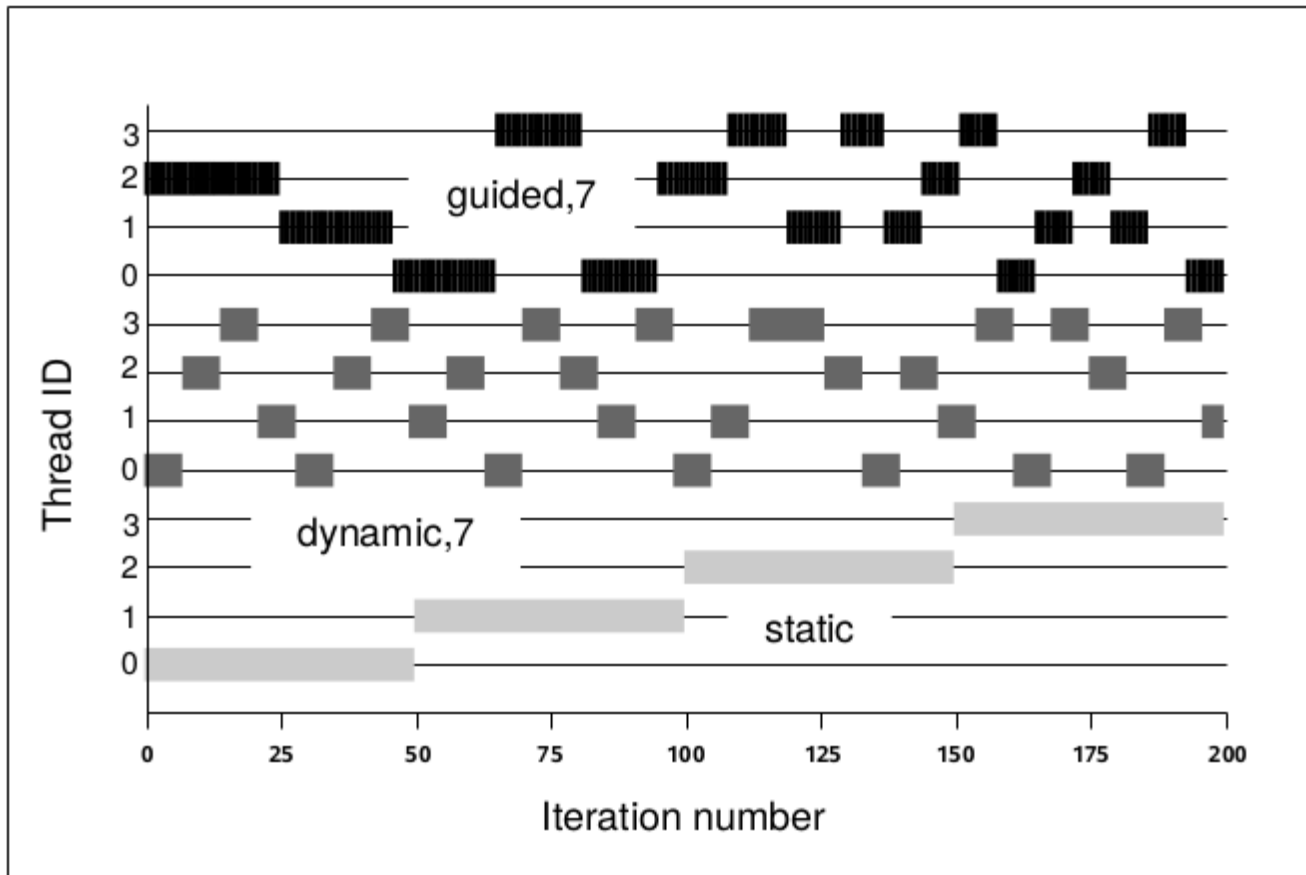
# CONSTRUÇÕES DE DIVISÃO DE LAÇOS : A DECLARAÇÃO **SCHEDULE**

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)

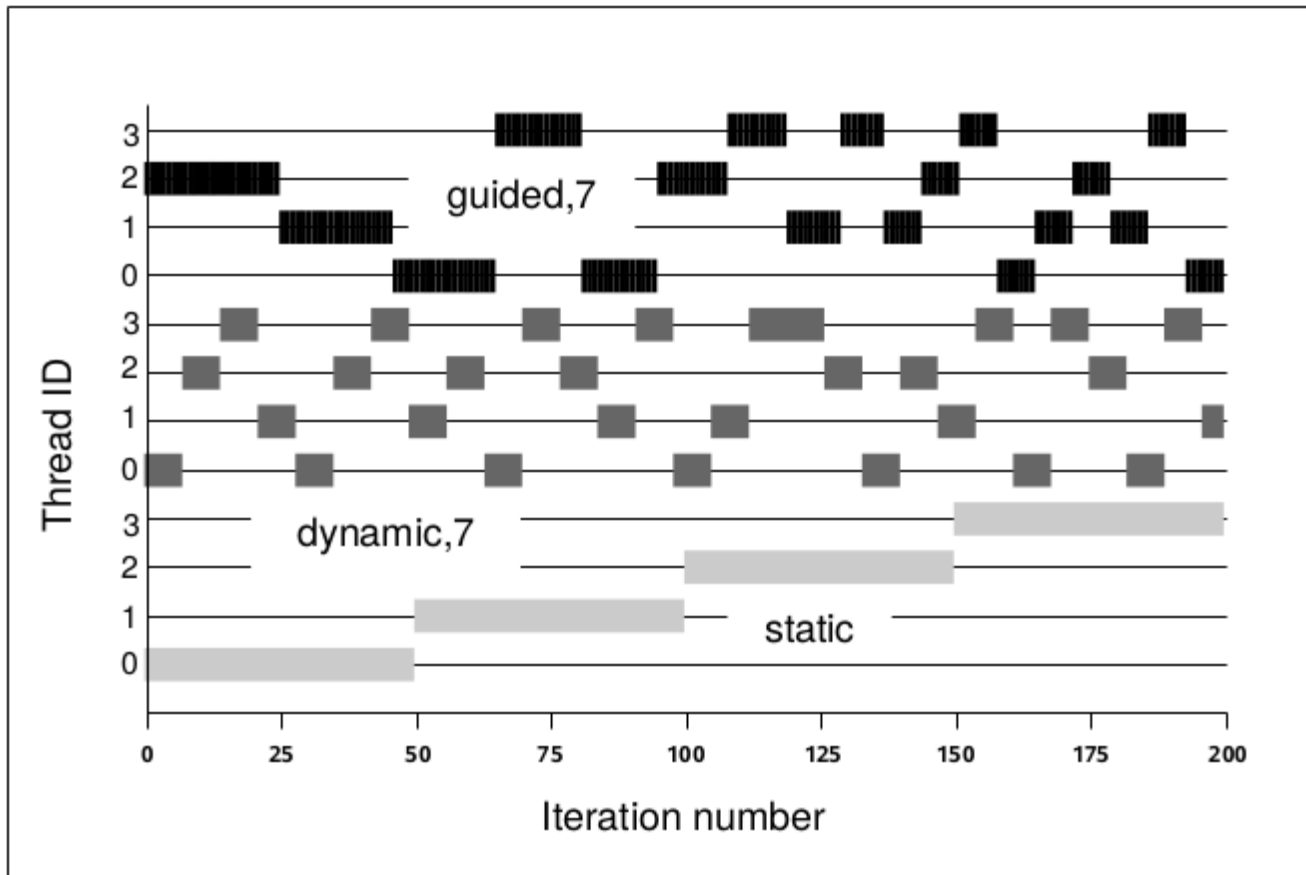
# CONSTRUÇÕES DE DIVISÃO DE LAÇOS : A DECLARAÇÃO **SCHEDULE**



Exemplo de escalonamento para 200 iterações.

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

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



Exemplo de escalonamento para 200 iterações.

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

# EXERCÍCIO 4: VECTOR SUM - SCHEDULE

```
cd vectorSum/    make    ./vectorSum.exec <elementos>
```

```
sum = 0;
```

```
#pragma omp parallel for private(i) reduction(+ : sum)
```

```
for(i = 0; i < N; i++)
```

```
    sum += v[i];
```

```
...
```

# SOLUÇÃO 4.1: VECTOR SUM - SCHEDULE

```
cd vectorSum/    make    ./vectorSum.exec <elementos>
```

```
sum = 0;

#pragma omp parallel for private(i) reduction(+ : sum)
schedule(dynamic,4)
for(i = 0; i < N; i++)
    sum += v[i];

...
```



# SOLUÇÃO 4.2: VECTOR SUM - SCHEDULE

```
cd vectorSum/    make    ./vectorSum.exec <elementos>
```

```
sum = 0;

#pragma omp parallel for private(i) reduction(+ : sum)
schedule(guided,4)
for(i = 0; i < N; i++)
    sum += v[i];

...
```

# SOLUÇÃO 4.3: VECTOR SUM - SCHEDULE

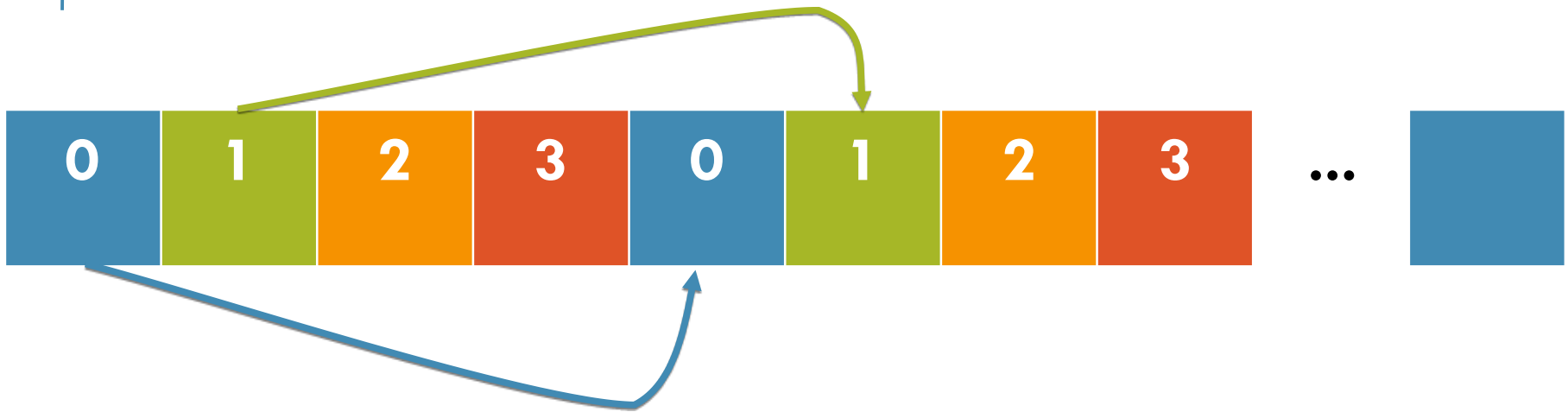
```
cd vectorSum/    make    ./vectorSum.exec <elementos>
```

```
sum = 0;

#pragma omp parallel for private(i) reduction(+ : sum)
schedule(runtime)
for(i = 0; i < N; i++)
    sum += v[i];

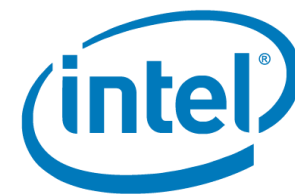
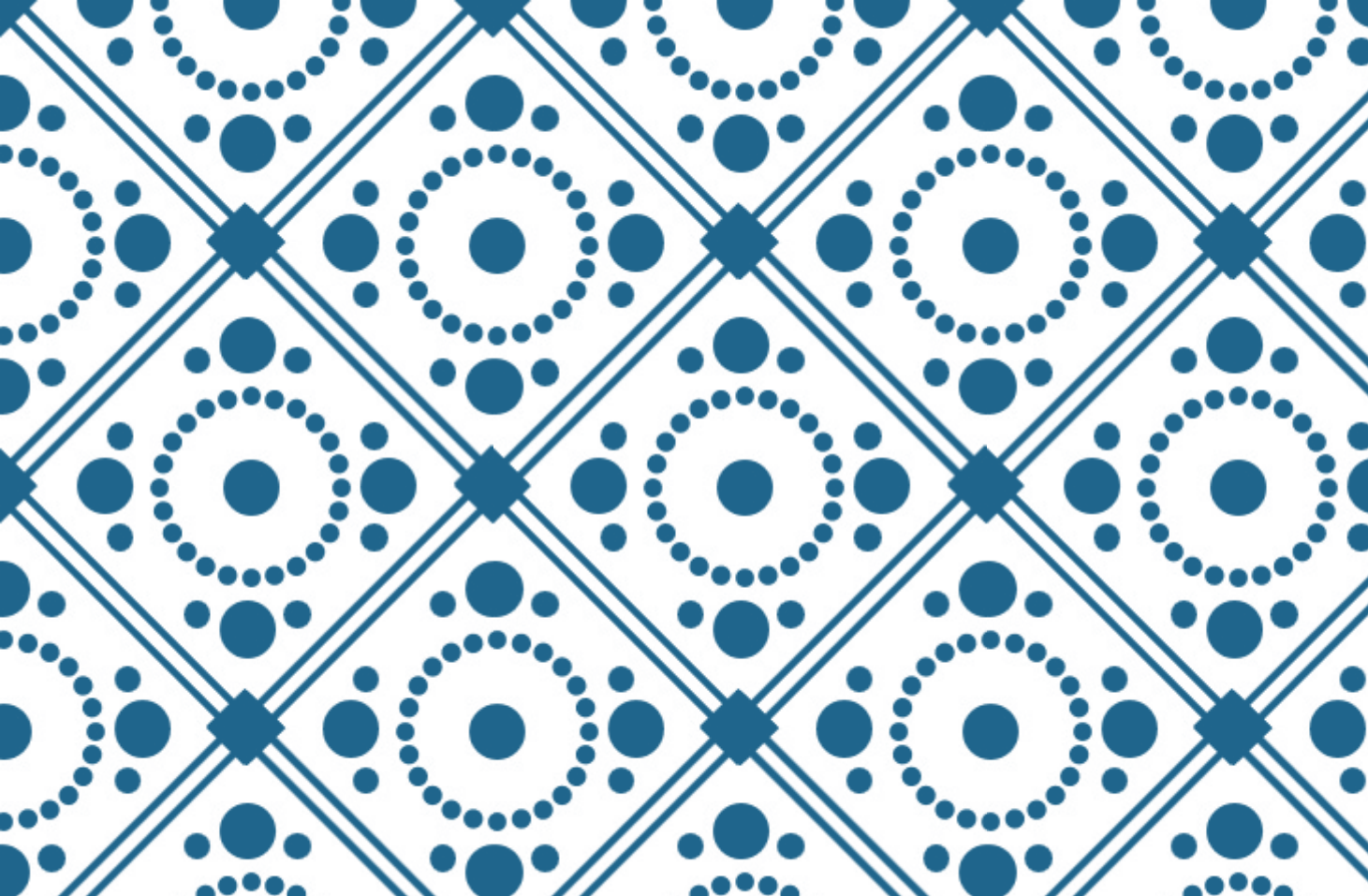
...
```

## EXERCÍCIO 4: VECTOR SUM - SCHEDULE



Qual é a diretiva schedule equivalente a essa distribuição (Exercício 2, Parte E)?





# INTEL MODERN CODE PARTNER

## UFPEL

