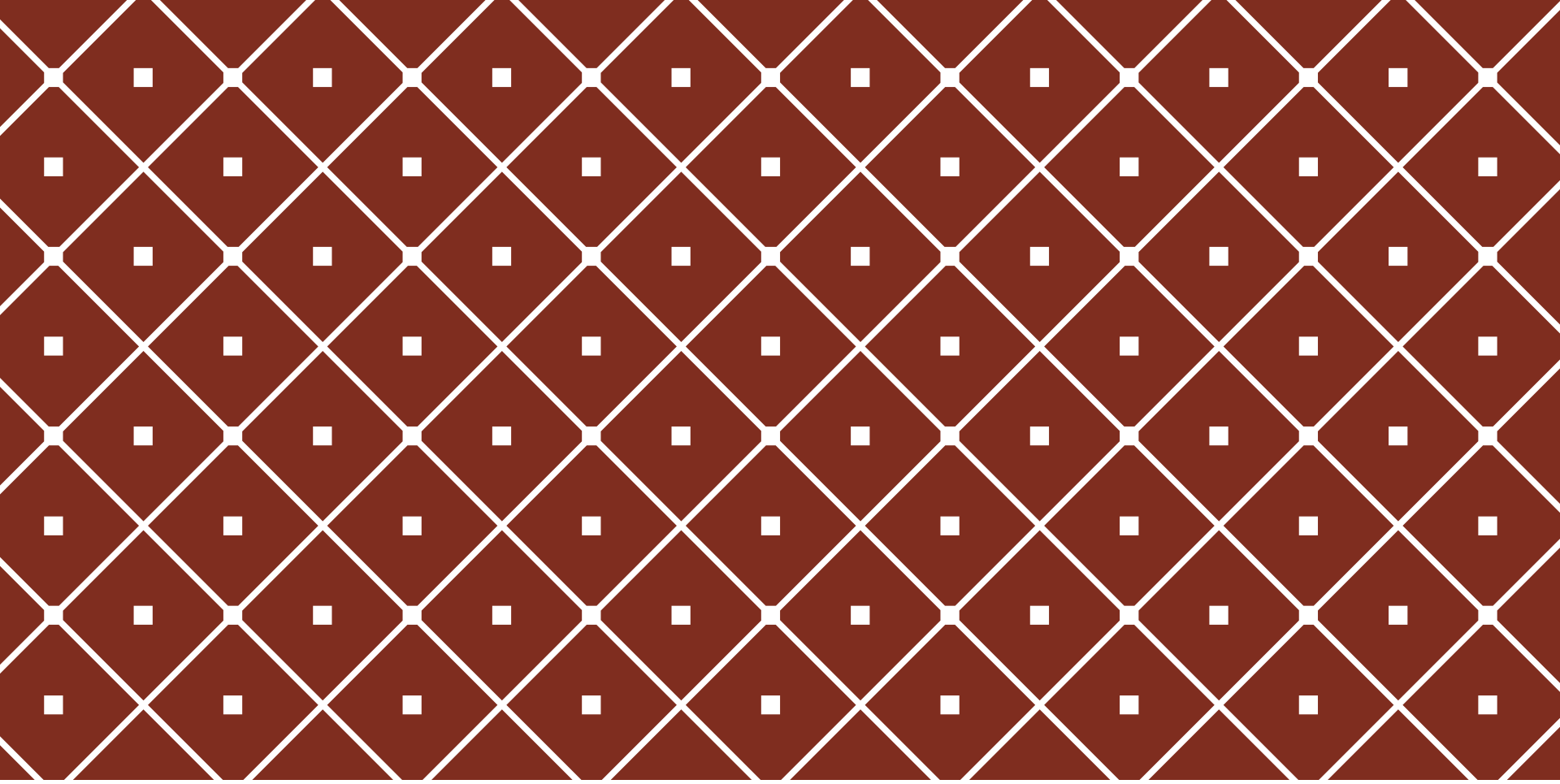
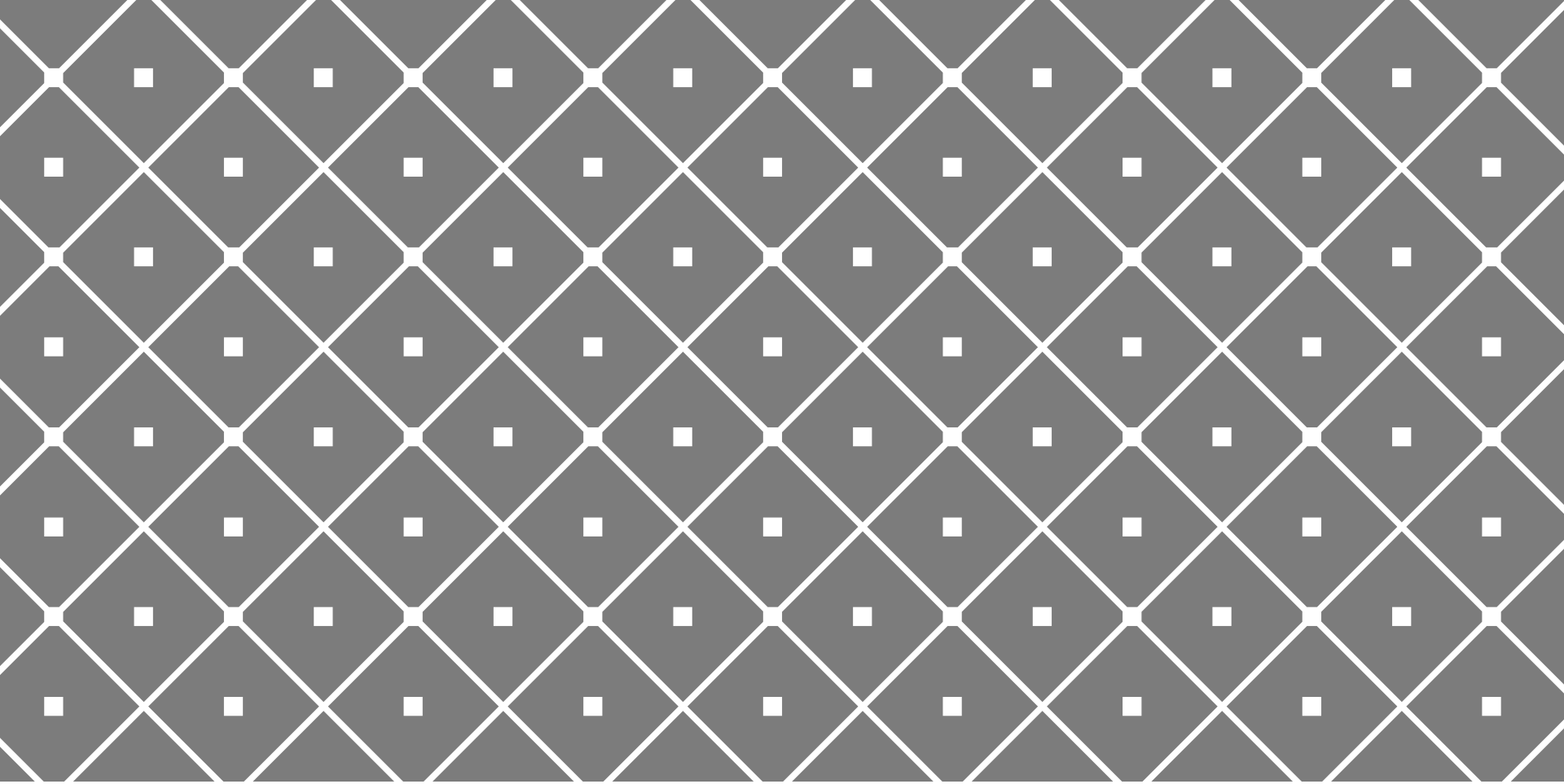


PROGRAMAÇÃO PARALELA OPENMP – AULA 04

Marco A. Zanata Alves

Slides baseados no material de Tim Mattson - Intel Corp. (2010)
A "Hands-on" Introduction to OpenMP *



PRÁTICA DE CONHECIMENTOS... LISTA ENCADEADA EM OPENMP

AS PRINCIPAIS CONSTRUÇÕES OPENMP VISTAS ATÉ AGORA

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

Para criar um conjunto de threads

- #pragma omp parallel

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 virgula

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.

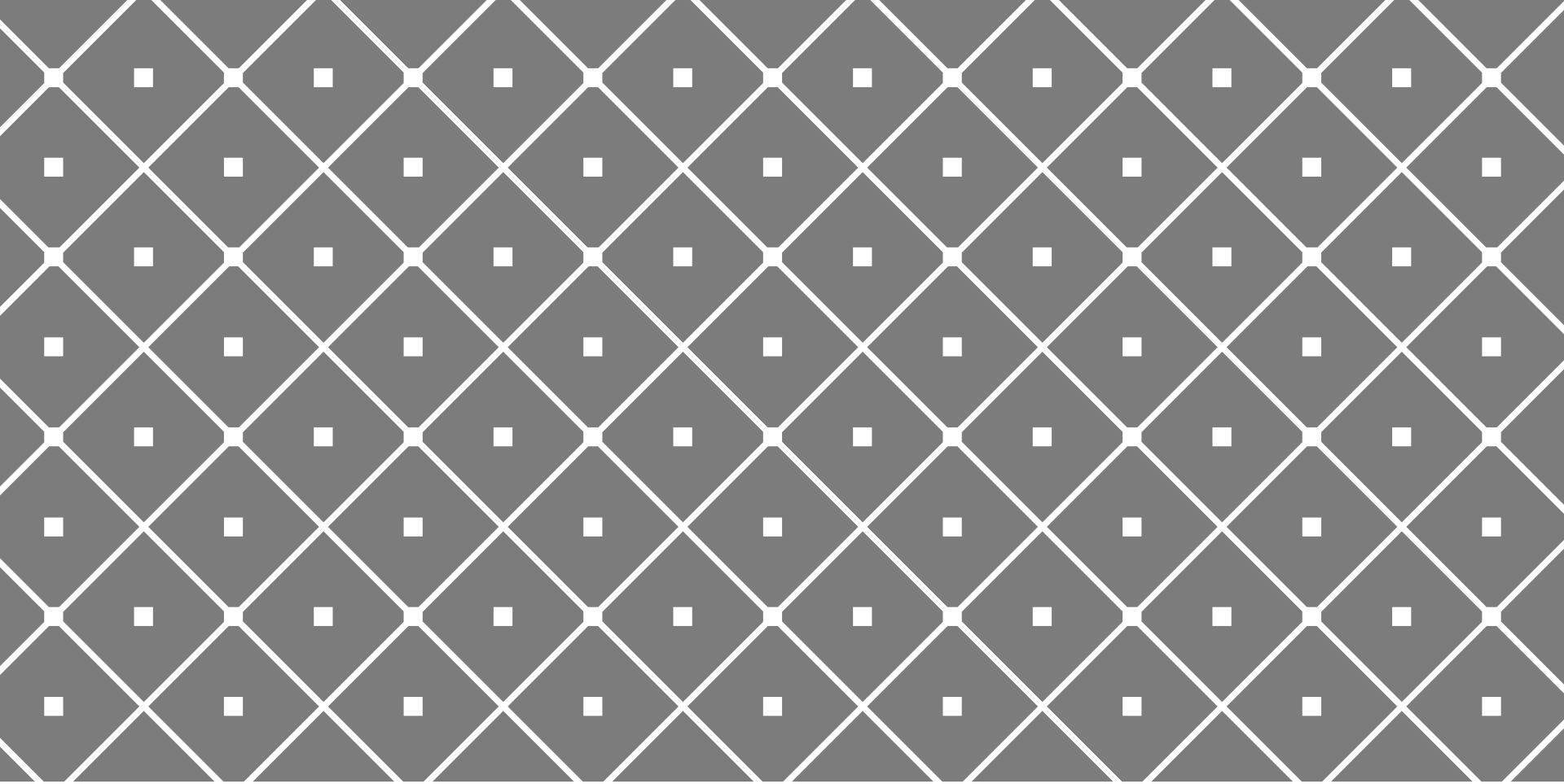
EXERCÍCIO 7: LISTA ENCADEADA (MODO DIFÍCIL)

Considere o programa linked.c

- Atravessa uma lista encadeada computando uma sequencia 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.



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

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]);
}
```

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

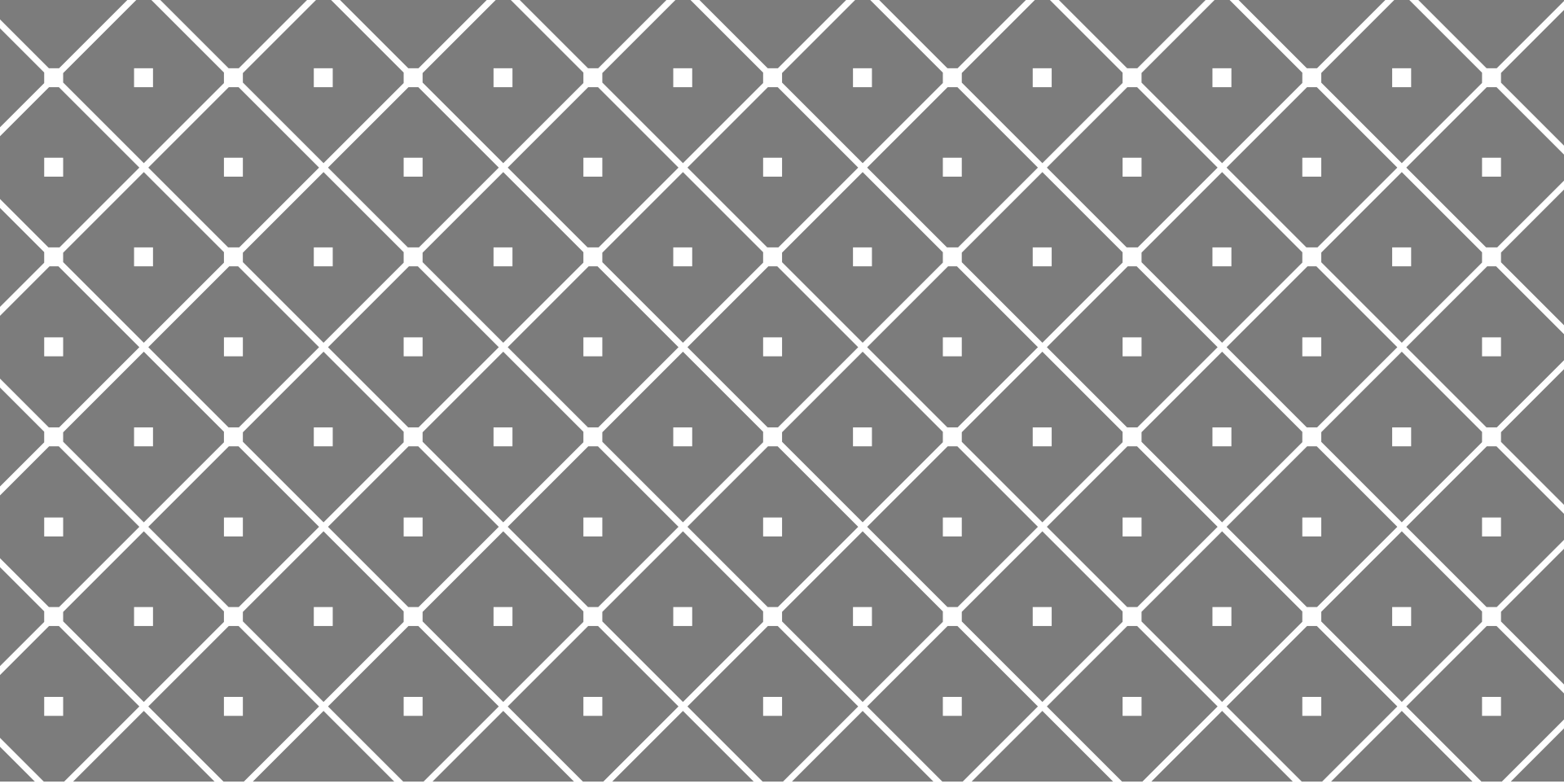
CONCLUSÕES

Somos capazes de paralelizar listas encadeadas …
mas isso foi feio!

Precisamos de 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.

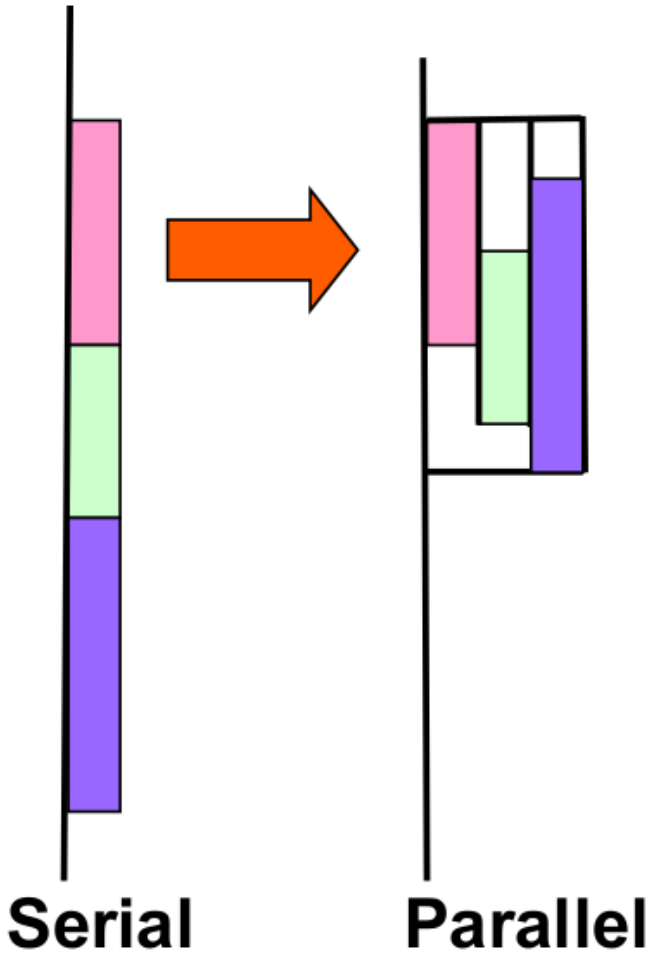
Tasks são compostas de:

- **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



DEFINIÇÕES

Construção da Task – Diretiva + bloco estruturado

Task – O pacote de código e instruções para alocar os dados criados quando uma thread encontra uma construção task.

Região da Task – A sequencia dinâmica de instruções produzidas para executar uma task por uma thread

QUANDO PODEMOS GARANTIR QUE AS TAKS 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.

```
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;
}
```

Exemplo de divisão e conquista

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

O que está errado aqui?

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!=null; 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!=null; 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**.

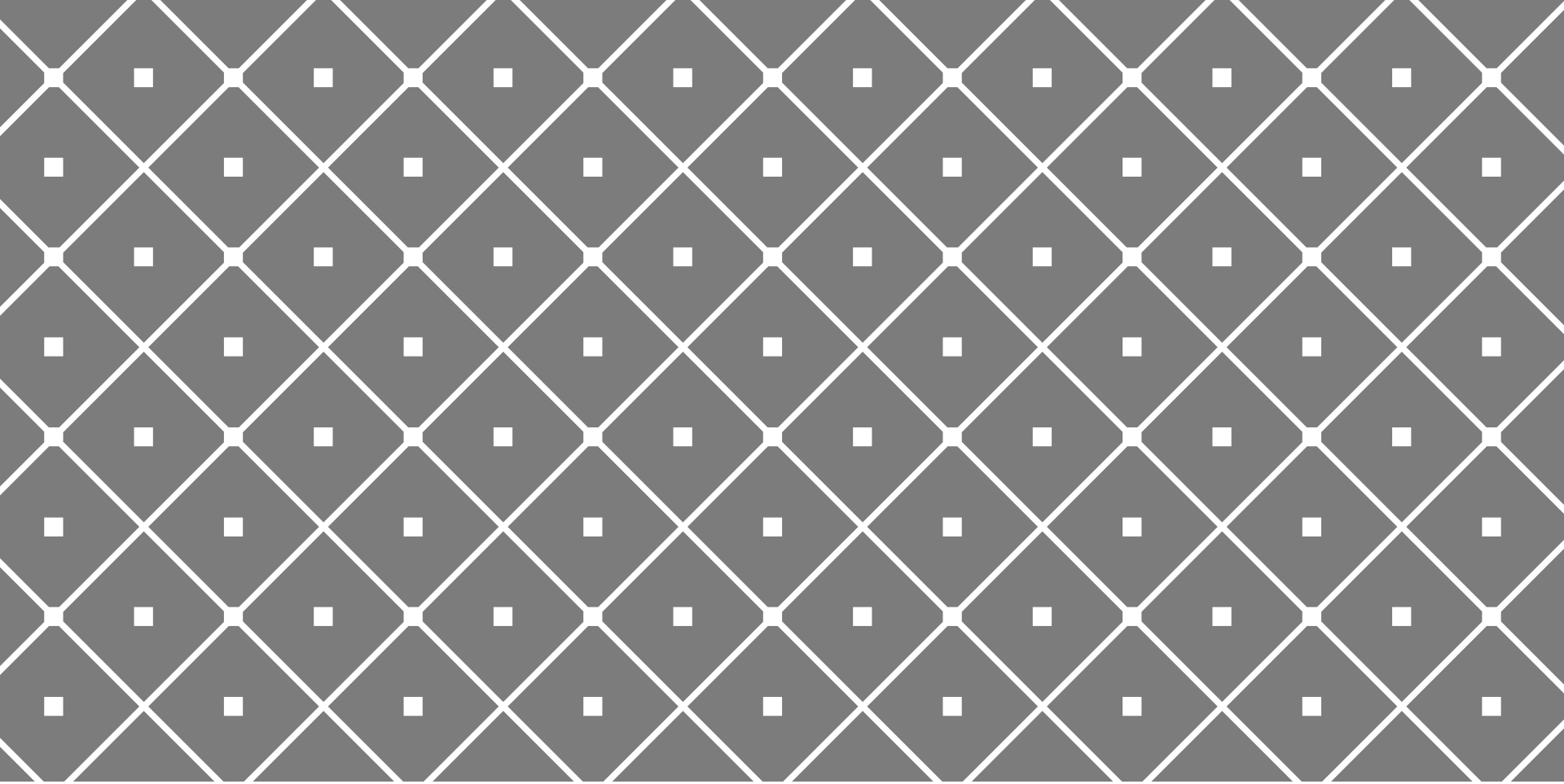
Demais variáveis declaradas nas rotinas chamadas **serão privadas**.

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*.



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 threads 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

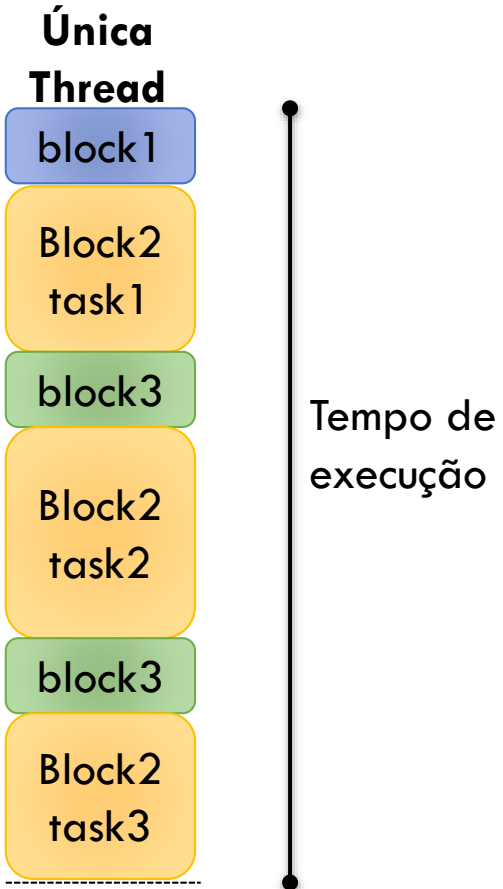
4. As threads aguardando na barreira executam as tasks.

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.

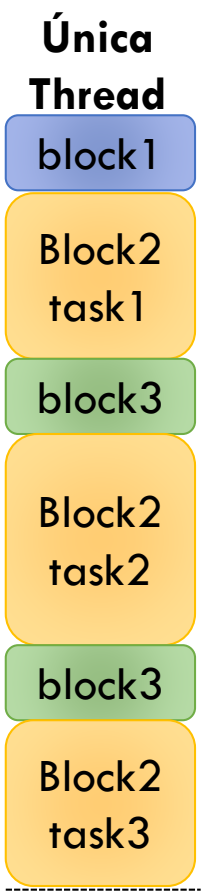
```
{

  {
//block 1
    node * p = head;
    while (p) {
// block 2

        process(p);
//block 3
      p = p->next;
    }
  }
}
```

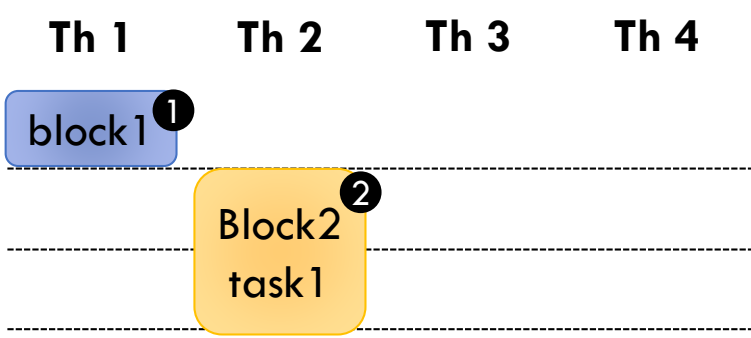


EXECUÇÃO DE TASKS

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;
    }
  }
}
```

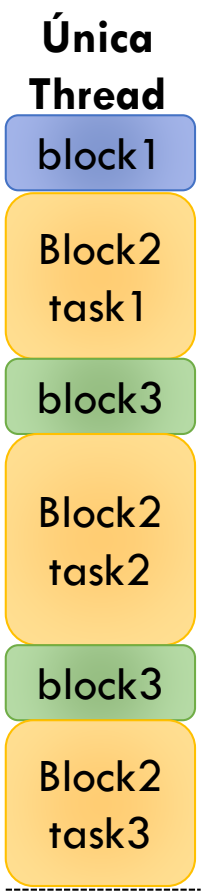


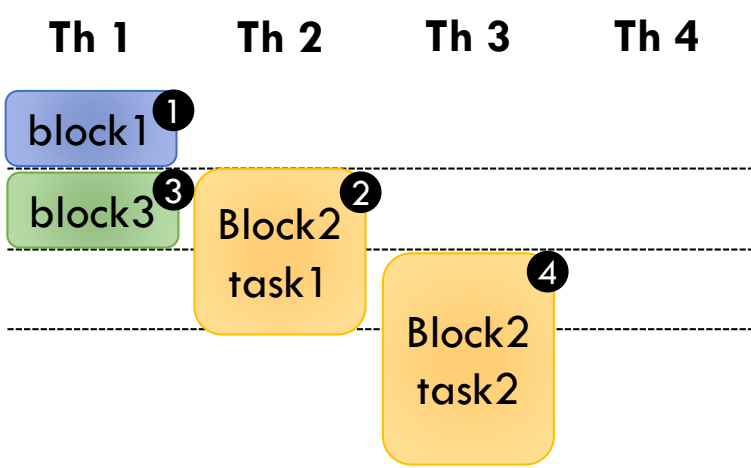


EXECUÇÃO DE TASKS

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;
    }
  }
}
```

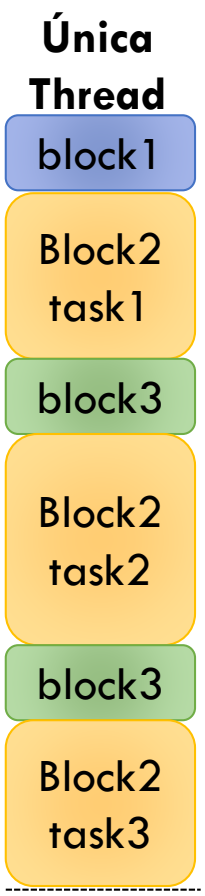


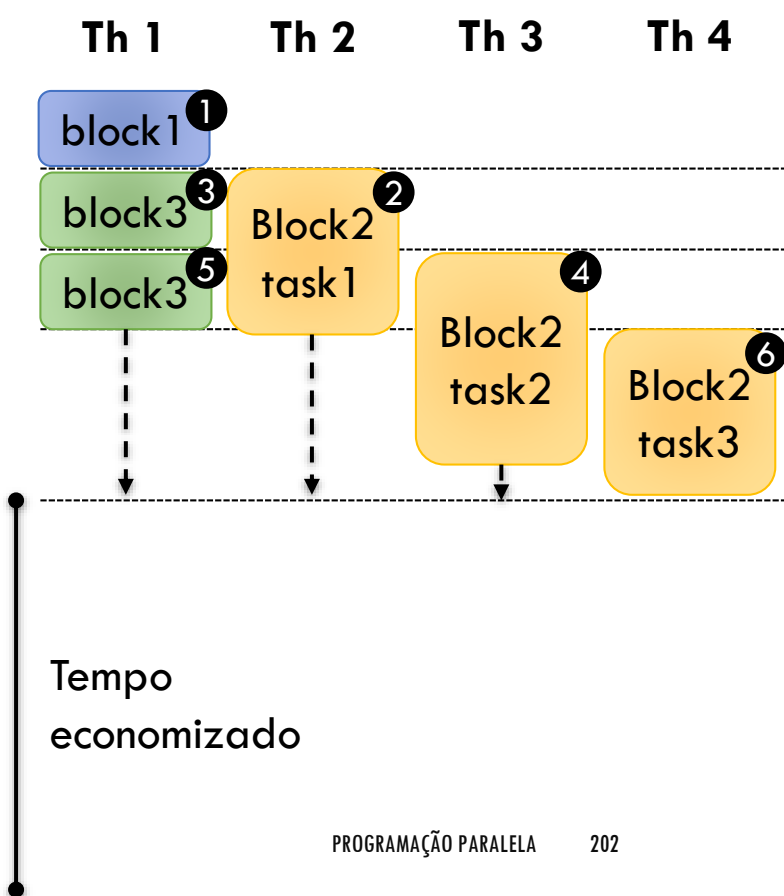


EXECUÇÃO DE TASKS

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;
    }
  }
}
```

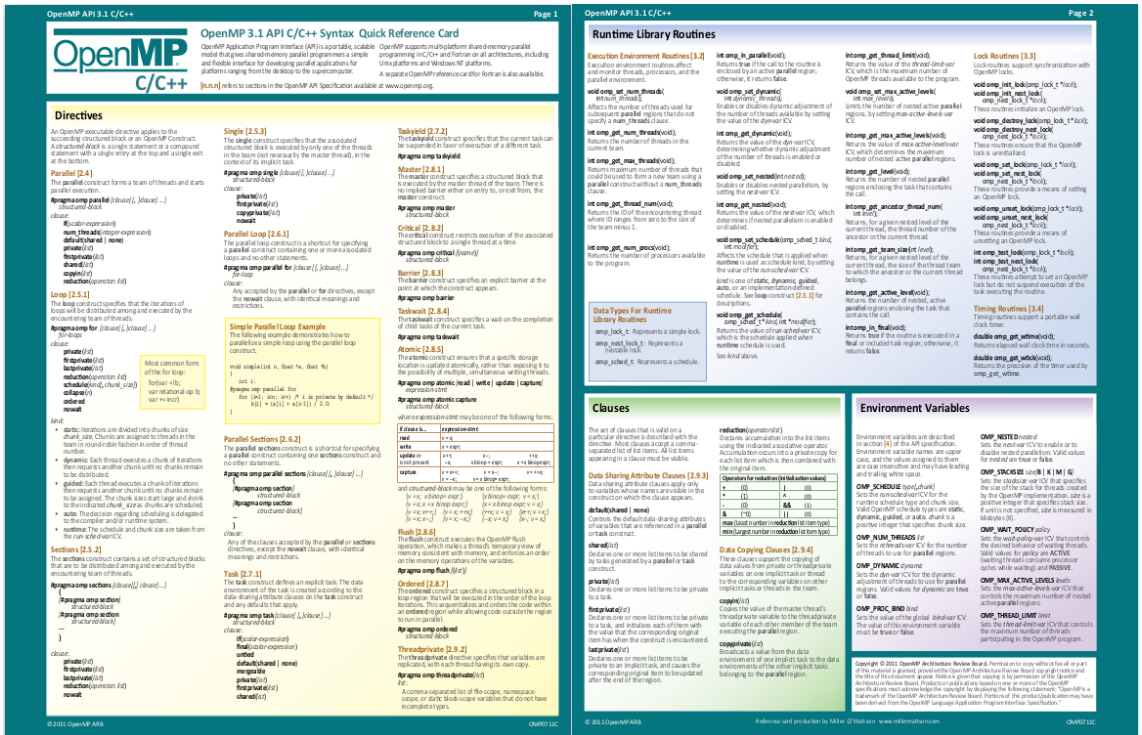




O CARTÃO DE REFERÊNCIA OPENMP

Duas páginas de resumo de todas as construções OpenMP … não escreva código sem isso. :-)

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



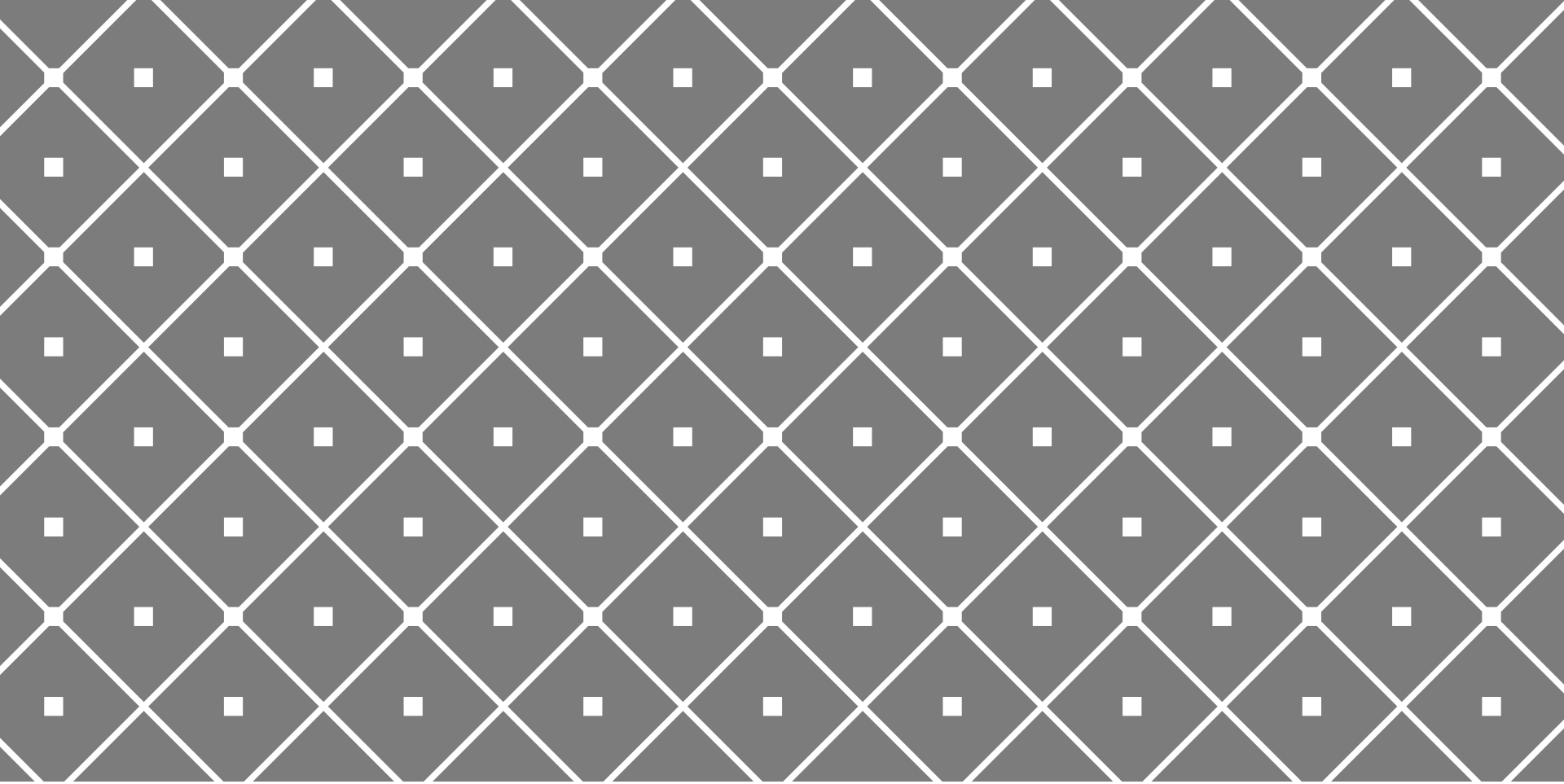
EXERCÍCIO 8: TASKS EM OPENMP

Considere o programa linked.c

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

Paralelize esse programa usando **tasks**.

Compare a solução obtida com a versão sem tasks.



NOVIDADES DAS VERSÕES OPENMP 4 E 5

NOVAS CONSTRUÇÕES E CLÁUSULAS OPENMP DA VERSÃO 4.0 / 5.0

Construções

simd

target

teams

taskgroup

taskloop

cancel

...

Cláusulas

requires (app)

proc_bind (app)

depend (task/target)

hint (critical)

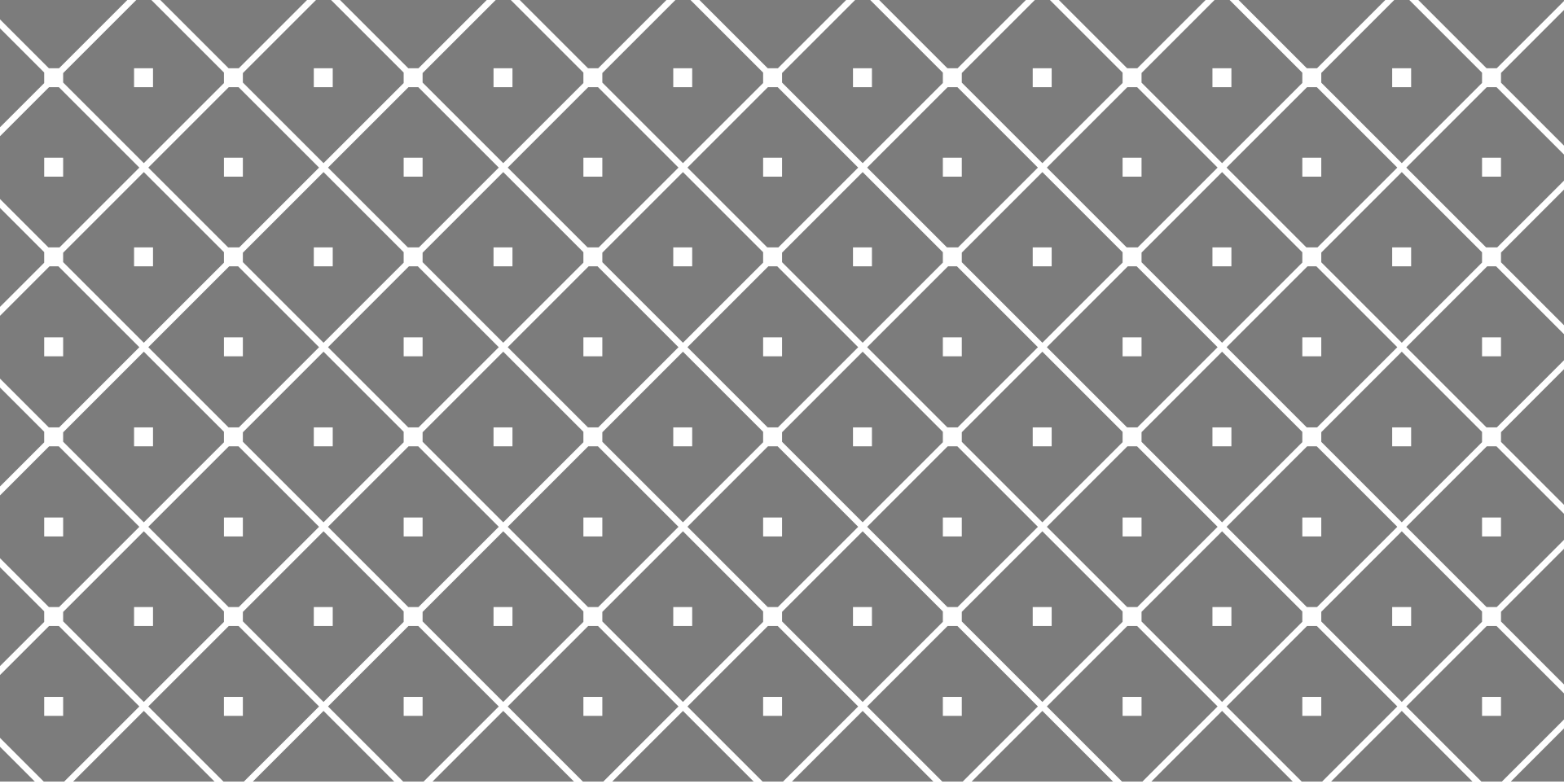
ordered (loop)

linear (loop)

priority (task)

affinity (task)

...



SIMD – SINGLE INSTRUCTION MULTIPLE DATA

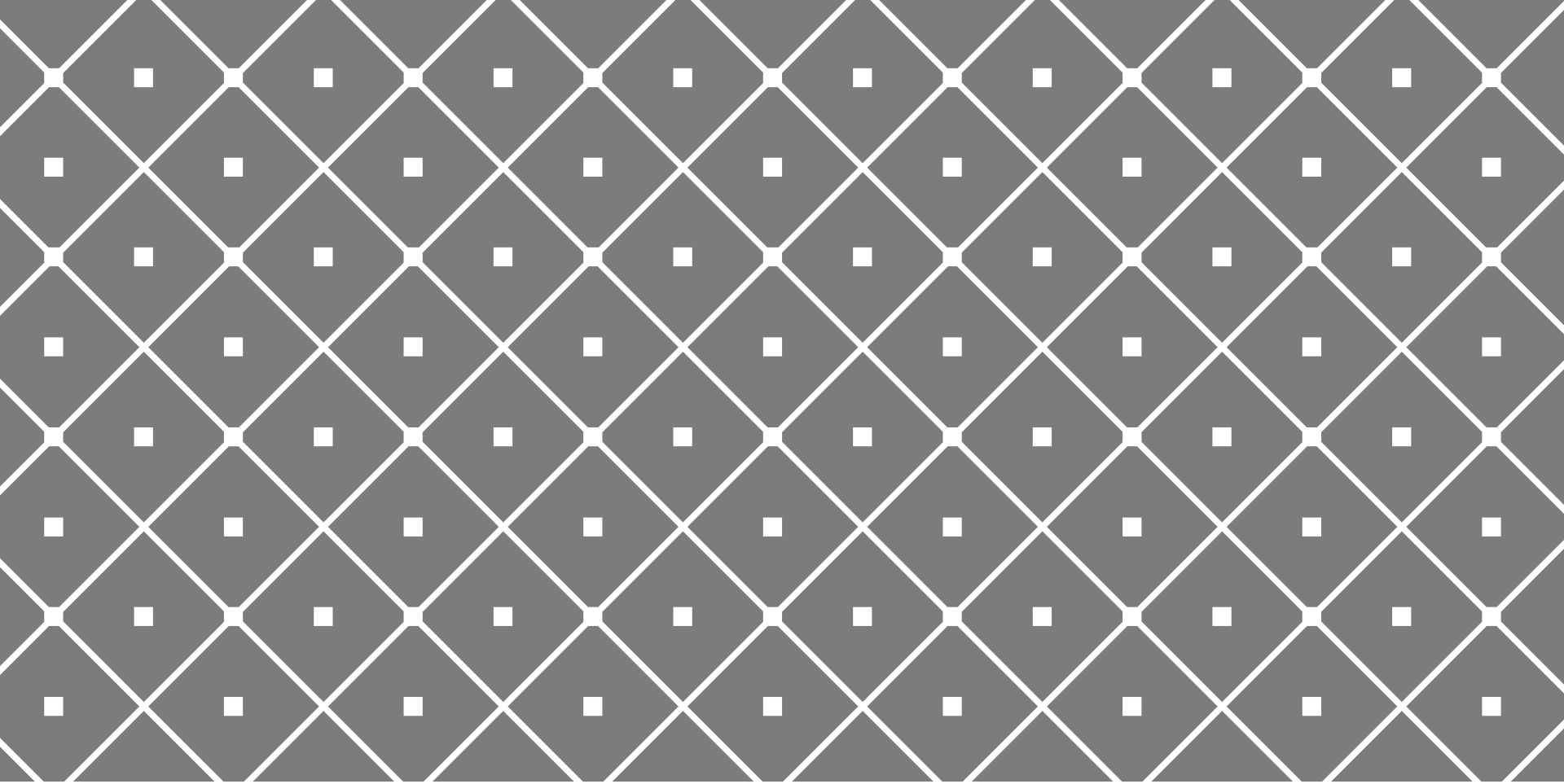
SIMD

Única Instrução, múltiplos dados (SIMD) é uma forma de execução paralela

A mesma operação é realizada em vários elementos de dados independentemente, em unidades de processamento vetorial de hardware (VPU), também chamadas de unidades SIMD.

A adição de dois vetores para formar um terceiro vetor é uma operação SIMD.

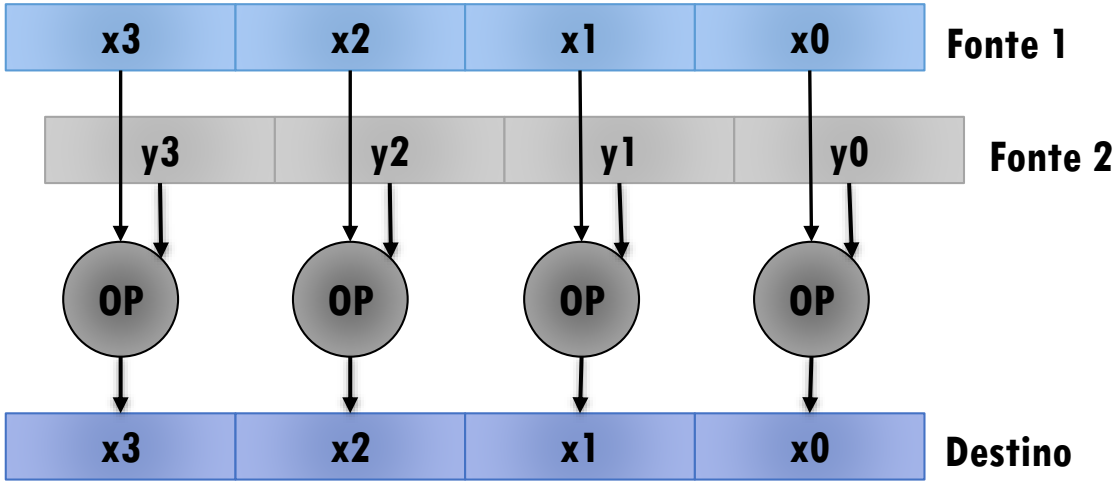
Muitos processadores possuem unidades SIMD (vetor) que podem realizar simultaneamente 2, 4, 8 ou mais execuções da mesma operação (por uma única unidade SIMD).



INSTRUÇÕES SIMD

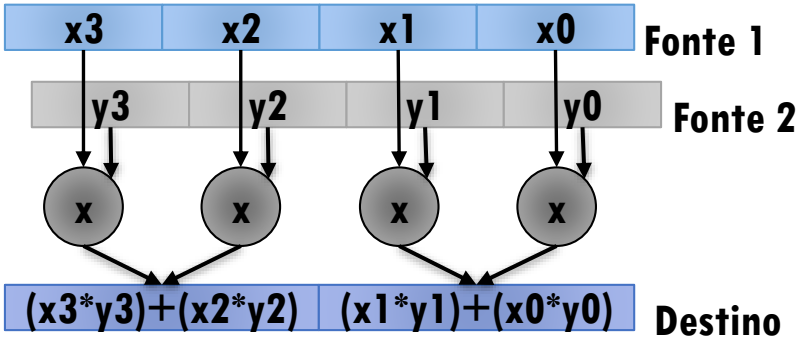
INSTRUÇÕES SIMD

Máquinas SIMD reais têm uma mistura de instruções SISD e SIMD

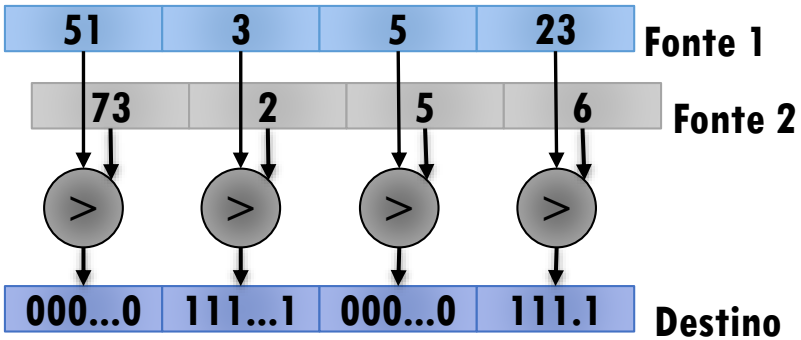


EXEMPLOS

Mult/Add



Compares



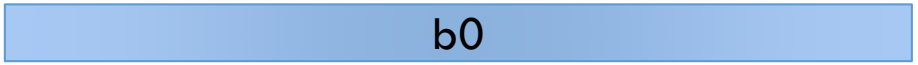
INTEL MMX – 8X64 BITS REGISTRADORES

(8bits x 8) Packed Bytes

(16bits x 4) Packed Words

(32bits x 2) Packed Doublewords

(64bits x 1) Packed Quadword



INTEL SSE - 8X128 BITS REGISTRADORES

128-Bit Packed Double-Precision FP

128-Bit Packed Byte Integers

128-Bit Packed Word Integers

128-Bit Packed Doubleword Integers

128-Bit Packed Quadword Integers

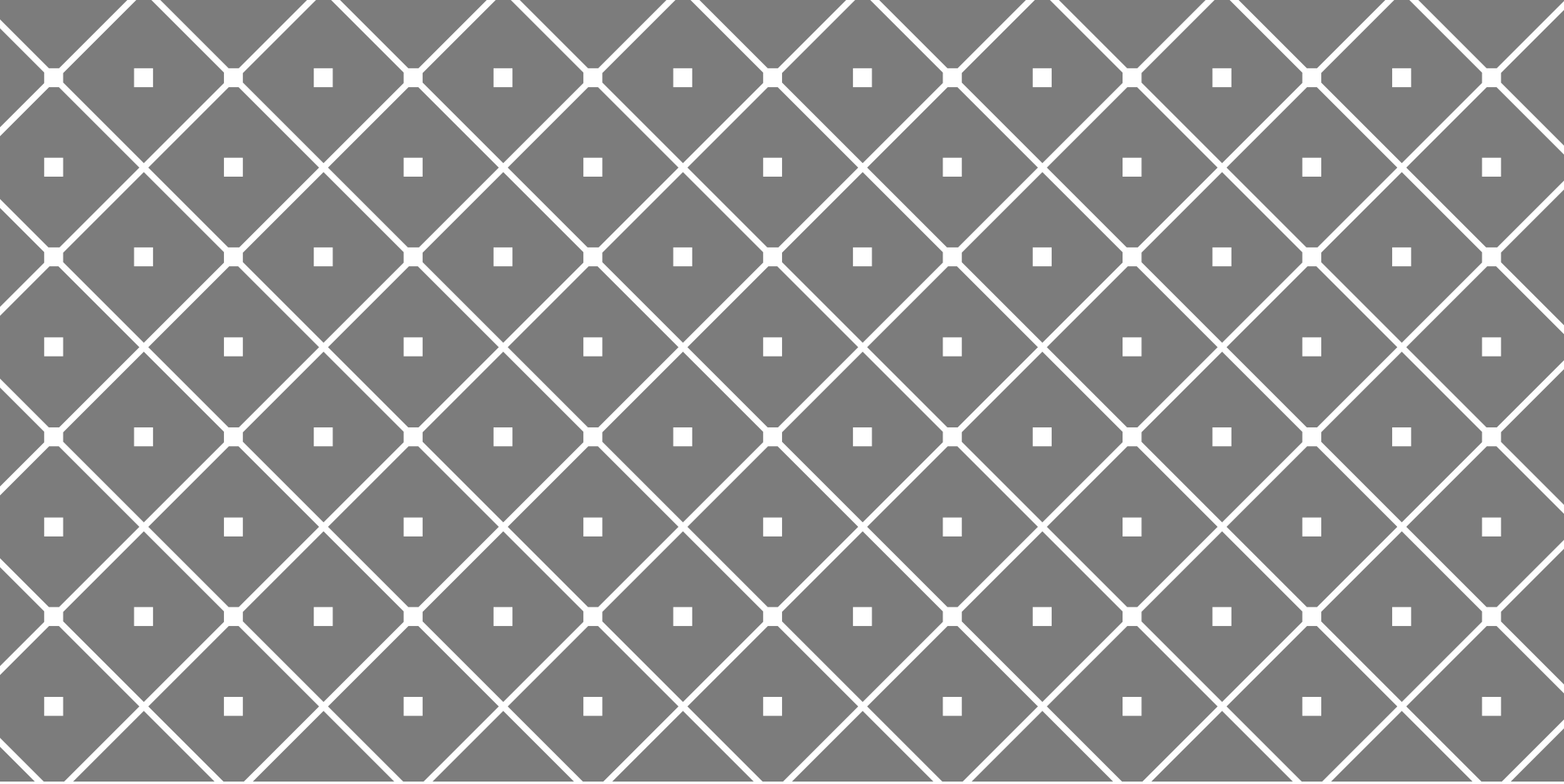
REGISTRADORES X86 (INTEL/AMD)

Extensão Multimedia e
Registradores de Ponto-Flutuante

MM0/ST0
MM1/ST1
MM2/ST2
MM3/ST3
MM4/ST4
MM5/ST5
MM6/ST6
MM7/ST7

Fluxo SIMD
Registradores de Extensão SSE

XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15



EXEMPLO COM SIMD

EXEMPLO COM SIMD

```
void star( double *a, double *b, double *c, int n, int *ioff )
{
  int i;
  #pragma omp simd
  for ( i = 0; i < n; i++ )
    a[i] *= b[i] * c[i+ *ioff];
}
```

A construção simd garante ao compilador que o loop pode ser vetorizado

EXEMPLO COM SIMD

```
void star( double *a, double *b, double *c, int n, int *ioff )
{
  int i;
  #pragma omp simd
  for ( i = 0; i < n; i+=4 )
    a[i+0] *= b[i+0] * c[i+0 + *ioff];
    a[i+1] *= b[i+1] * c[i+1 + *ioff];
    a[i+2] *= b[i+2] * c[i+2 + *ioff];
    a[i+3] *= b[i+3] * c[i+3 + *ioff];
}
```

A vetorização irá primeiro desenrolar o laço algumas vezes...

EXEMPLO COM SIMD

```
void star( double *a, double *b, double *c, int n, int *ioff )
{
  int i;
  #pragma omp simd
  for ( i = 0; i < n; i+=4 )
    a[i+0] *= b[i+0] * c[i+0 + *ioff];
    a[i+1] *= b[i+1] * c[i+1 + *ioff];
    a[i+2] *= b[i+2] * c[i+2 + *ioff];
    a[i+3] *= b[i+3] * c[i+3 + *ioff];
}
```

... Depois poderá vetorizar

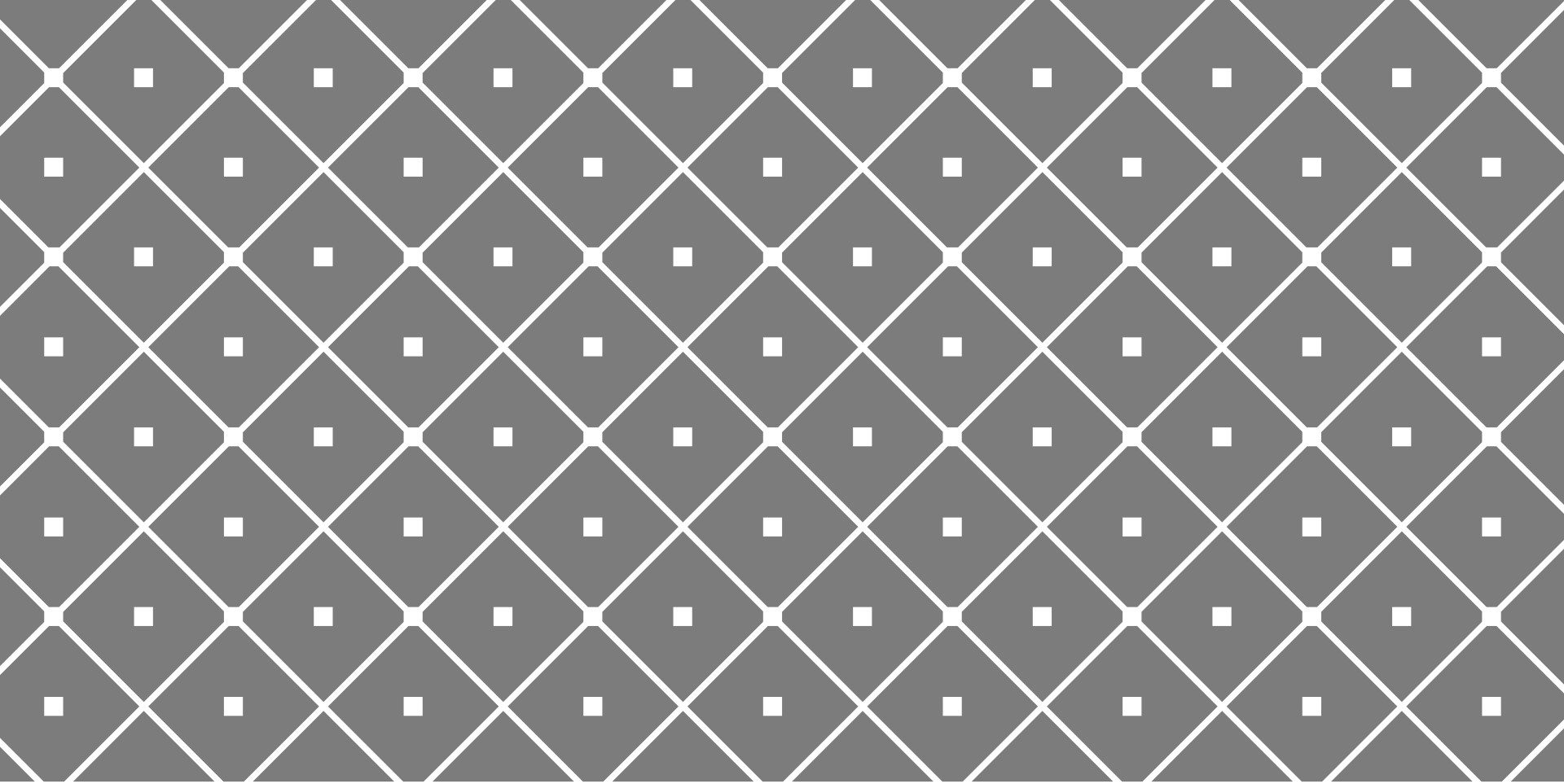
EXEMPLO COM SIMD E REDUCTION

```
double work( double *a, double *b, int n )
{
  int i;
  double tmp, sum;
  sum = 0.0;
  #pragma omp simd private(tmp) reduction(+:sum)
  for (i = 0; i < n; i++) {
    tmp = a[i] + b[i];
    sum += tmp;
  }
  return sum;
}
```

Podemos usar a construção reduction da mesma forma que usamos na paralelização de laços

EXEMPLO COM SIMD PARALELO

```
void work( double **a, double **b, double **c, int n )
{
  int i, j;
  double tmp;
  #pragma omp for simd collapse(2) private(tmp)
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      tmp = a[i][j] + b[i][j];
      c[i][j] = tmp;
    }
  }
}
```



CONSTRUÇÃO TASKLOOP

TASKLOOP

A construção taskloop é uma construção geradora de tasks.

Quando um thread encontra uma construção de taskloop, a construção particiona as iterações dos loops associados em tasks explícitas para execução paralela.

O ambiente de dados de cada tarefa gerada é criado de acordo com as cláusulas do atributo de compartilhamento de dados na construção do taskloop, ICVs de ambiente por dados e quaisquer padrões aplicáveis.

A ordem de criação das tasks do loop não é especificada.

Os programas que dependem de qualquer ordem de execução das iterações do loop lógico não estão em conformidade.

ARMADILHA

```
#include <stdio.h>
#define T 16
#define N 1024
void main() {
  int x1 = 0, x2 = 0;
  #pragma omp parallel shared(x1,x2) num_threads(T)
  {
    #pragma omp taskloop
    for (int i = 0; i < N; ++i) {
      #pragma omp atomic
        x1++;
    }
  }
  printf("x1 = %d\n", x1);
}
```

Qual o valor impresso?

ARMADILHA

```
#include <stdio.h>
#define T 16
#define N 1024
void main() {
  int x1 = 0, x2 = 0;
  #pragma omp parallel shared(x1,x2) num_threads(T)
  {
    #pragma omp taskloop
    for (int i = 0; i < N; ++i) {
      #pragma omp atomic
        x1++;
    }
  }
  printf("x1 = %d\n", x1);
}
```

Qual o valor impresso?

N*T
(ou seja, cada thread criou N tasks)

ARMADILHA

```
#include <stdio.h>
#define T 16
#define N 1024
void main() {
  int x1 = 0, x2 = 0;
  #pragma omp parallel shared(x1,x2) num_threads(T)
  {
    #pragma omp single
    #pragma omp taskloop
    for (int i = 0; i < N; ++i) {
      #pragma omp atomic
        x1++;
    }
  }
  printf("x1 = %d\n", x1);
}
```

Qual o valor impresso?

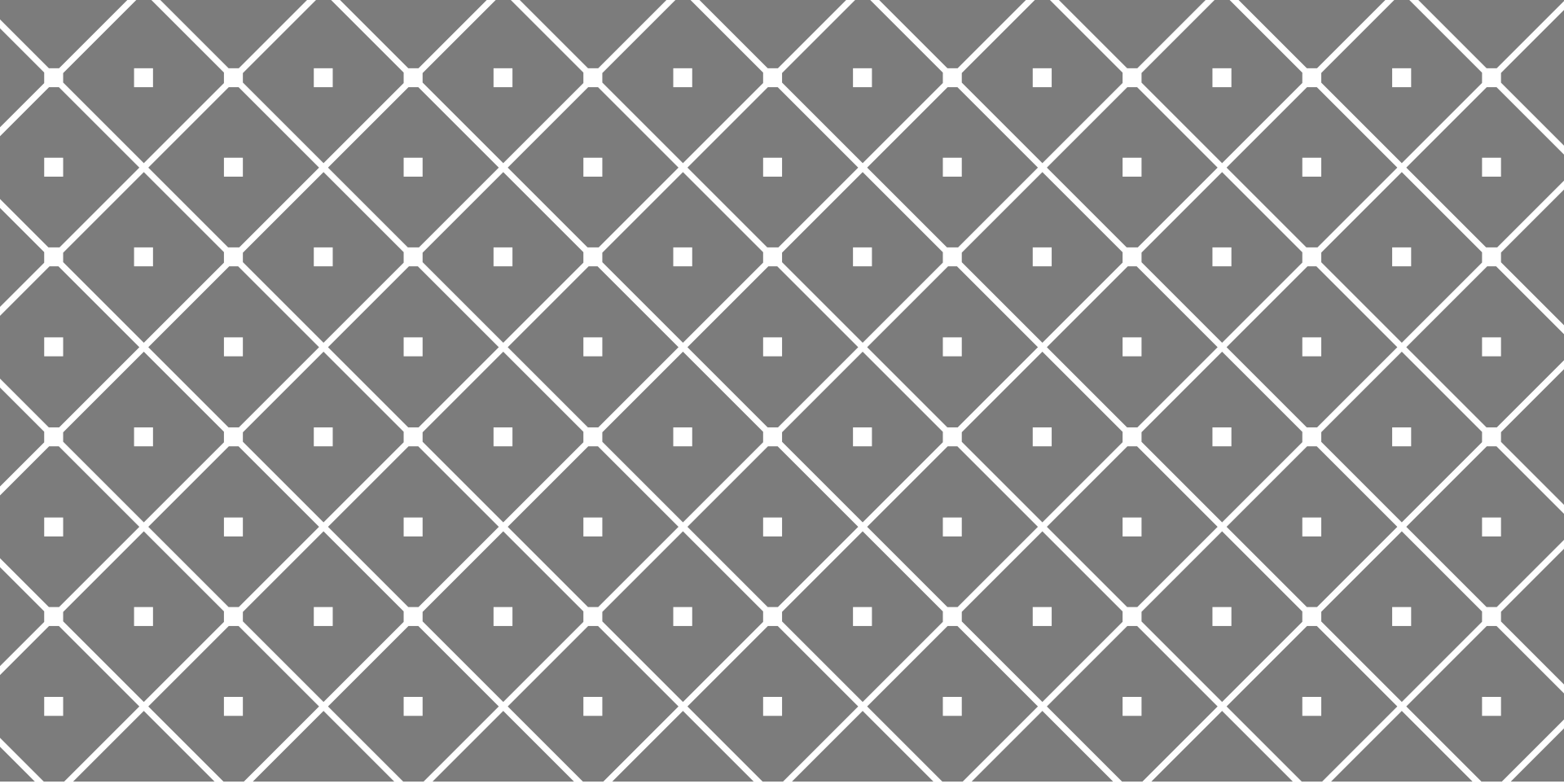
ARMADILHA

```
#include <stdio.h>
#define T 16
#define N 1024
void main() {
  int x1 = 0, x2 = 0;
  #pragma omp parallel shared(x1,x2) num_threads(T)
  {
    #pragma omp single
    #pragma omp taskloop
    for (int i = 0; i < N; ++i) {
      #pragma omp atomic
        x1++;
    }
  }
  printf("x1 = %d\n", x1);
}
```

Qual o valor impresso?

N

(apenas 1 thread criou N tasks)



OPENMP AFFINITY

OPENMP AFFINITY

OpenMP Affinity consiste em

- Uma política proc_bind (política de afinidade de thread)
- Uma especificação de locais ("unidades de localização" ou processadores que podem ser núcleos, threads de hardware, soquetes, etc.).

O OpenMP Affinity permite que os usuários vinculem threads em locais específicos.

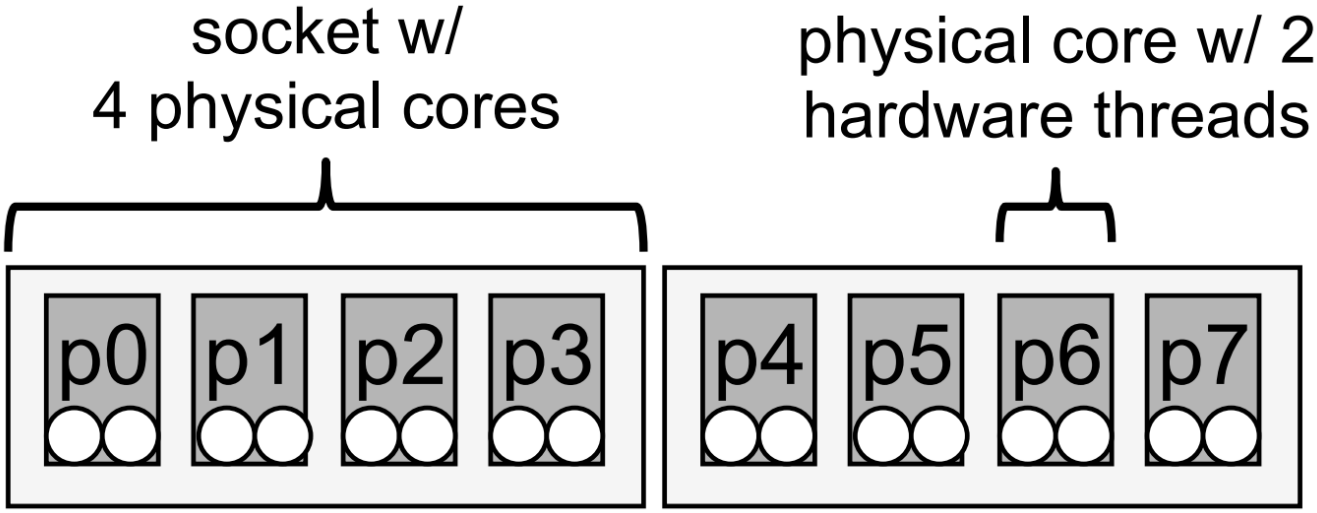
A colocação será mantida durante a região paralela.

No entanto, o runtime é livre para migrar os threads do OpenMP para diferentes núcleos (threads de hardware, soquetes, etc.) prescritos em um determinado local, se dois ou mais núcleos (threads de hardware, soquetes, etc.) foram atribuídos a um determinado local.

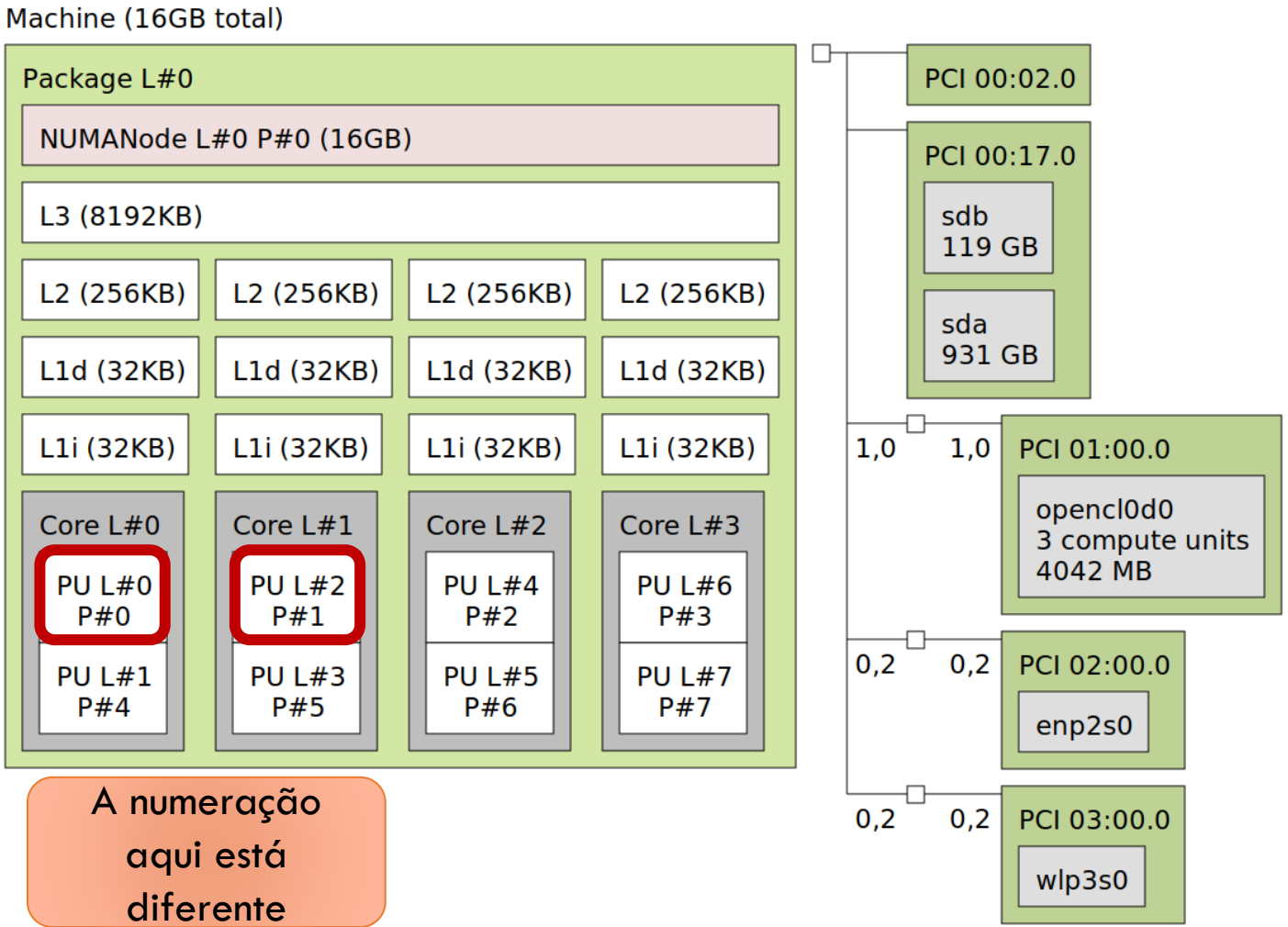
MÁQUINA TESTE

Ele consiste em dois soquetes, cada um equipado com um processador quad-core e configurado para executar dois threads de hardware simultaneamente em cada núcleo.

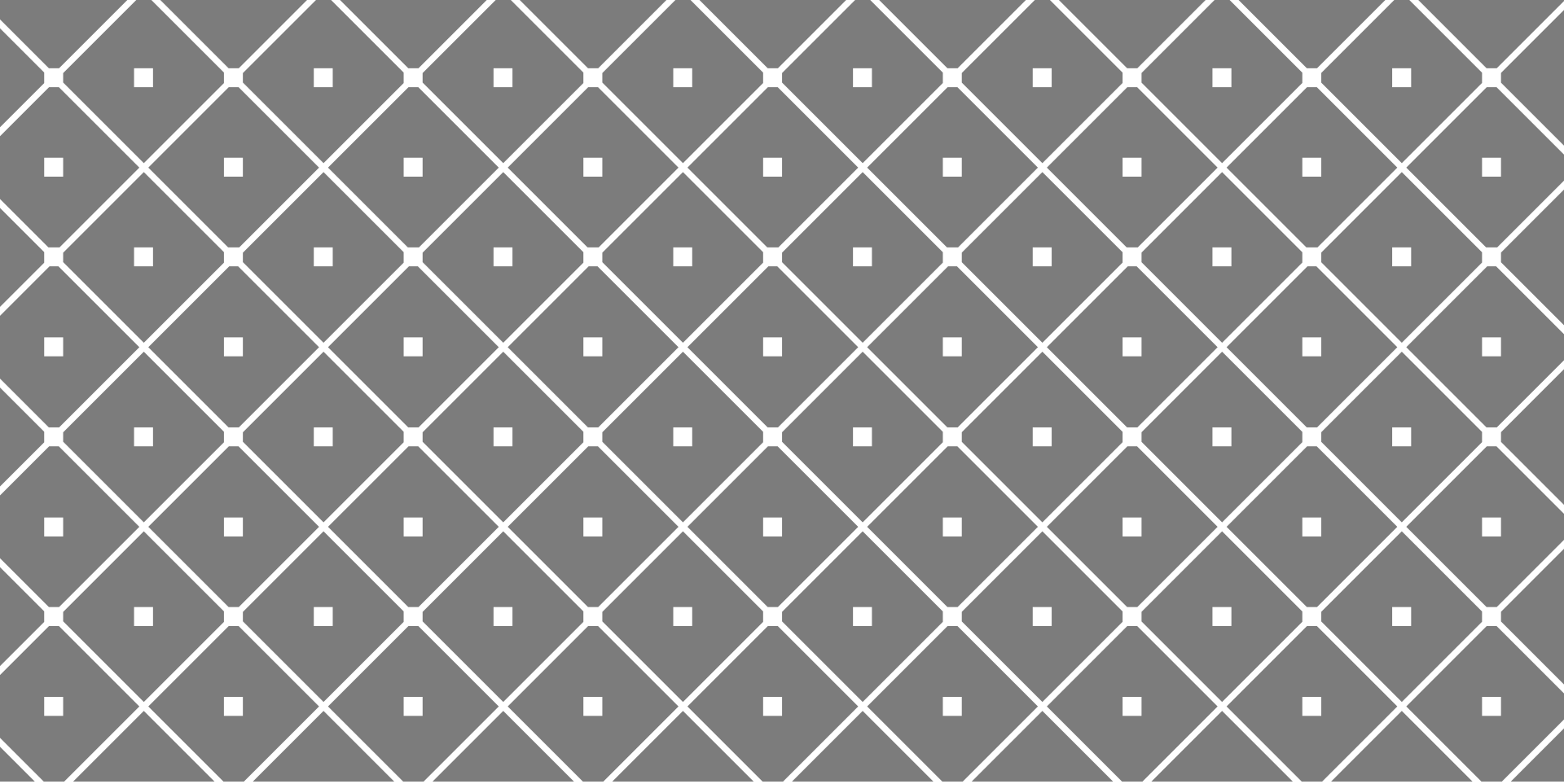
Esses exemplos pressupõem uma numeração de núcleo contígua começando em 0, de modo que os threads de hardware 0,1 formam o primeiro núcleo físico.



OBTENDO DETALHES DA MÁQUINA COM LSTOPO



A numeração aqui está diferente



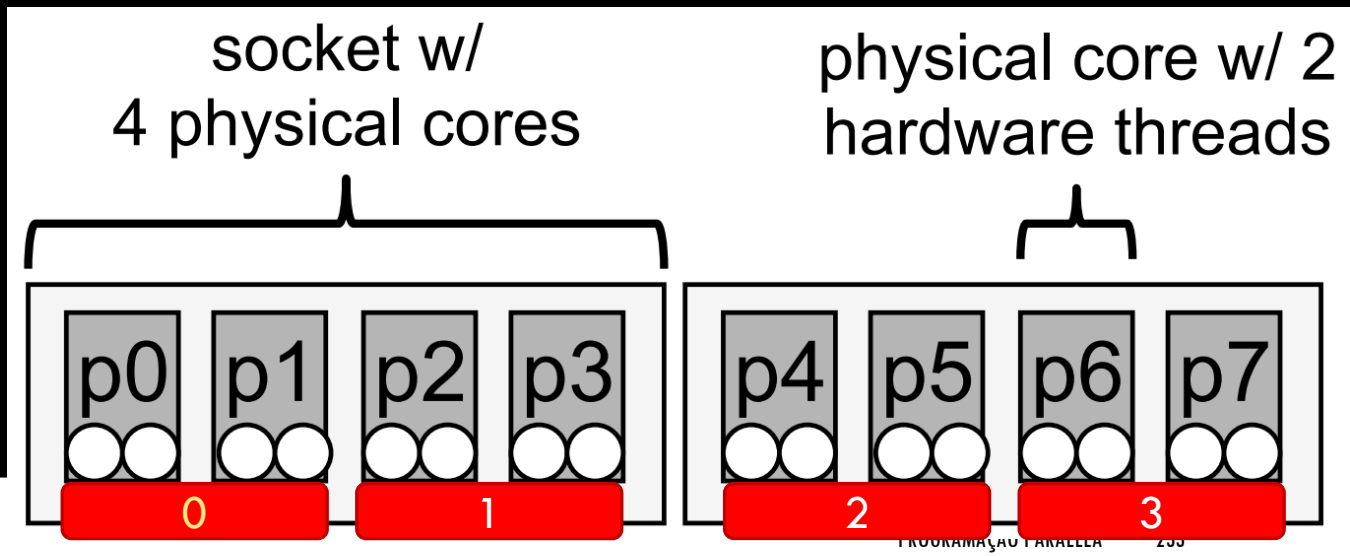
AFINIDADE ESPARSA

SPREAD AFFINITY POLICY

```
void work();
int main()
{
  #pragma omp parallel proc_bind(spread) num_threads(4)
  {
    work();
  }
  return 0;
}
```

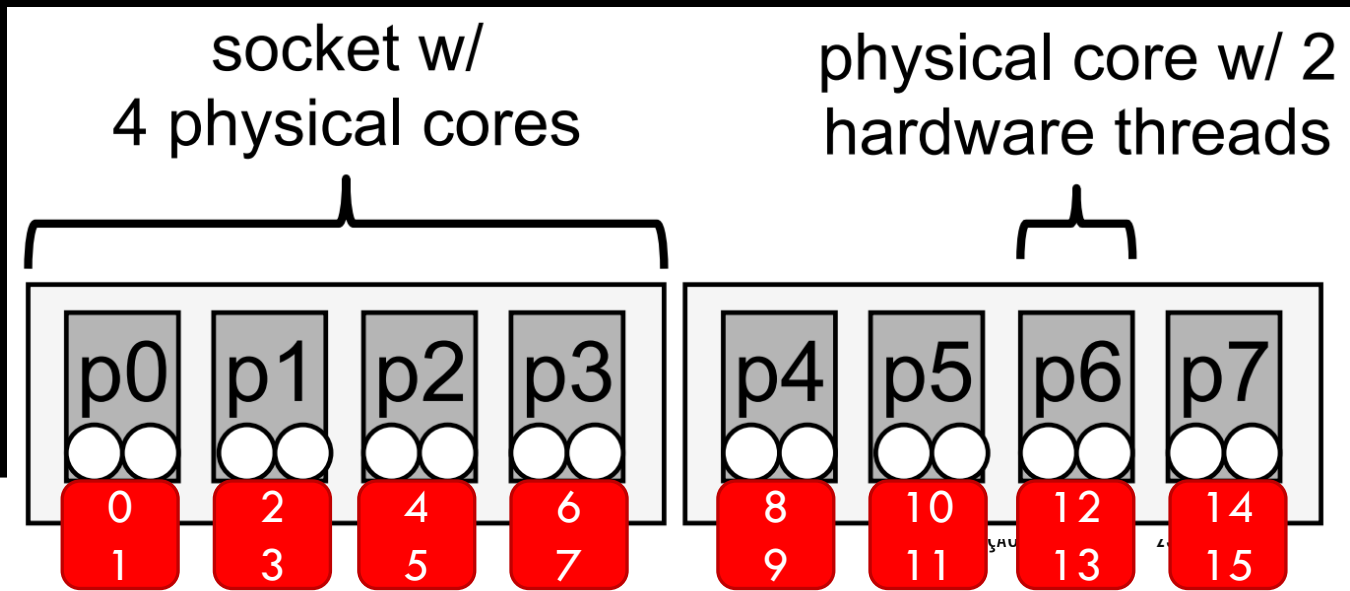
SPREAD AFFINITY POLICY

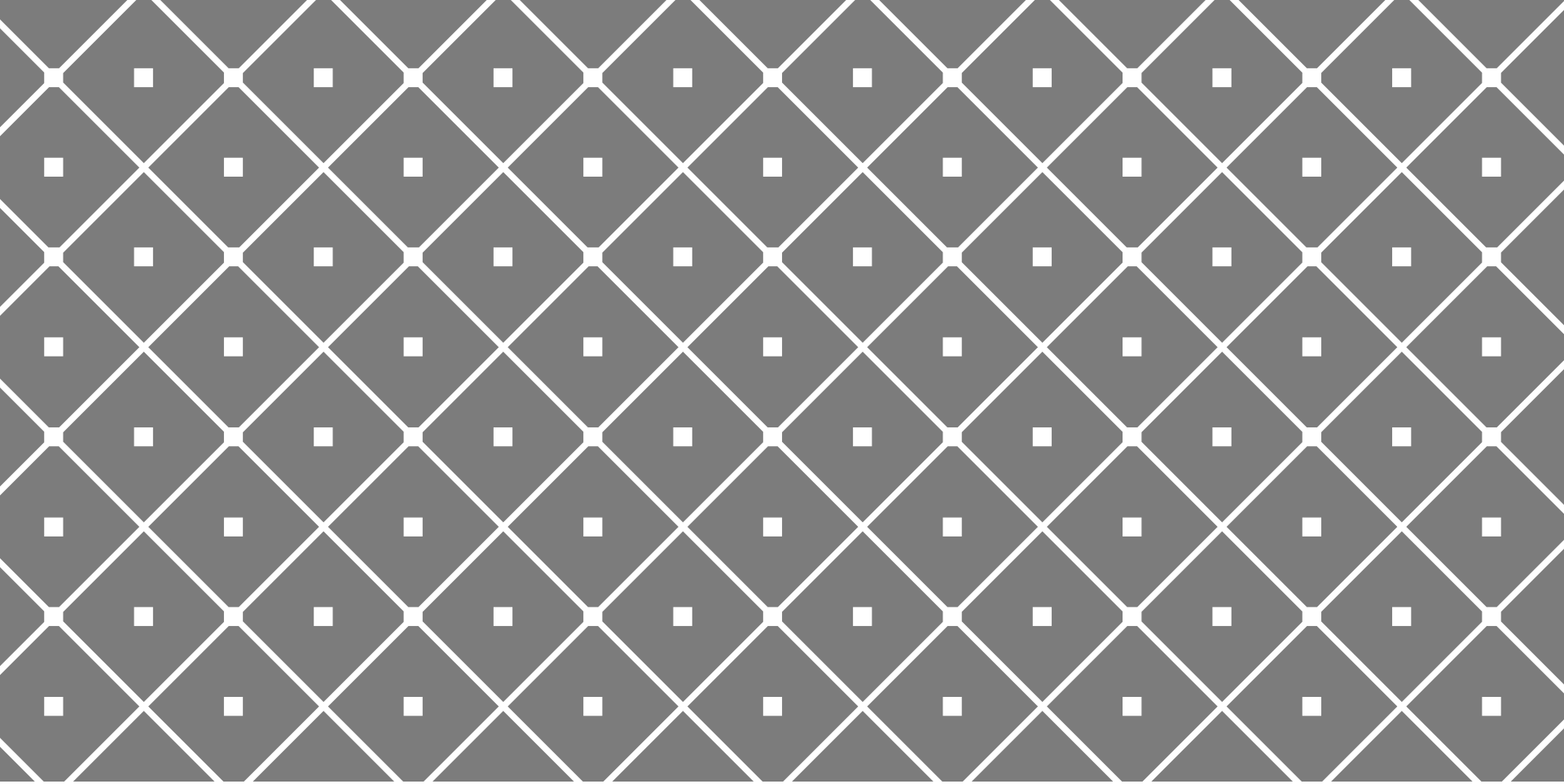
```
void work();
int main()
{
  #pragma omp parallel proc_bind(spread) num_threads(4)
  {
    work();
  }
  return 0;
}
```



SPREAD AFFINITY POLICY

```
void work();
int main()
{
  #pragma omp parallel proc_bind(spread) num_threads(16)
  {
    work();
  }
  return 0;
}
```





AFINIDADE PRÓXIMA

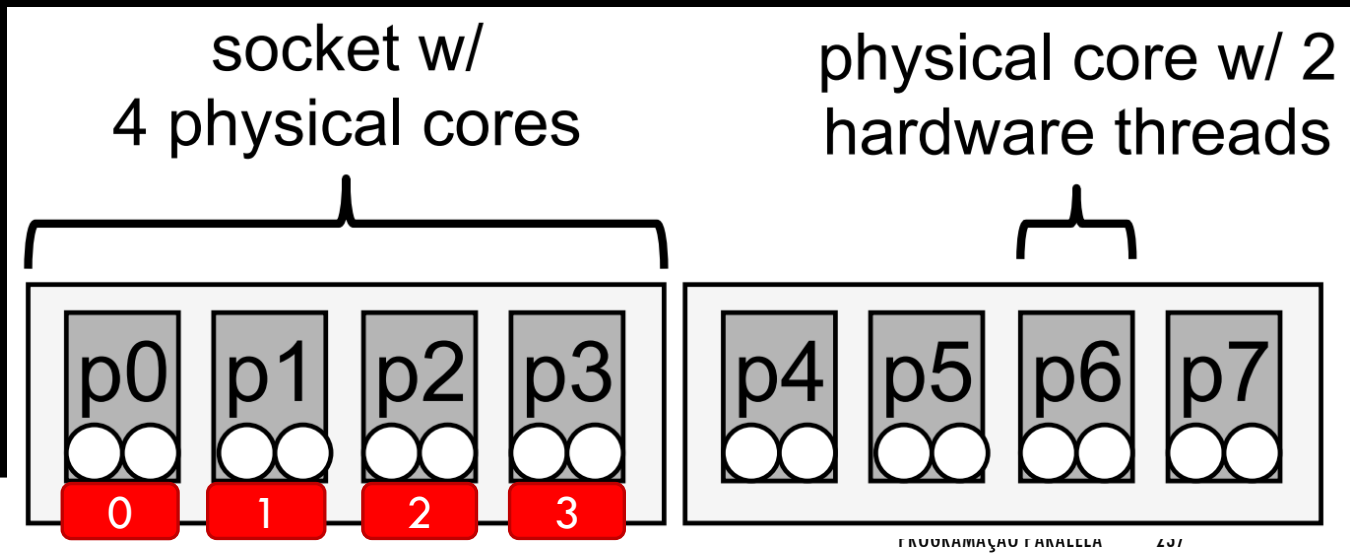
CLOSE AFFINITY POLICY

```
void work();
int main()
{
  #pragma omp parallel proc_bind(close) num_threads(4)
  {
    work();
  }
  return 0;
}
```

CLOSE AFFINITY POLICY

Considerando que a máster estava no core 0 (processador 0)

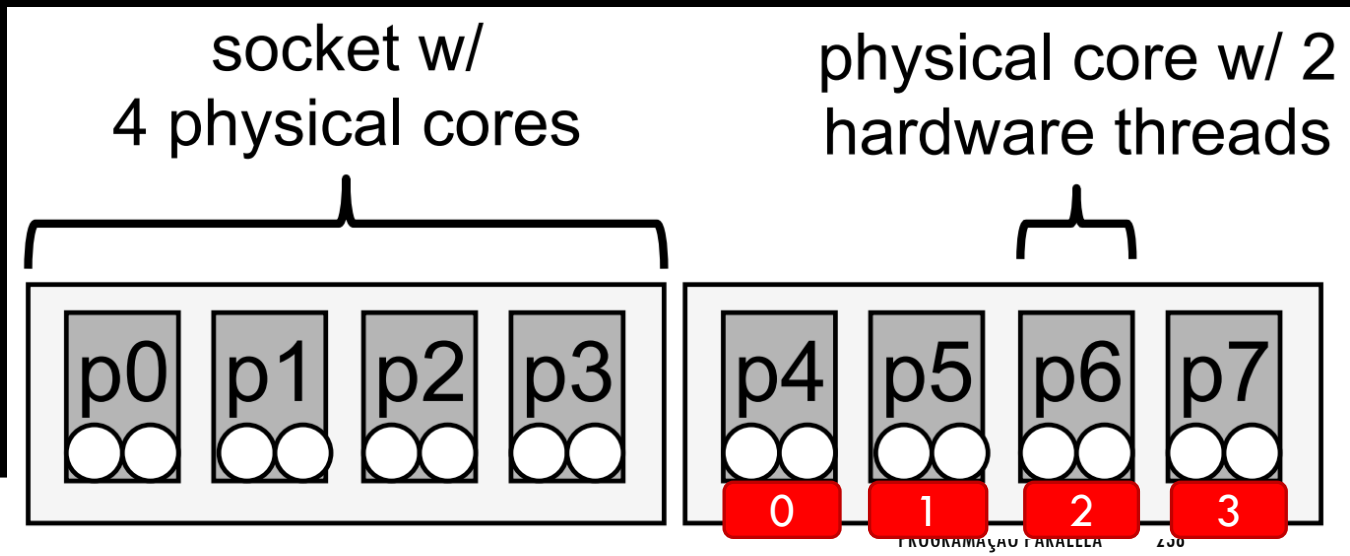
```
void work();
int main()
{
  #pragma omp parallel proc_bind(close) num_threads(4)
  {
    work();
  }
  return 0;
}
```



CLOSE AFFINITY POLICY

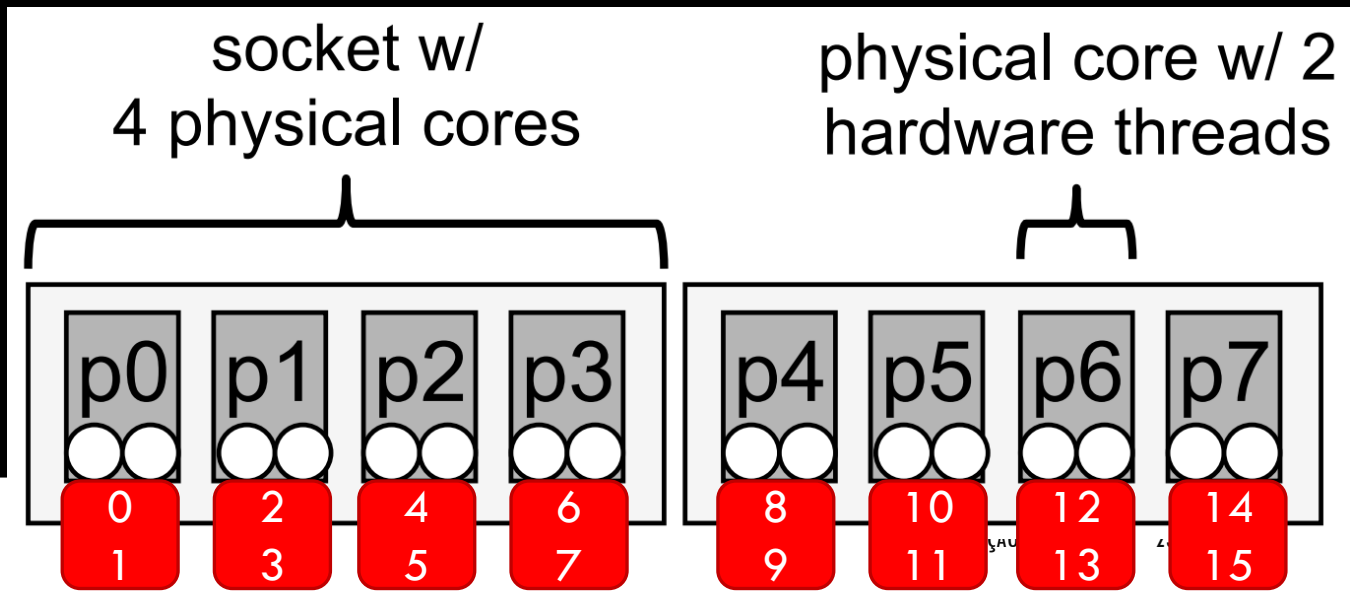
Considerando que a máster estava no core 0 (processador 1)

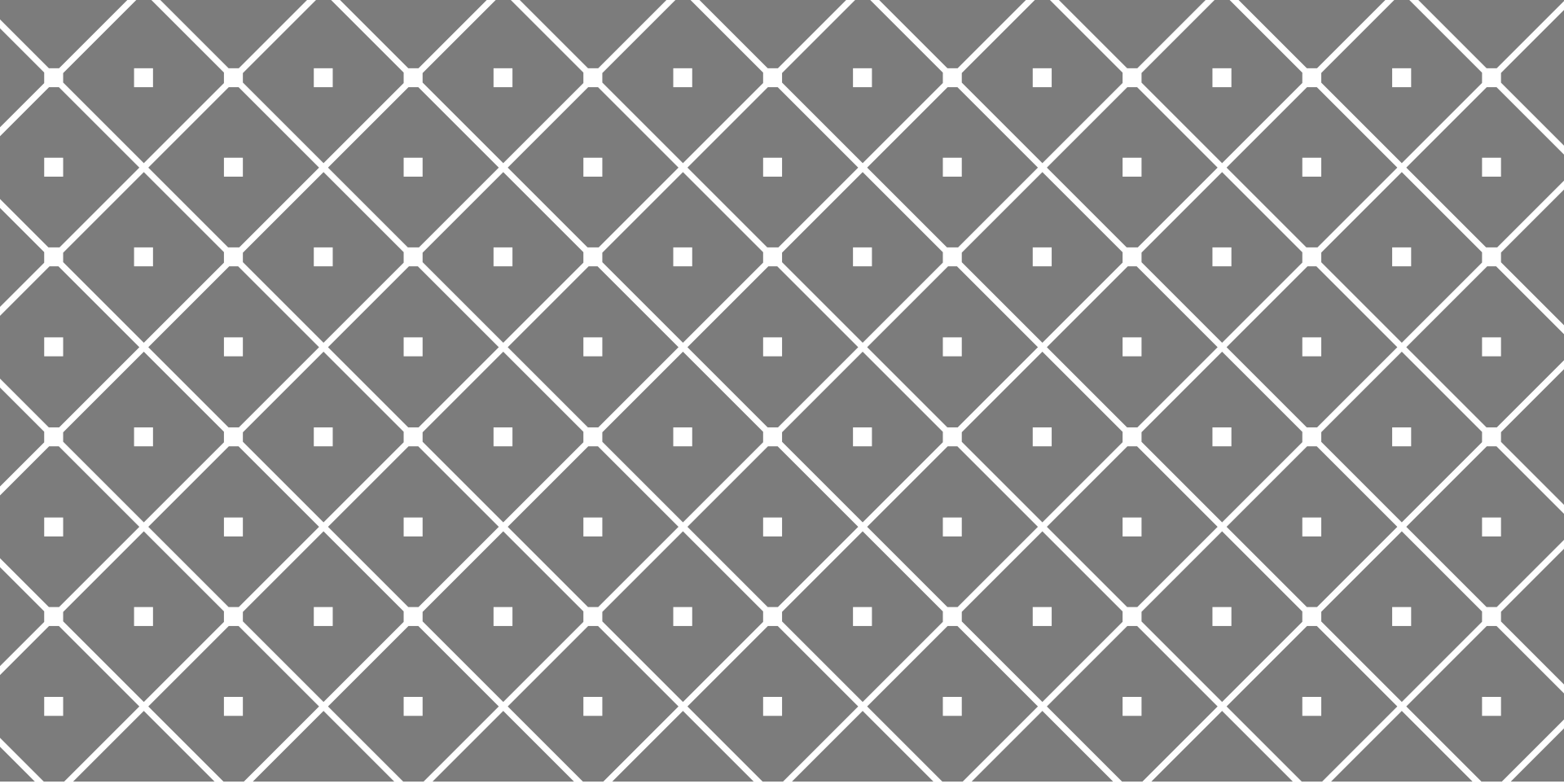
```
void work();
int main()
{
  #pragma omp parallel proc_bind(close) num_threads(4)
  {
    work();
  }
  return 0;
}
```



CLOSE AFFINITY POLICY

```
void work();
int main()
{
  #pragma omp parallel proc_bind(close) num_threads(16)
  {
    work();
  }
  return 0;
}
```





AFINIDADE MASTER

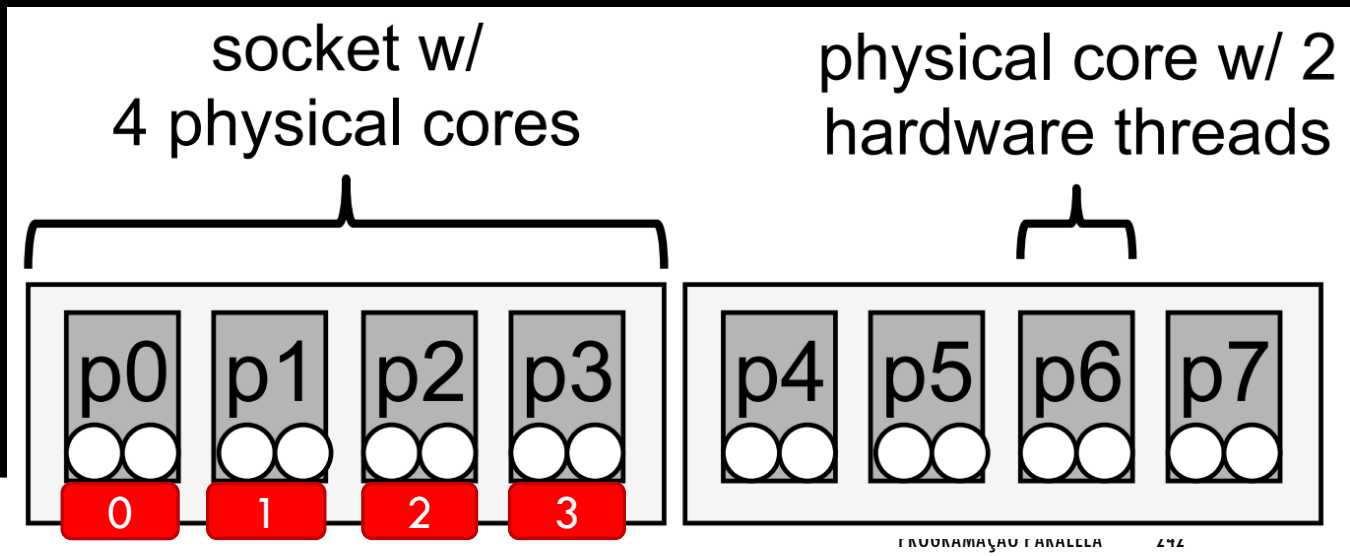
CLOSE AFFINITY POLICY

```
void work();
int main()
{
  #pragma omp parallel proc_bind(master) num_threads(4)
  {
    work();
  }
  return 0;
}
```

CLOSE AFFINITY POLICY

Considerando que a master estava no core 0 (processador 0)

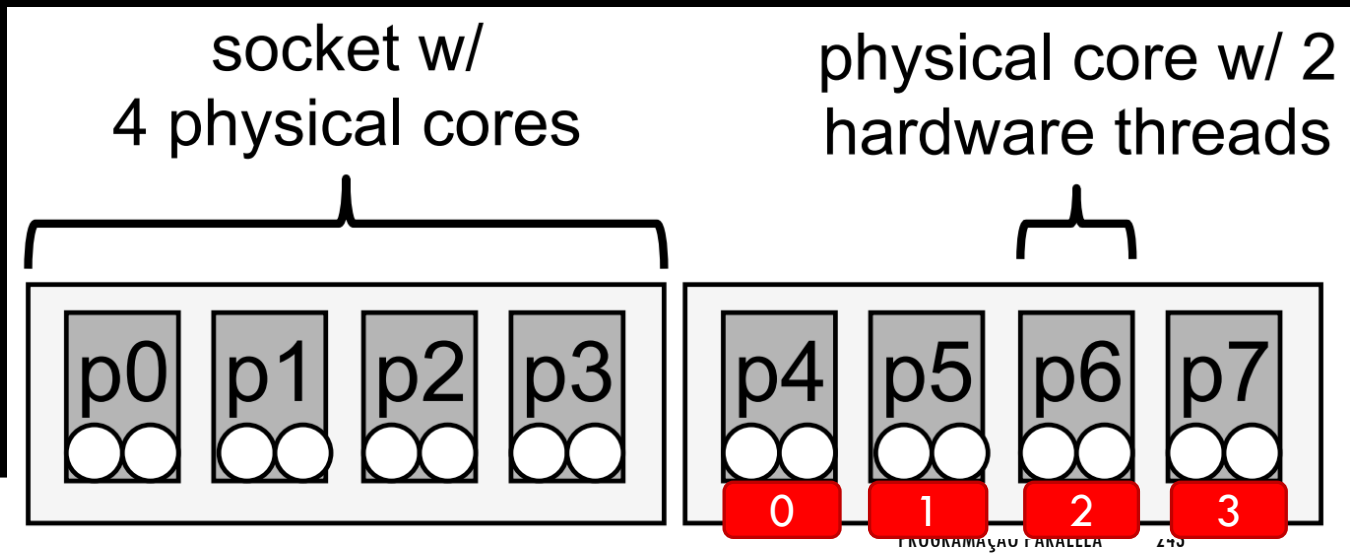
```
void work();
int main()
{
  #pragma omp parallel proc_bind(master) num_threads(4)
  {
    work();
  }
  return 0;
}
```



CLOSE AFFINITY POLICY

Considerando que a master estava no core 0 (processador 1)

```
void work();
int main()
{
  #pragma omp parallel proc_bind(master) num_threads(4)
  {
    work();
  }
  return 0;
}
```



POLITICAS DE AFINIDADE

Threads de uma equipe são posicionados em locais de maneira defininda na variável de ambiente OMP_PROC_BIND ou a cláusula proc_bind.

Quando OMP_PROC_BIND é definido como

- FALSE, nenhuma afinidade é imposta;
- TRUE, a afinidade é a implementação definida para um conjunto de locais na variável OMP_PLACES ou para locais definidos pela implementação se a variável OMP_PLACES não estiver definida.

A variável OMP_PLACES também pode ser definida como um nome abstrato (threads, núcleos, soquetes) para especificar que um local é um único thread de hardware, um núcleo ou um soquete, respectivamente.

Esta descrição do OMP_PLACES é mais útil quando o número de threads é igual ao número de threads de hardware, núcleos ou soquetes. Também pode ser usado com uma política de distribuição de fechamento ou propagação quando a igualdade não se mantém.