

# INTEL MODERN CODE PARTNER

## UFPEL



# TESTANDO O AMBIENTE REMOTO

Login remoto a sistemas de computadores

```
ssh ufpelintel@
```

```
senha: ufpelintel
```

**conecta ou conecta2**

Copie os exercícios

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

```
cd seu-nome/
```

Trabalhe dentro de seu diretório.

# OPENMP - REVISÃO

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
```

# OPENMP - REVISÃO

Diretivas do OpenMP.

```
#pragma omp parallel
```

```
#pragma omp for
```

```
#pragma omp single
```

# OPENMP - REVISÃO

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
```

```
OMP_NUM_THREADS=30 ./a.out
```



Para shell bash

Por padrão é o nº de  
proc. virtuais.

# 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)

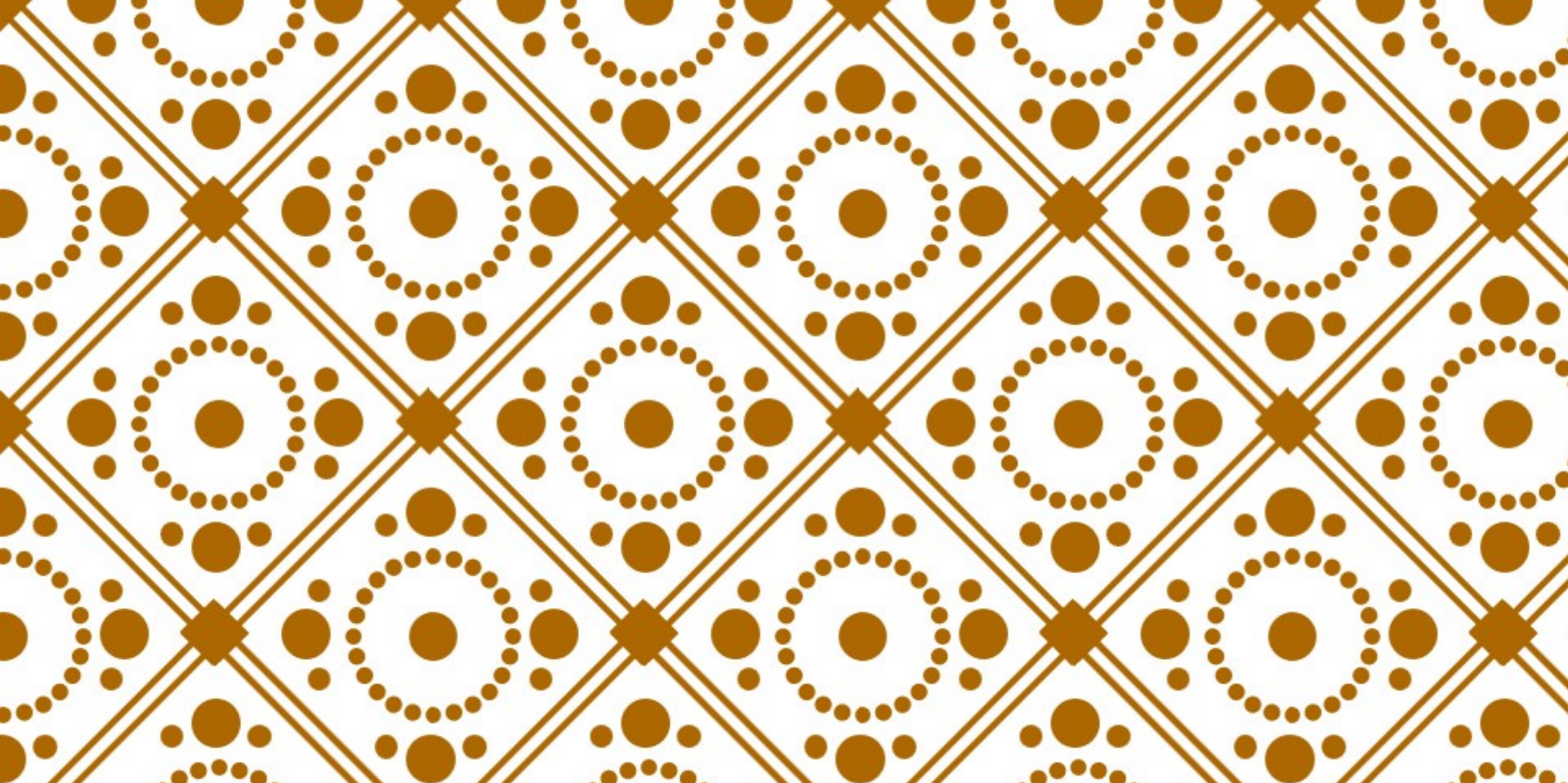
# SPMD VS. WORKSHARING

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**




# SECTIONS



# CONSTRUÇÃO **SECTIONS** PARA DIVISÃO DE TRABALHO

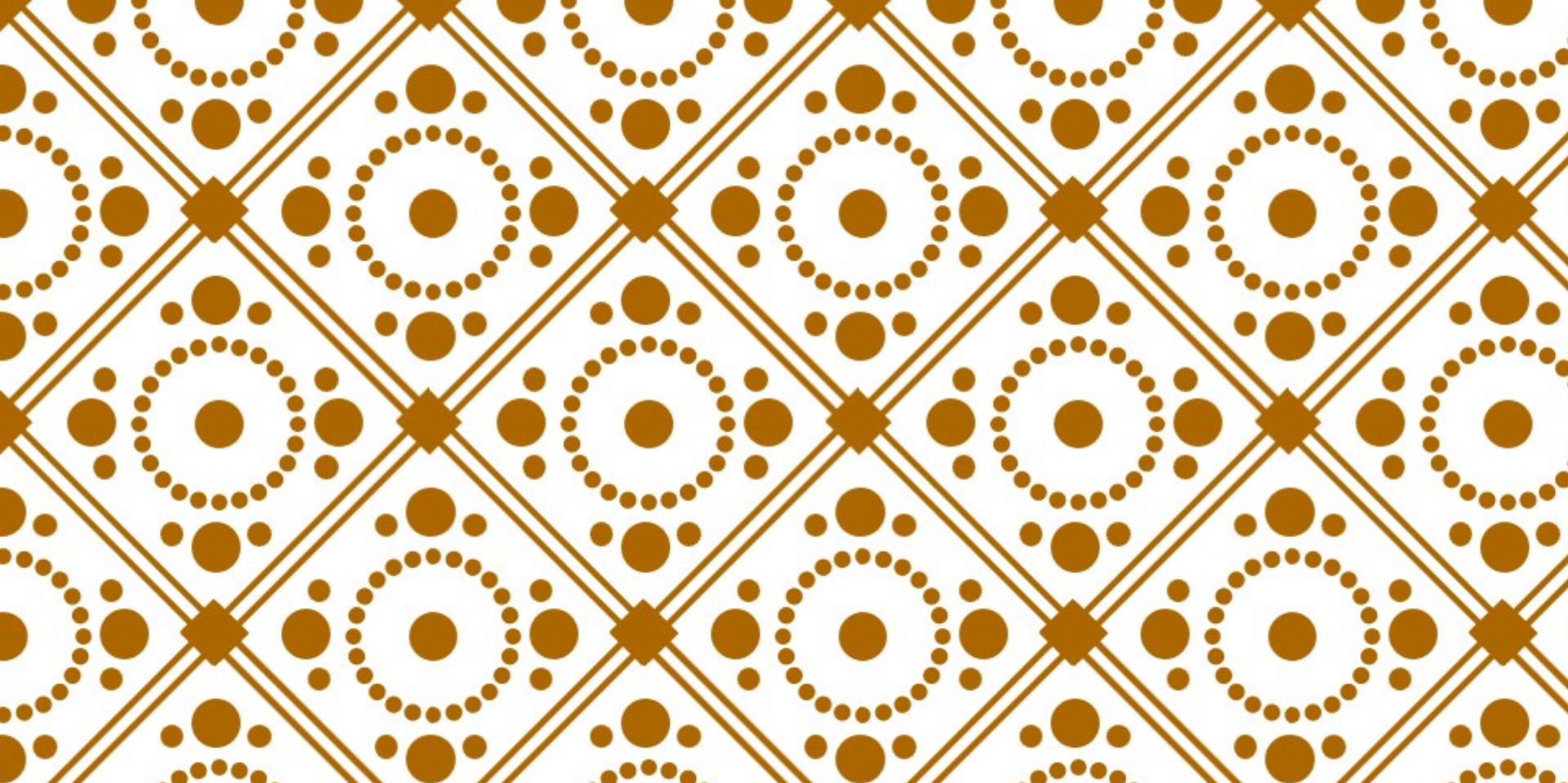
A construção de divisão de trabalho com **sections** prove um bloco estruturado diferente para cada thread.

```
#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.



# MASTER SINGLE

# 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

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

**Barreira implícita**

Nesse caso, podemos usar “**nowait**”

# 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

# EXEMPLO, SINGLE MASTER

`cd singleMaster/`    `make`    `./singleMaster.exec`

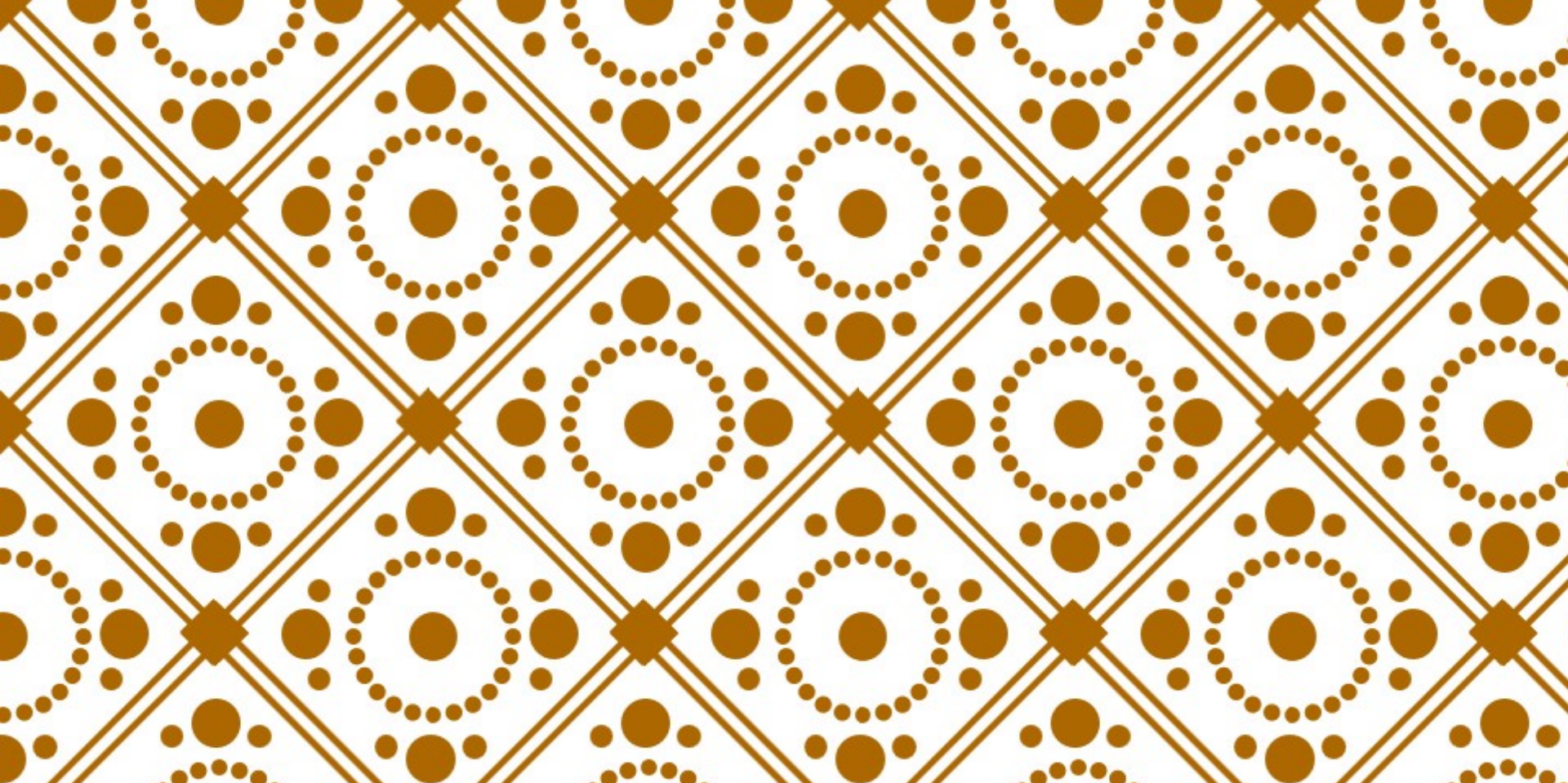
```
#include <stdio.h>
#include <omp.h>

int main() {

    #pragma omp parallel
    {
        #pragma omp master
        printf(" master - this is thread %d\n", omp_get_thread_num());

        #pragma omp single
        printf(" single - this is thread %d\n", omp_get_thread_num());
    }

}
```



# TASKS

# OPENMP TASKS

Tasks são unidades de trabalho independentes.

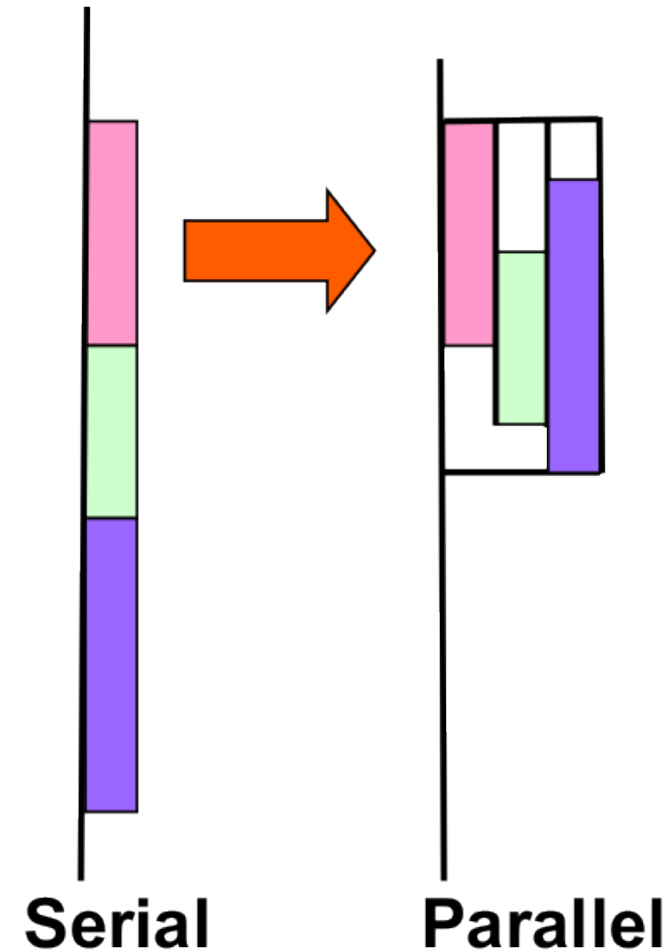
Tasks são compostas de:

- 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

O 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

# EXERCÍCIO, FIBONACCI TASK

```
cd fibonacciTask/ make ./fibonacciTask.exec
```

```
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;  
}
```

Compare o desempenho da  
versão **sequencial**:

**./fibonacciTask.exec 45**

com a versão **paralela**:

**./fibonacciTaskPar.exec 45**

# EXERCÍCIO, FIBONACCI TASK

```
cd fibonacciTask/ make ./fibonacciTask.exec
```

```
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;  
}
```

./fibonacciTask.exec 45

time: 7.236 seconds

com a versão paralela:

./fibonacciTaskPar.exec 45

time: 65.665 seconds

por que isto acontece??

# EXERCÍCIO, FIBONACCI TASK

```
cd fibonacciTask/ make ./fibonacciTask.exec
```

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

./fibonacciTask.exec 45

time: 7.236 seconds

Criamos muitas tarefas com muito pouco trabalho. Sobrecusto de criação de tarefas muito alto...  
Solução: criar menos tarefas

paralela:

par.exec 45

time: 65.665 seconds

por que isto acontece??

# SOLUÇÃO 1, FIBONACCI TASK

```
cd fibonacciTask/ make ./fibonacciTask.exec
```

```
int fib (int n){  
    int x,y;  
    if ( n < 2 ) return n;  
    #pragma omp task shared(x) if(n > 35)  
    x = fib(n-1);  
    #pragma omp task shared(y) if(n > 35)  
    y = fib(n-2);  
    #pragma omp taskwait  
    return x+y;  
}
```

# SOLUÇÃO 1, FIBONACCI TASK

```
cd fibonacciTask/ make ./fibonacciTask.exec
```

```
int fib (int n){  
    int x,y;  
    if ( n < 2 ) return n;  
    #pragma omp task shared(x) if(n > 35)  
    x = fib(n-1);  
    #pragma omp task shared(y) if(n > 35)  
    y = fib(n-2);  
    #pragma omp taskwait  
    return x+y;  
}
```

Algo mais?



# SOLUÇÃO 2, FIBONACCI TASK

```
cd fibonacciTask/ make ./fibonacciTask.exec
```

```
int fib (int n){  
    int x,y;  
    if ( n < 2 ) return n;  
    #pragma omp task shared(x) if(n > 35)  
    x = fib(n-1);  
  
    y = fib(n-2);  
    #pragma omp taskwait  
    return x+y;  
}
```

Deixar uma das  
chamadas para ser  
executada pela  
tarefa mãe!

# SOLUÇÃO 2, FIBONACCI TASK

```
cd fibonacciTask/ make ./fibonacciTask.exec
```

```
int fib (int n){  
    int x,y;  
    if ( n < 2 ) return n;  
    #pragma omp task shared(x) if(n > 35)  
    x = fib(n-1);  
  
    y = fib(n-2);  
    #pragma omp taskwait  
    return x+y;  
}
```

Ok... versão paralela melhorou mas ainda é mais lenta que a sequencial. Talvez seja melhor resolver outros problemas.

Dica: vários benchmarks com tasks

(<https://github.com/bsc-pm/bots>)

# TASKS

- **if( cond):** cria nova task apenas se a condição for verdadeira
  - `#pragma omp task shared(x) if(n > 35)`
- **tied / untied**
  - quando a execução de uma tarefa **untied** é suspensa, a continuação da **task** pode ser retomada por qualquer outra thread do team.
  - default é **tied**
    - são executadas sempre pela mesma thread

# 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.

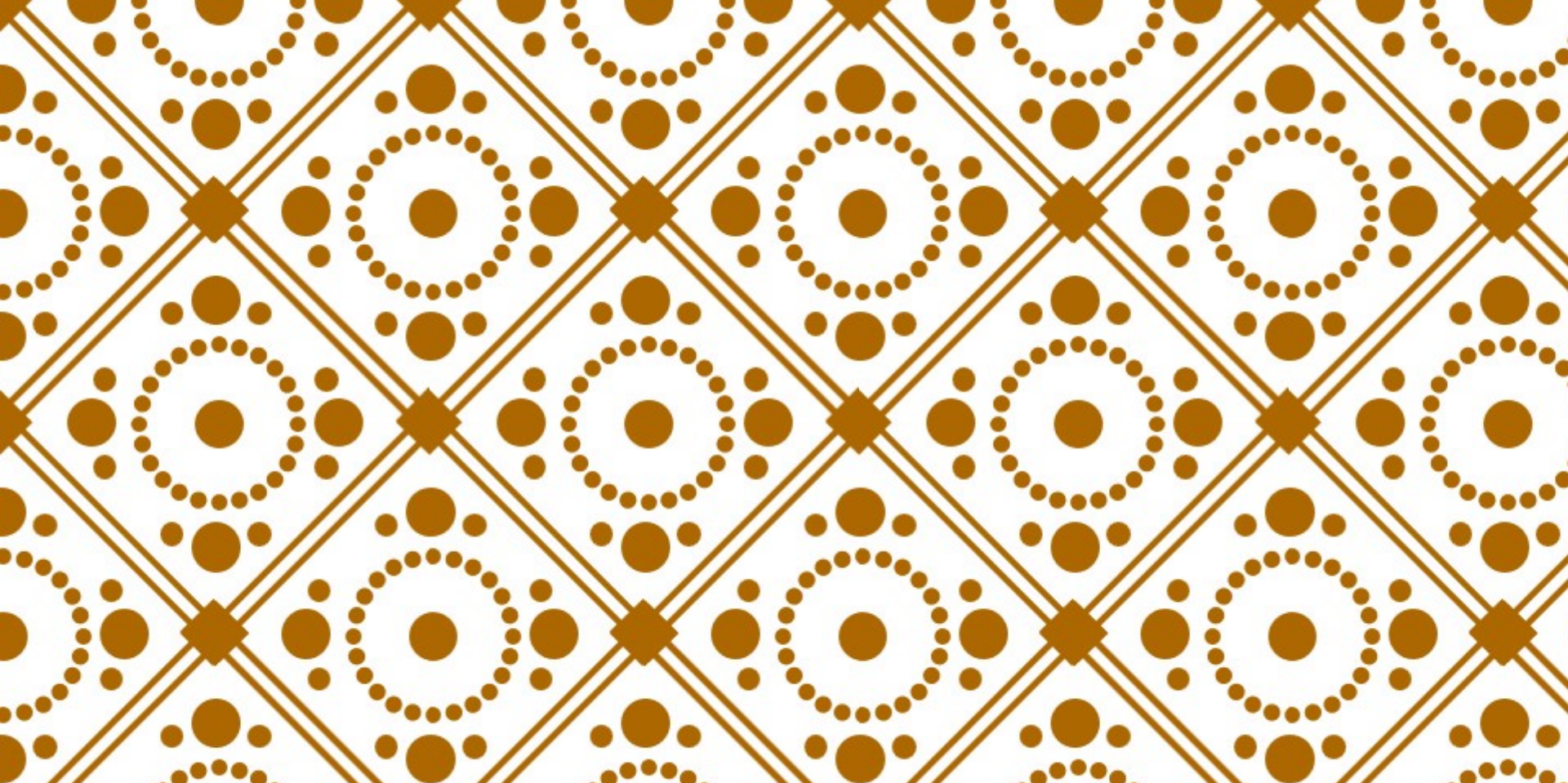
Variáveis alocadas no **heap**, serão **compartilhadas**.

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

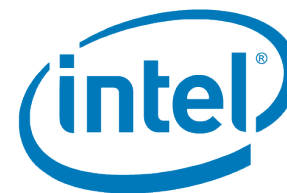
# REGRAS DE ESCOPO DE VARIÁVEIS

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*.



# INTRODUÇÃO A PROGRAMAÇÃO VETORIAL



# *SINGLE INSTRUCTION MULTIPLE DATA* (SIMD)

Técnica aplicada por unidade de execução

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

Elementos são armazenados em registradores SIMD

## Scalar

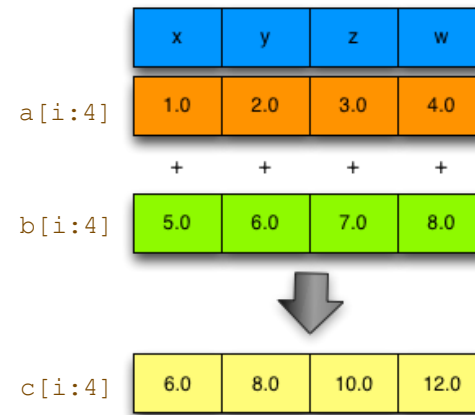
Uma instrução. Uma operação.

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

## Vector

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

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



# *SINGLE INSTRUCTION MULTIPLE DATA* (SIMD)

Técnica aplicada por unidade de execução

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

Elementos são armazenados em registradores SIMD

## Scalar

Uma instrução. Uma operação.

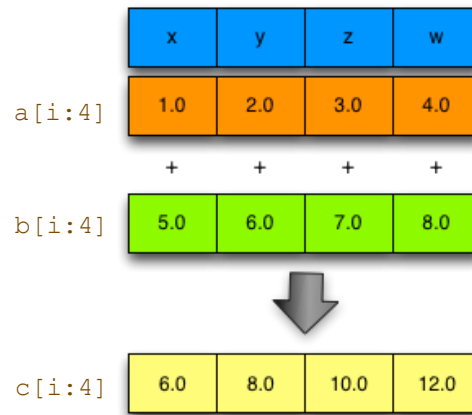
```
for(i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

## Vector

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

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

Dados contíguos para desempenho ótimo  
c[0] c[1] c[2] c[3] ...





# ALINHAMENTO DE MEMÓRIA

## Alinhamento de dados

Funções do compilador **icc**.

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

## Indicar ao compilador que dados estão alinhados

Ajuda na auto vetorização.

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

# PROGRAMAÇÃO VETORIAL

## Vetorização

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

## Vetorização com redução

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

# EXERCÍCIO 4, PARTE A: DOT PRODUCT SIMD

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

```
double dotproduct(double *a, int *b, long long int N){  
    long long int i;  
    double dot = 0.0;  
  
    for(i = 0; i < N; i++)  
        dot += a[i] * b[i];  
  
    return dot;  
}
```

# SOLUÇÃO 4.1, PARTE A: DOT PRODUCT SIMD

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

```
double dotproduct(double *a, int *b, long long int N){
    long long int i;
    double dot = 0.0;

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

    return dot;
}
```

# EXERCÍCIO 4, PARTE B:

## DOT PRODUCT PARALLEL

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

```
double dotproduct(double *a, int *b, long long int N){
    long long int i;
    double dot = 0.0;

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

    return dot;
}
```

# SOLUÇÃO 4.2, PARTE B: DOT PRODUCT PARALLEL

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

```
double dotproduct(double *a, int *b, long long int N){  
    long long int i;  
    double dot = 0.0;  
  
    #pragma omp parallel for private(i) reduction(+ : dot)  
    for(i = 0; i < N; i++)  
        dot += a[i] * b[i];  
  
    return dot;  
}
```

# EXERCÍCIO 4, PARTE C:

## DOT PRODUCT PARALLEL SIMD

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

```
double dotproduct(double *a, int *b, long long int N){
    long long int i;
    double dot = 0.0;

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

    return dot;
}
```

# SOLUÇÃO 4.3, PARTE C: DOT PRODUCT PARALLEL SIMD

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

```
double dotproduct(double *a, int *b, long long int N){  
    long long int i;  
    double dot = 0.0;  
  
    #pragma vector aligned  
    #pragma omp parallel for simd reduction(+ : dot)  
    for(i = 0; i < N; i++)  
        dot += a[i] * b[i];  
  
    return dot;  
}
```



# RESULTADOS\*

./4-dot-product.exec 400000000, executou em 6.49 seg.

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

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

4.1 parallel 32 threads	4.2 SIMD 1 thread	4.3 parallel SIMD 32 threads
2.19	5.77	2.11

# EXERCÍCIO 5, PARTE A:

## MM - PARALLEL

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

```
void matrix_mult(double *A, *B, *C, int N){
    int i, j, k;

    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            for(k = 0; k < N; k++){
                C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
        }
    }
}
```

# SOLUÇÃO 5.1, PARTE A: MM - PARALLEL

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

```
void matrix_mult(double *A, *B, *C, int N){
    int i, j, k;

    #pragma omp parallel for private(i, j, k)
    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            for(k = 0; k < N; k++){
                C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
        }
    }
}
```

# EXERCÍCIO 5, PARTE B:

## MM - SIMD

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

```
void matrix_mult(double *A, *B, *C, int N){
    int i, j, k;

    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            for(k = 0; k < N; k++){
                C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
        }
    }
}
```

# ~~SOLUÇÃO 5.2~~, PARTE B:

## MM — SIMD WRONG

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

```
void matrix_mult(double *A, *B, *C, int N){
    int i, j, k;

    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            #pragma vector aligned
            #pragma simd
            for(k = 0; k < N; k++){
                C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
        }
    }
}
```

# EXERCÍCIO 5, PARTE C:

## MM - SIMD

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

```
void matrix_mult(double *A, *B, *C, int N){
    int i, j, k;

    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            #pragma vector aligned
            #pragma simd
            for(k = 0; k < N; k++){
                C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
        }
    }
}
```

# SOLUÇÃO 5.3, PARTE C:

## MM - SIMD

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

```
void matrix_mult(double *A, *B, *C, int N){
    int i, j, k;

    for(i = 0; i < N; i++){
        for(k = 0; k < N; k++){
            #pragma vector aligned
            #pragma simd
            for(j = 0; j < N; j++){
                C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
        }
    }
}
```

# EXERCÍCIO 5, PARTE D:

## MM — PARALLEL SIMD

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

```
void matrix_mult(double *A, *B, *C, int N){
    int i, j, k;

    for(i = 0; i < N; i++){
        for(k = 0; k < N; k++){
            #pragma vector aligned
            #pragma simd
            for(j = 0; j < N; j++){
                C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
        }
    }
}
```



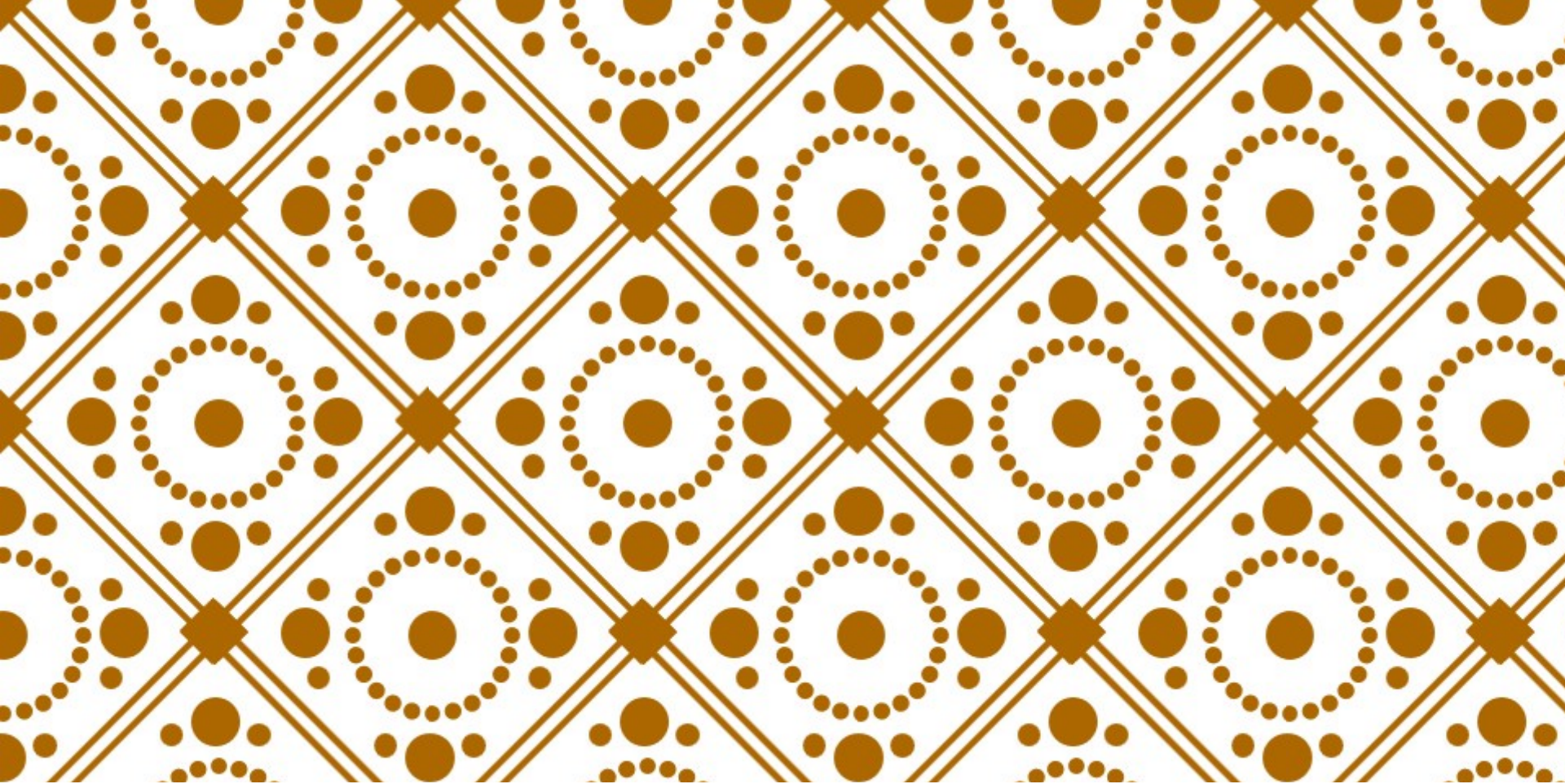
# RESULTADOS\*

./5-matrix-mult.exec 2048, executou em 135.29 seg.

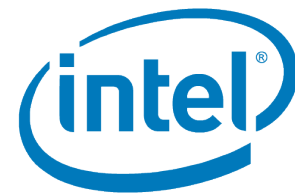
\* 2 x Xeon E5-2640 v2, 8 cores, 2 SMT-cores

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

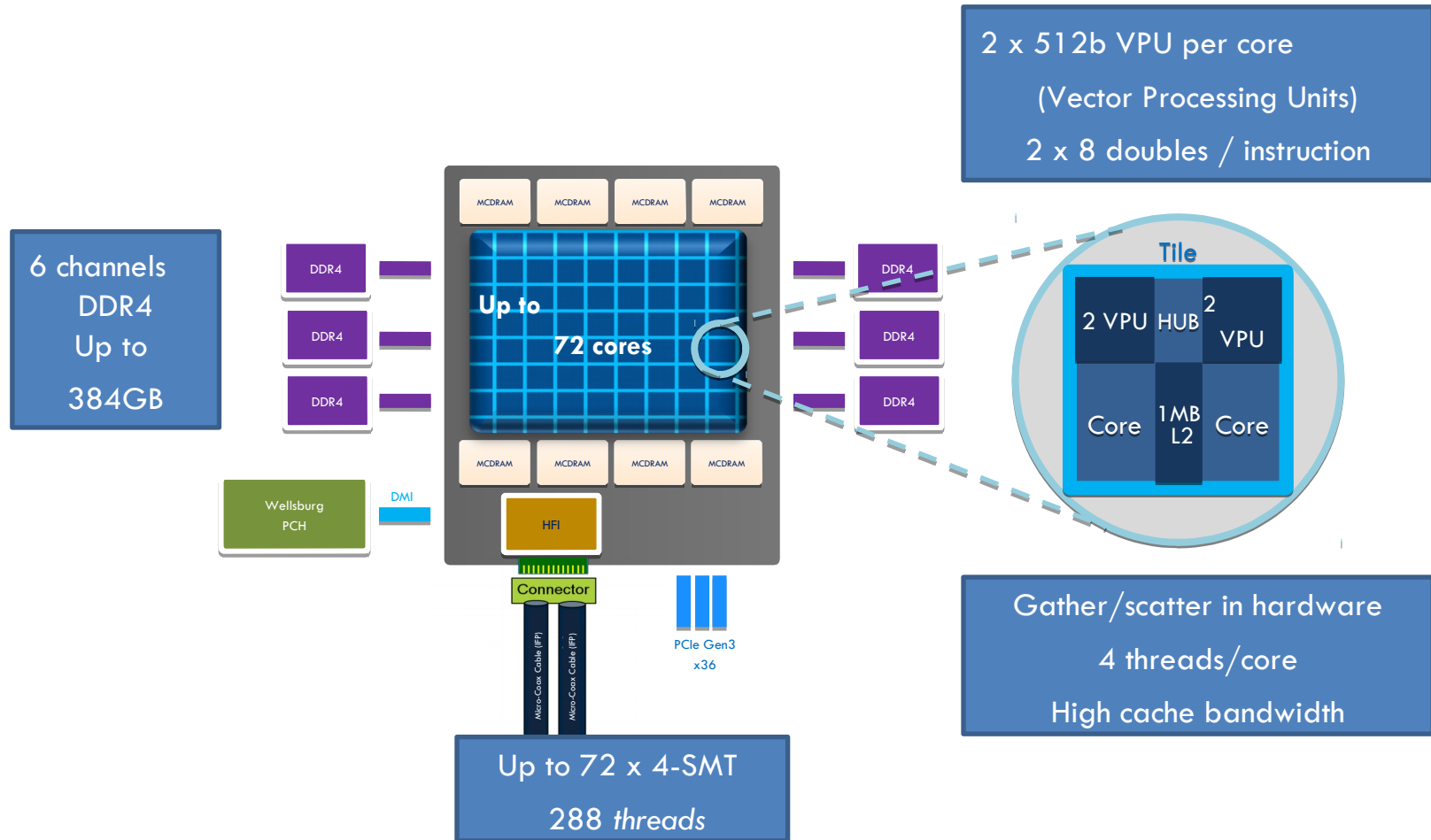
	5.1 parallel	5.2 SIMD <b>WRONG</b>	5.3 SIMD	5.4 parallel SIMD
Time	6.51	130.52	8.89	0.54
Speedup	20.78		15.22	<b>250,54</b>



**INTEL XEON PHI PROCESSOR**

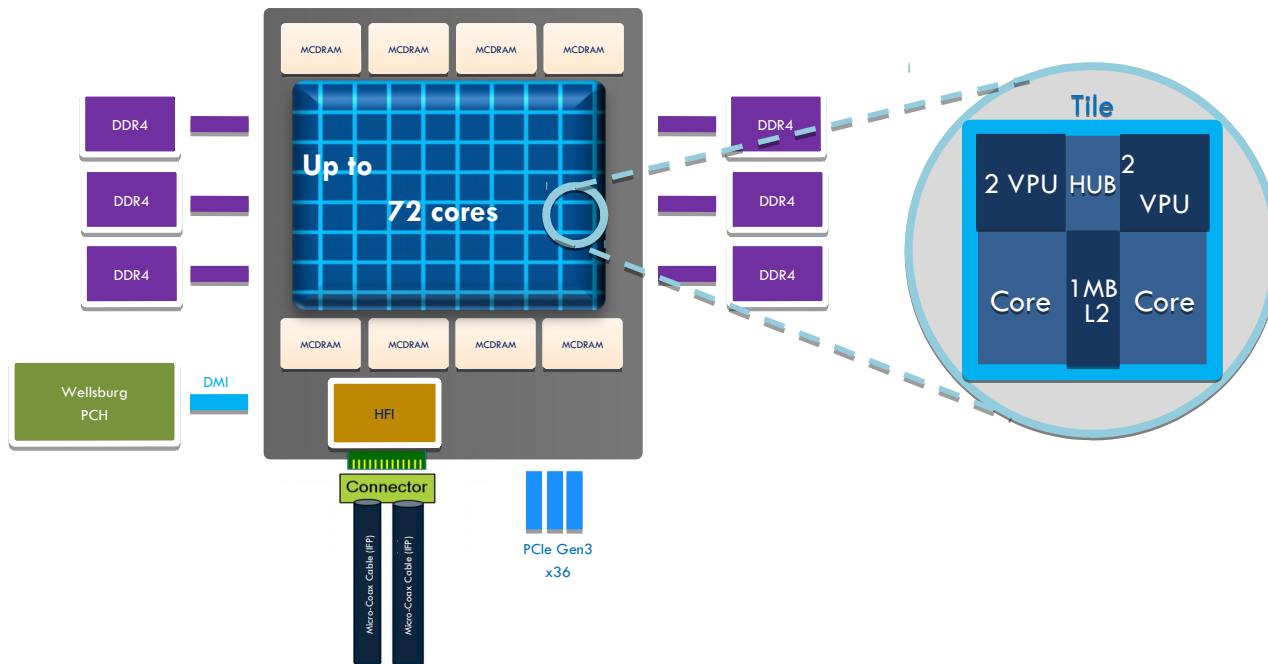


# INTEL XEON PHI PROCESSOR



# INTEL XEON PHI PROCESSOR

FREE TEST DRIVE  
[colfaxresearch.com](http://colfaxresearch.com)



# INTEL XEON PHI COPROCESSOR

Funciona como um complemento para o processador.

PCI Express.

**Possui memória própria.**

Modo nativo.

Recompilar e executar.

```
icc -mmic sum.c; scp a.out mic0:~/; ssh mic0; ./a.out
```

Modo *offload*

Inicia no processador.

Trechos são executados no coprocessador.

Termina no processador.



# DIRETIVAS - OFFLOAD

Indicando região a ser executada no coprocessador

```
#pragma offload target(mic)
{
  for(i = 0; i < N; i++)
    c += a + i;
}
```

Indicando região e vetores a ser copiados

```
#pragma offload target(mic)
inout(a:length(N)) in(b:length(N))
out(c:length(N))
{
  for(i = 0; i < N; i++){
    c[i] = a[i] + b[i];
    a[i] += c[i];
  }
}
```

# DIRETIVAS - OFFLOAD

Offload, paralelismo, vetorização, alinhamento e redução

```
#pragma offload target(mic)
in(v:length(N))
{
    #pragma omp parallel for simd
    align(v : 64) reduction(+ : sum)
    for(i = 0; i < N; i++)
        sum += v[i];
}
```

## EXERCÍCIO 6, PARTE A: MM — PARALLEL XEON PHI OFFLOAD

```
cd matrix/MultiplicationOffload/ make ./matrixMultiplicationOffload.exec <elementos>
```

```
void matrix_mult(double *A, *B, *C, int N){  
    int i, j, k;  
  
    for(i = 0; i < N; i++)  
        for(j = 0; j < N; j++){  
  
            for(k = 0; k < N; k++)  
                C[i * N + j] += A[i * N + k] * B[k * N + j];  
        }  
}
```



## EXERCÍCIO 6, PARTE B: MM — PARALLEL SIMD XEON PHI OFFLOAD

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

```
void matrix_mult(double *A, *B, *C, int N){
    int i, j, k;

    #pragma offload target(mic) in(A:length(N*N))
    in(B:length(N*N)) out(C:length(N*N))
    {
        #pragma omp parallel for default(shared) private(i, j, k)
        for(i = 0; i < N; i++)
            for(j = 0; j < N; j++){

                for(k = 0; k < N; k++)
                    C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
    }
}
```

# RESULTADOS\*

A multiplicação de matrizes sequencial com entrada 1024 x 1024, executou em 143.75 seg.

\* Xeon Phi 3120P – Launched in 2013

*57 cores x 4 SMT = 228 Threads*

	parallel Offload	parallel SIMD Offload
Time	1.80	1.30
Speedup	<b>79.86</b>	<b>110.58</b>

# TESTANDO O AMBIENTE REMOTO

Login remoto a sistemas de computadores

```
ssh ufpelintel@
```

```
senha: ufpelintel
```

```
conectaphikn1
```

Copie os exercícios

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

```
cd seu-nome/
```

Trabalhe dentro de seu diretório.

## SOLUÇÃO 6.1, PARTE A: MM — PARALLEL XEON PHI OFFLOAD

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

```
void matrix_mult(double *A, *B, *C, int N){  
    int i, j,
```

```
    #pragma omp for  
    for(int(B:length
```

```
{
```

```
    #pragma omp for
```

```
    for(i = 0
```

```
        for(j =
```

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

```
                C[i * N + j] += A[i * N + k] * B[k * N + j];
```

```
            }
```

```
        }
```

```
    }
```

Agora no Intel Xeon Phi KNL.

**Precisamos da Versão Offload?**

```
, k)
```

## SOLUÇÃO 6.1, PARTE A: MM — PARALLEL XEON PHI OFFLOAD

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

```
void matrix_mult(double *A, *B, *C, int N){  
    int i, j,
```

Agora no Intel Xeon Phi KNL.

**Precisamos da Versão Offload?**

*Na versão KNL podemos usar o mesmo código*

*que roda em um processador Xeon normal*

*(sem diretivas offload)*

```
    #pragma omp for  
    for(int(B: length
```

```
    {
```

```
        #pragma omp for
```

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

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

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

```
                    C[i * N + j] += A[i * N + k] * B[k * N + j];
```

```
            }
```

```
        }
```

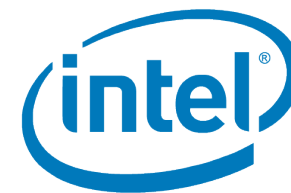
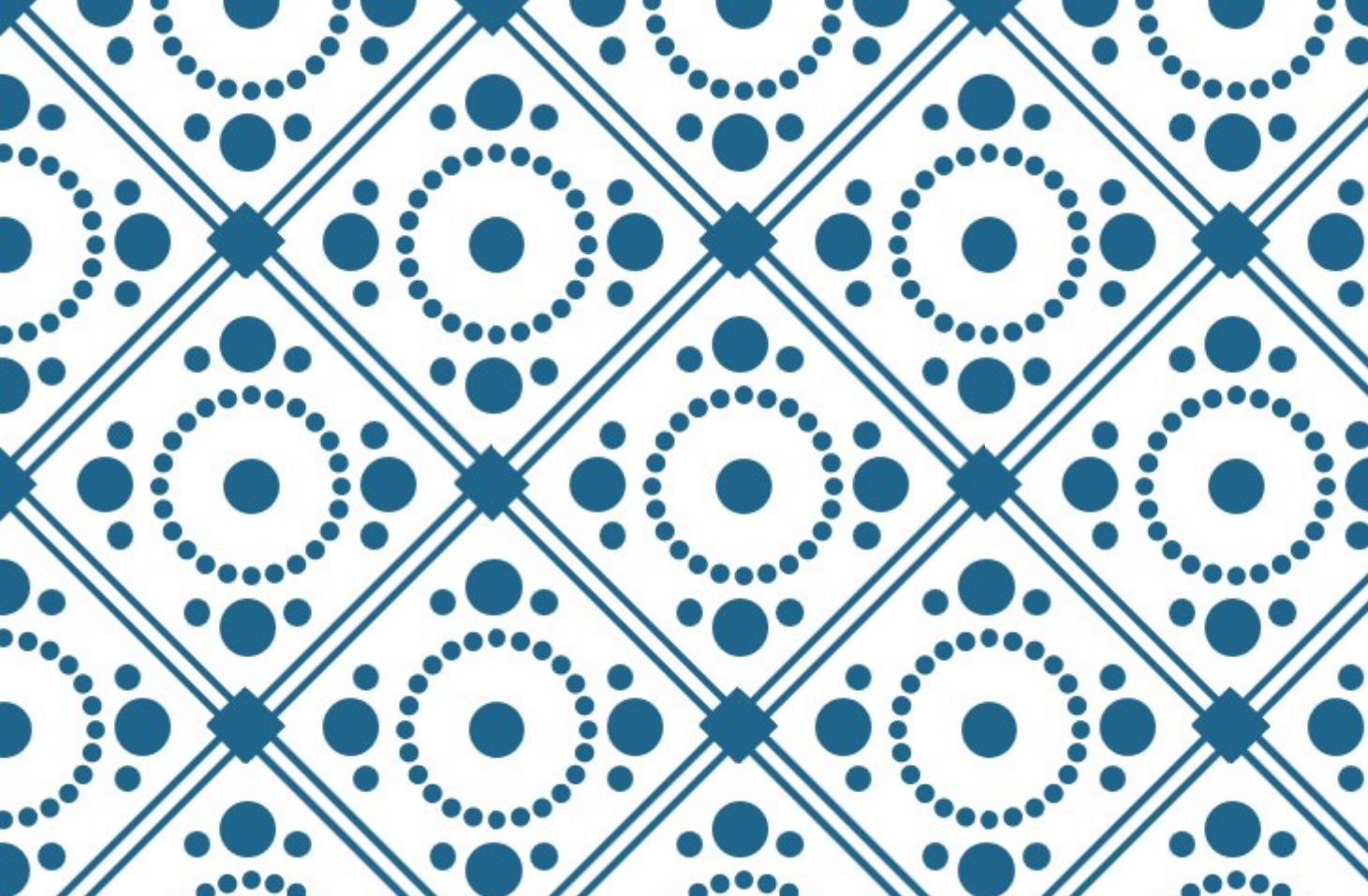
```
    }
```

# EXERCÍCIO - KNL: MM — PARALLEL SIMD

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

```
void matrix_mult(double *A, *B, *C, int N){
    int i, j, k;

    #pragma omp parallel for private(i, j, k)
    for(i = 0; i < N; i++){
        for(k = 0; k < N; k++){
            #pragma vector aligned
            #pragma simd
            for(j = 0; j < N; j++){
                C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
        }
    }
}
```



# INTEL MODERN CODE PARTNER

## UFPEL

