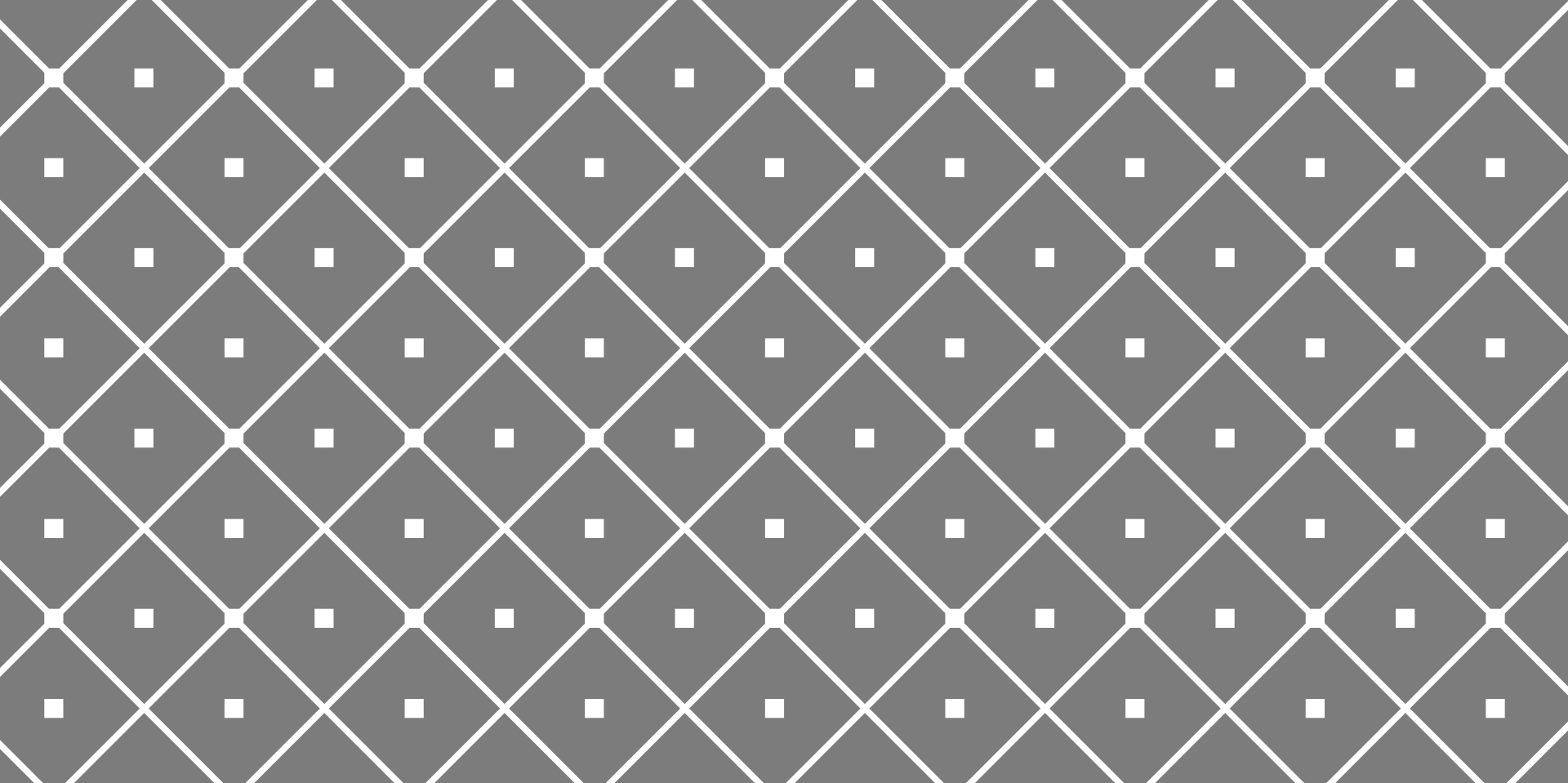




PROGRAMAÇÃO PARALELA OPENMP — AULA 03

Marco A. Zanata Alves



PROGRAMA PI COMPLETO

SERIAL PI PROGRAM

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

EXEMPLO: PI COM UM LAÇO E REDUÇÃO

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

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

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

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

RESULTADOS*

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

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

Threads	1. SPMD	SPMD padding	SPMD critical	Pi Loop	S(p)
1	1.86	1.86	1.87	1.91	0,97
2	1.03	1.01	1.00	1.02	1,82
3	1.08	0.69	0.68	0.80	2,32
4	0.97	0.53	0.53	0.68	2,73

LAÇOS (CONTINUAÇÃO)

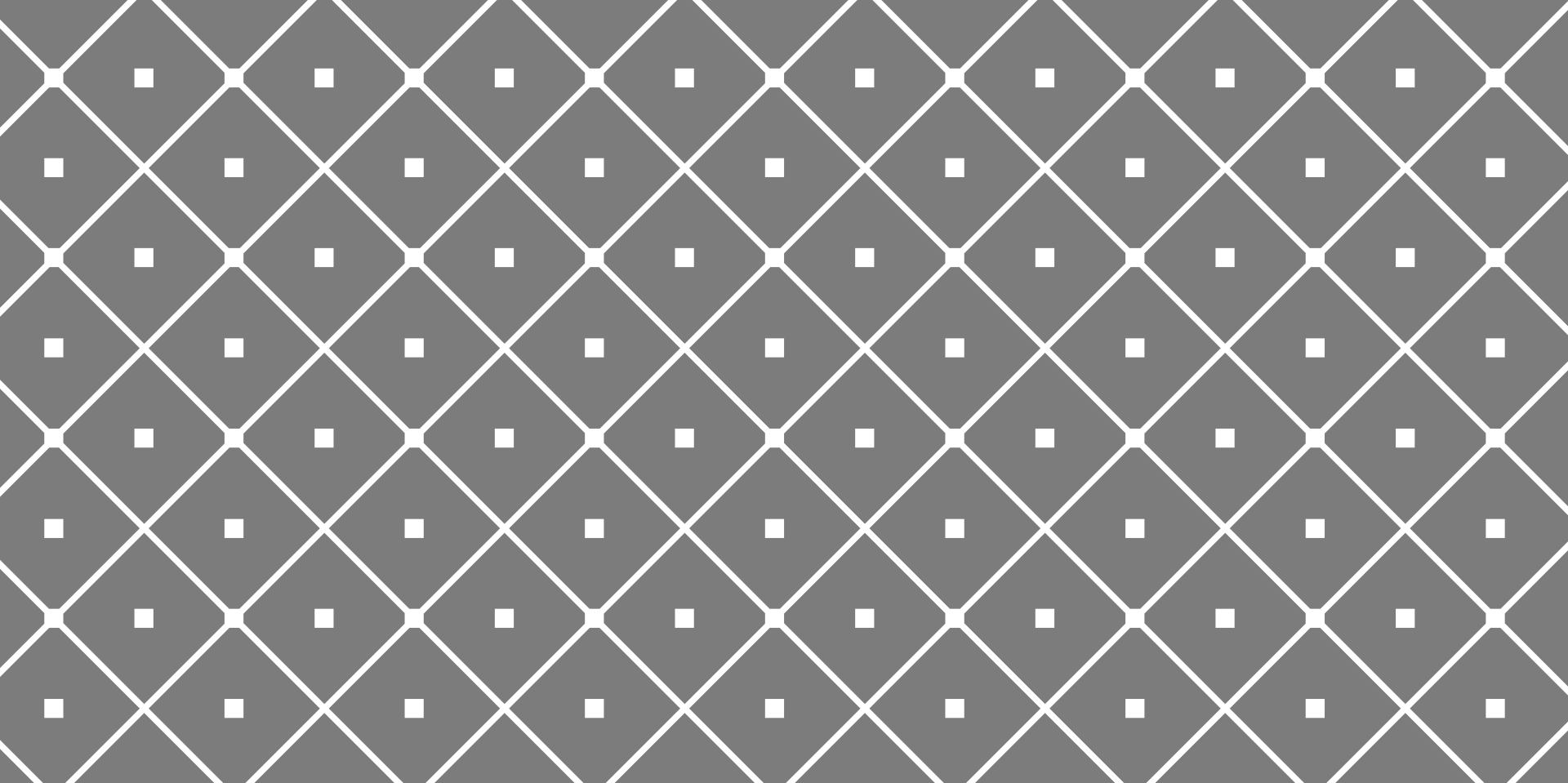
Novidades do OpenMP 3.0

Tornou o `schedule(runtime)` mais útil

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

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

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



SINCRONIZAÇÃO: SINGLE, MASTER, ETC.

SINCRONIZAÇÃO: BARRIER E NOWAIT

Barrier: Cada thread aguarda até que todas as demais cheguem

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

Barreira explícita

Barreira implícita no final da construção FOR

Remove barreira implícita devido ao nowait (use com cuidado)

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

CONSTRUÇÃO MASTER

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

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

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

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

Sem barreira implícita

Barreira explícita

CONSTRUÇÃO SINGLE

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

Uma **barreira implícita** estará no final do bloco single (podemos

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

Barreira implícita

Nesse caso, podemos usar **"nowait"**

CONSTRUÇÃO SECTIONS PARA DIVISÃO DE TRABALHO

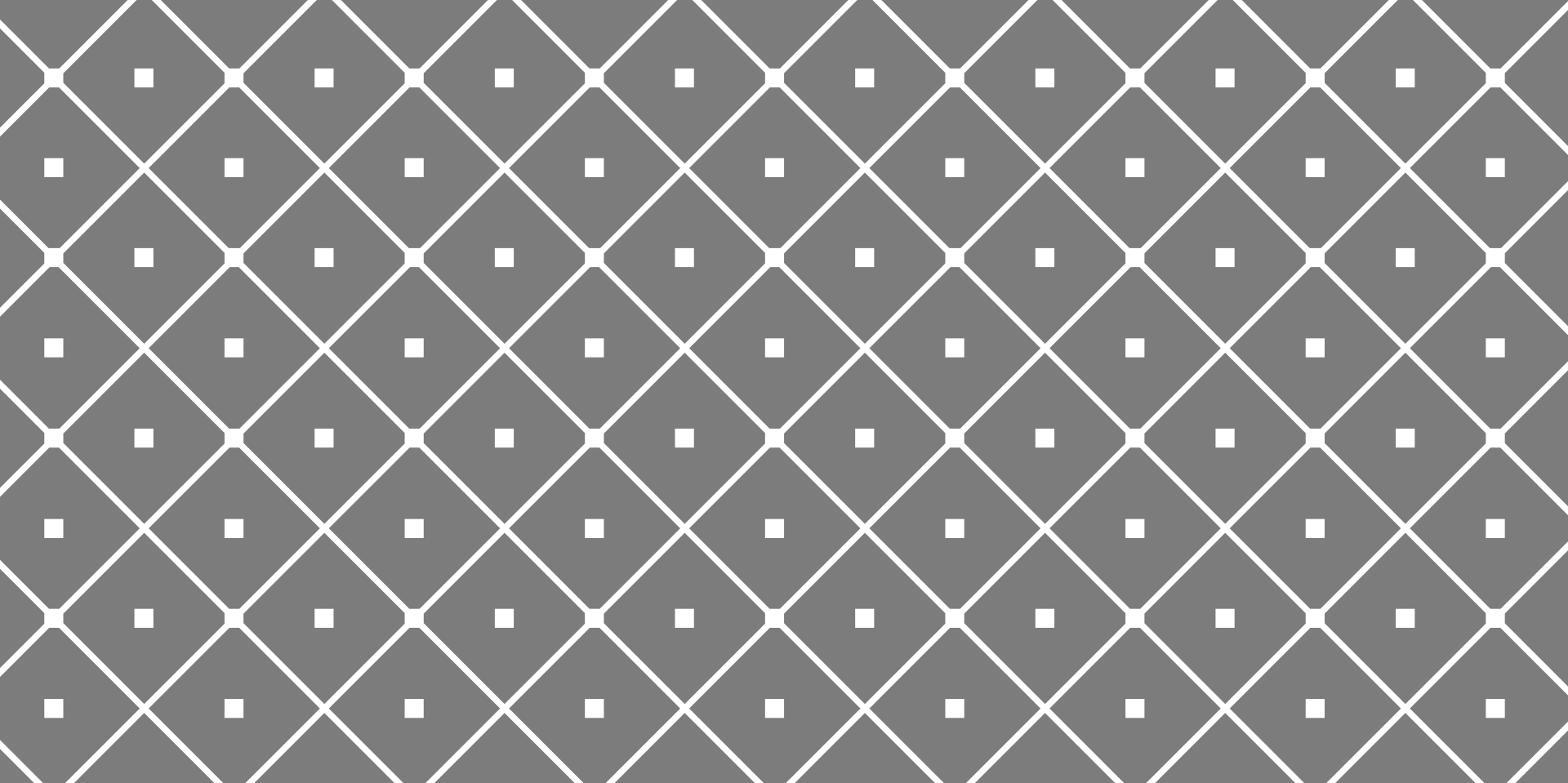
A construção de divisão de trabalho com **sections** prove um bloco estruturado diferente para cada thread.

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

Sections provê uma forma simples de atingir o paralelismo por rotinas (invés de domínio)

Por padrão, **existe uma barreira implícita** no final do "omp sections".

Use a diretiva "nowait" para desligar essa barreira.



SINCRONIZAÇÃO DE BAIXO NÍVEL

SINCRONIZAÇÃO: ROTINAS **LOCK**

Um lock está disponível caso esteja "não setado" (unset).

- `omp_init_lock()` – Inicializa o lock
- `omp_set_lock()` – Obtém o lock (bloqueia a execução durante a espera)
- `omp_unset_lock()` – Solta o lock (sai da região crítica)
- `omp_test_lock()` – Tenta obter o lock (não bloqueia a execução na espera)
- `omp_destroy_lock()` – Destrói o lock

SINCRONIZAÇÃO: ROTINAS LOCK

Um lock está disponível caso esteja "não setado" (unset).

- `omp_init_lock()` – Inicializa o lock
- `omp_set_lock()` – Obtém o lock (bloqueia a execução durante a espera)
- `omp_unset_lock()` – Solta o lock (sai da região crítica)
- `omp_test_lock()` – Tenta obter o lock (não bloqueia a execução na espera)
- `omp_destroy_lock()` – Destrói o lock

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

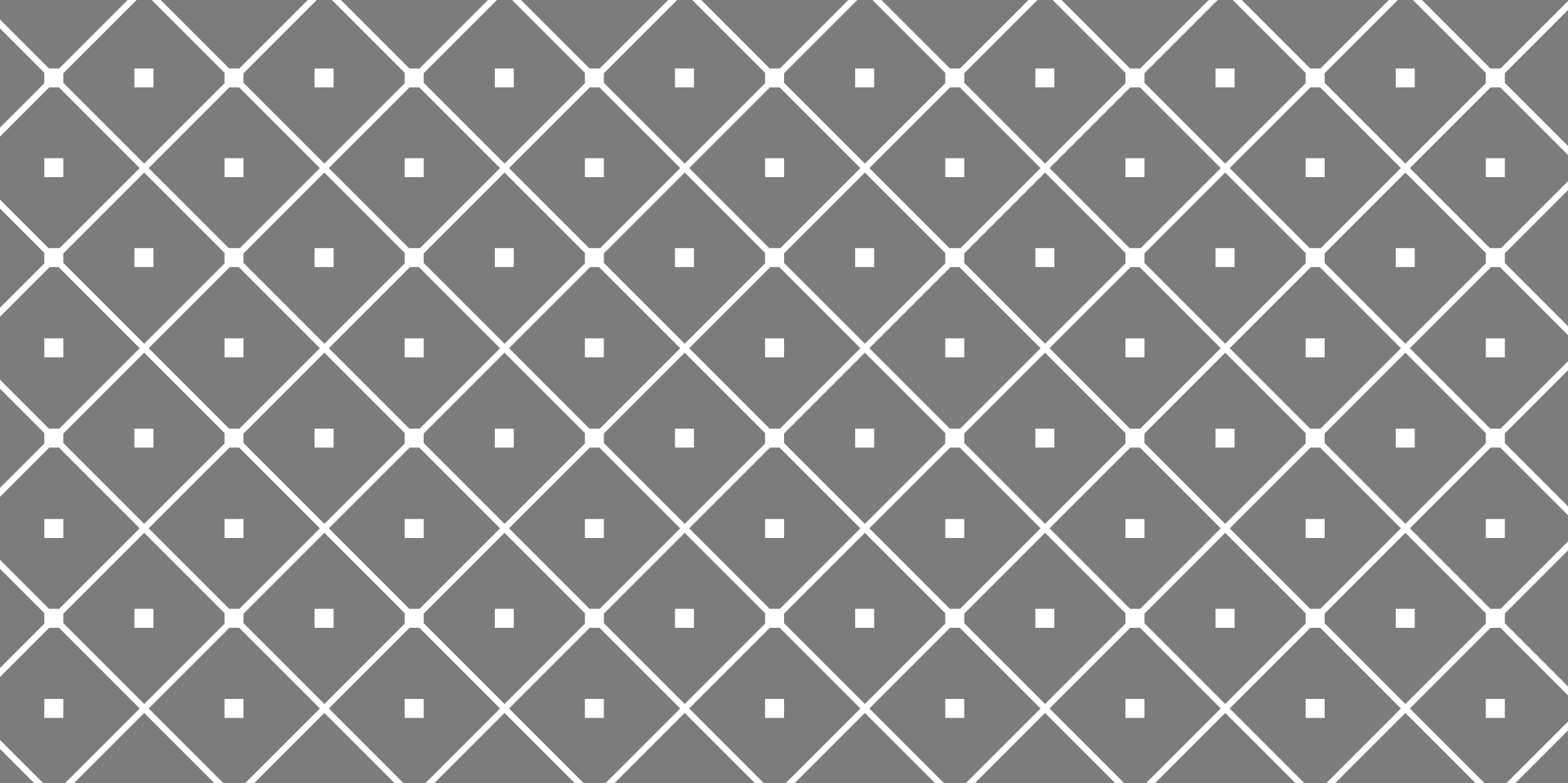
Note: uma thread sempre irá acessar a cópia mais recente do lock, logo, não precisamos fazer o flush na variável do lock.

SINCRONIZAÇÃO: LOCK ANINHADO

Um lock aninhado está disponível se estiver

- "unset" (nenhuma thread o possui) ou se está
- "set" e o dono for a thread que estiver executando a função com lock aninhado

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



EXEMPLO COM LOCK

EXEMPLO DO HISTOGRAMA

Vamos considerar o pedaço de texto abaixo:

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

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

EXEMPLO DO HISTOGRAMA

Vamos considerar o pedaço de texto abaixo:

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

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

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

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

EXEMPLO DO HISTOGRAMA

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

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

Thread A → Thread B

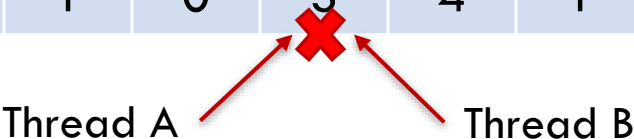
3 possíveis soluções

São raros os casos de conflito, mas para estarmos seguros, vamos garantir exclusão mútua para atualizar os elementos do histograma

EXEMPLO DO HISTOGRAMA

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

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



Thread A Thread B

3 possíveis soluções

- Região crítica em volta desse vetor
- Cópias locais do vetor para cada thread
- Lock por cada posição do vetor

SINCRONIZAÇÃO: LOCKS SIMPLES

```
#pragma omp parallel for
for(i=0;i<NBUCKETS; i++){
    omp_init_lock(&hist_locks[i]);
    hist[i] = 0;
}
```

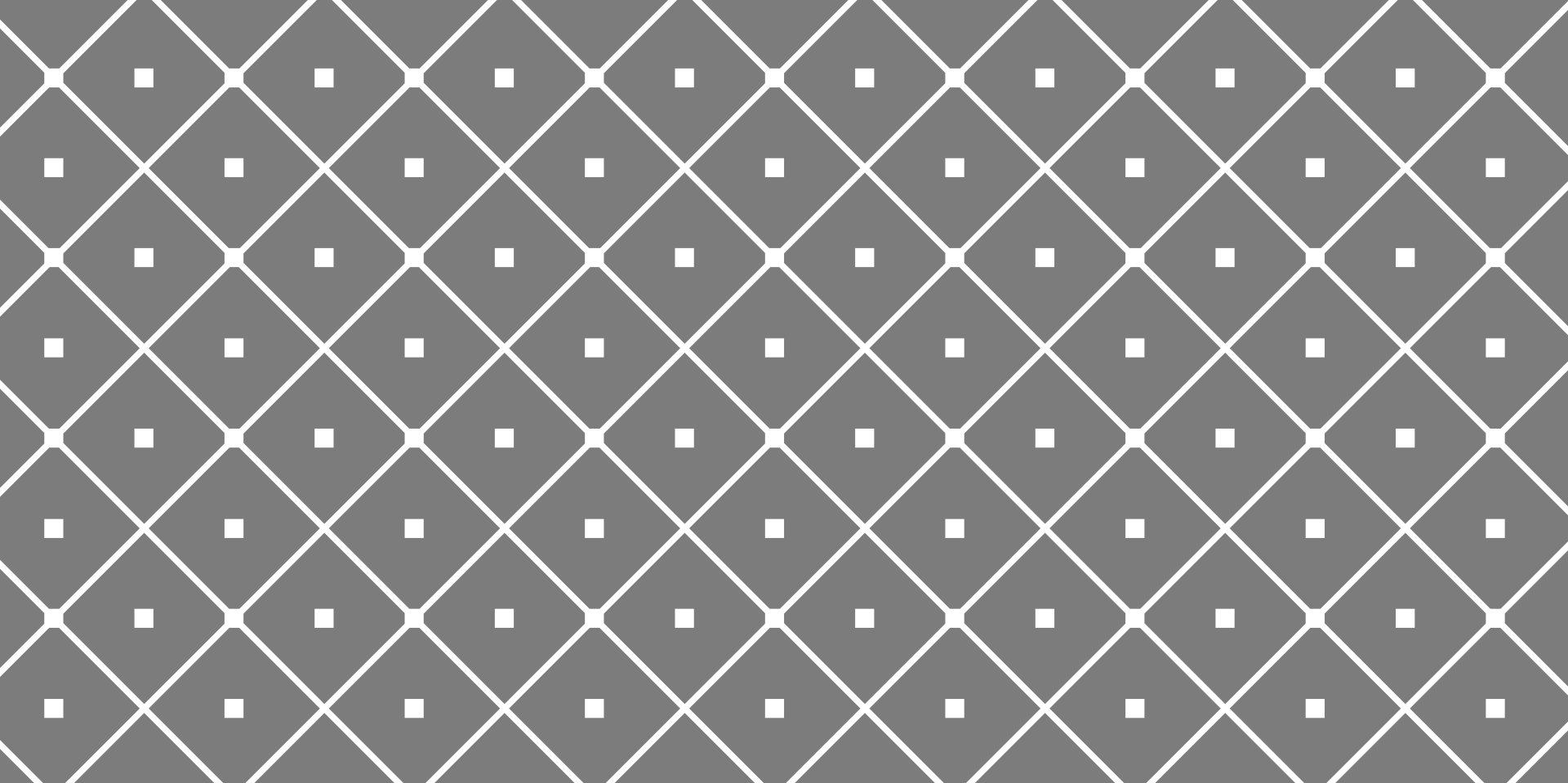
Um lock por elemento do histograma

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

Garante exclusão mútua ao atualizar o vetor do histograma

```
for(i=0;i<NBUCKETS; i++)
    omp_destroy_lock(&hist_locks[i]);
```

Libera a memória quando termina




OUTRAS ROTINAS DA BIBLIOTECA OMP.H

ROTINAS DA BIBLIOTECA DE EXECUÇÃO

Modifica/verifica o número de threads

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

Número máximo de threads que podemos requisitar ao SO



Estamos em uma região paralela ativa?

- `omp_in_parallel()`

ROTINAS DA BIBLIOTECA DE EXECUÇÃO

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

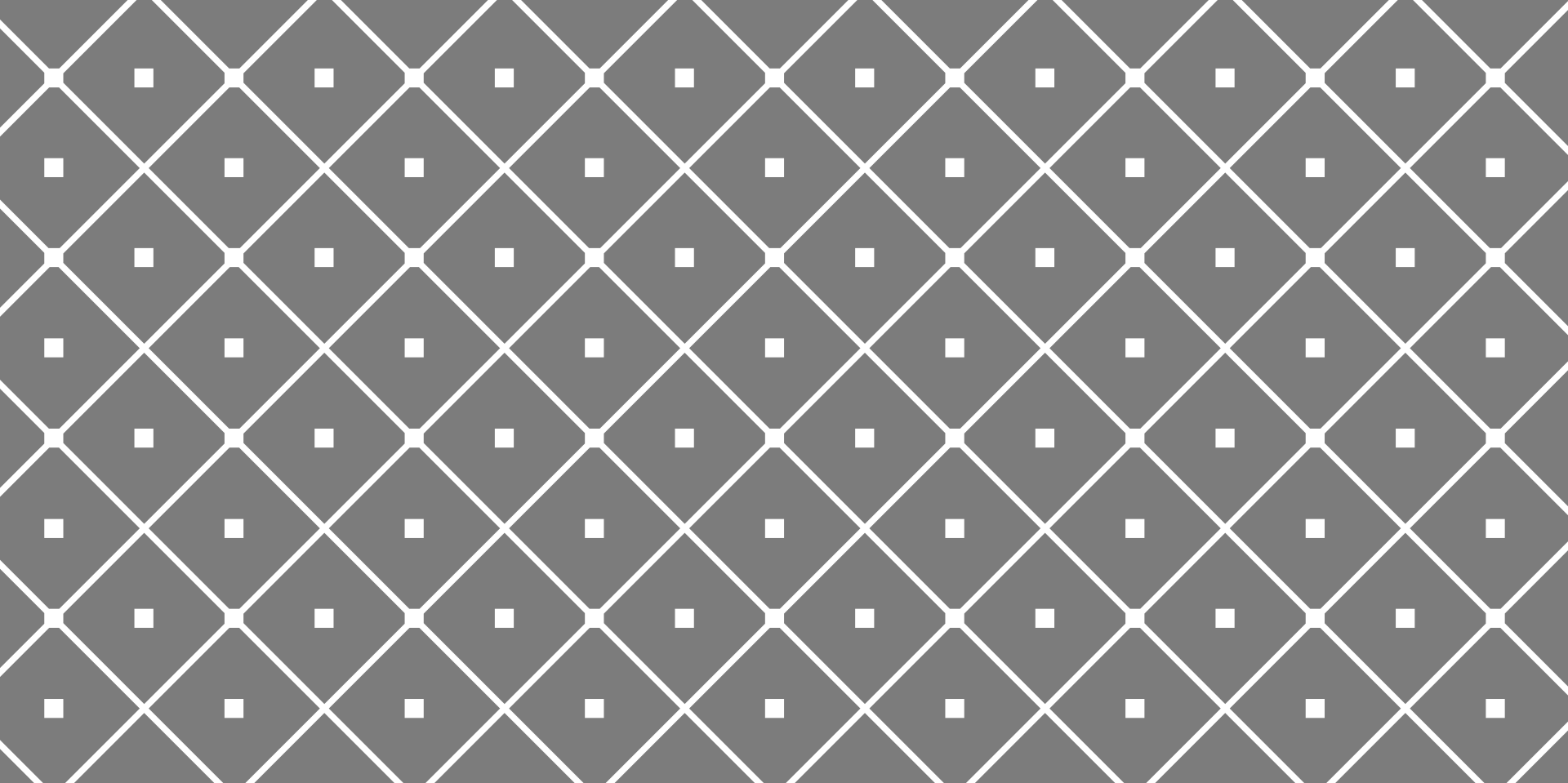
- `omp_set_dynamic()`
- `omp_get_dynamic()`

← Entra em modo dinâmico

Quantos processadores no sistema?

- `omp_num_procs()`

...mais algumas rotinas menos frequentemente utilizadas.



EXEMPLOS E ARMADILHAS

CONFIGURANDO O NÚMERO DE THREADS

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

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

Desligue o ajuste dinâmico de threads

Peça o número de threads igual ao número de processadores

Proteja essa operação para evitar condições de corrida

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

ARMADILHA

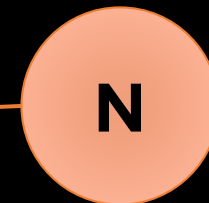
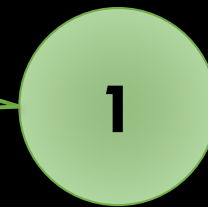
```
#include <omp.h>
void main() {
    int num_threads;
    omp_set_dynamic( 0 );
    omp_set_num_threads( omp_num_procs() );
    num_threads = omp_get_num_threads();
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        #pragma omp single
        num_threads = omp_get_num_threads();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

Quantas threads existem aqui?



ARMADILHA

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



VARIÁVEIS DE AMBIENTE

Define o número padrão de threads.

- **OMP_NUM_THREADS** int_literal

Controle do tamanho da pilha das threads filhas

- **OMP_STACKSIZE**

Ex.:

```
export OMP_NUM_THREADS=10
```

Essas variáveis são interessantes pois não precisamos recompilar o código

VARIÁVEIS DE AMBIENTE

Fornece dicas de como tratar as threads durante espera

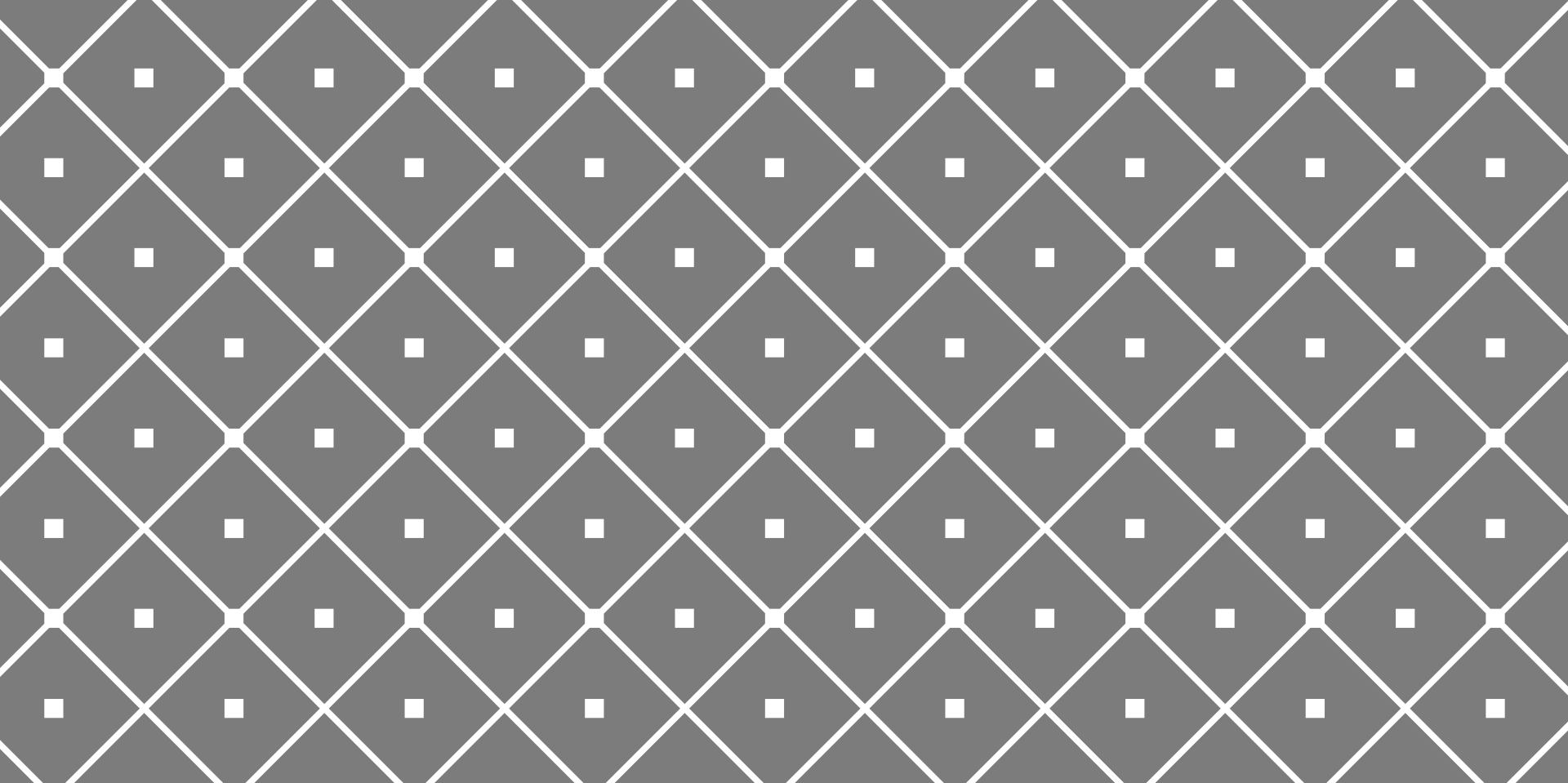
- **OMP_WAIT_POLICY**

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

PASSIVE tente liberar o processador nos barriers/locks (**sleep**)

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

- **OMP_PROC_BIND** true | false



ESCOPO DAS VARIÁVEIS

ESCOPO PADRÃO DE DADOS

A maioria das variáveis são compartilhadas por padrão

Região paralela

Fora → Global
Dentro → Privado

Código geral

Heap (malloc/new) → Global
Stack (variáveis nas rotinas) → Privado

ESCOPO PADRÃO DE DADOS

A maioria das variáveis são compartilhadas por padrão

Região paralela

Fora → Global
Dentro → Privado

Código geral

Heap (malloc/new) → Global
Stack (variáveis nas rotinas) → Privado

Variáveis globais são **compartilhadas** entre as threads:

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

Mas existem áreas **privadas**:

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

EX: COMPARTILHAMENTO DE DADOS

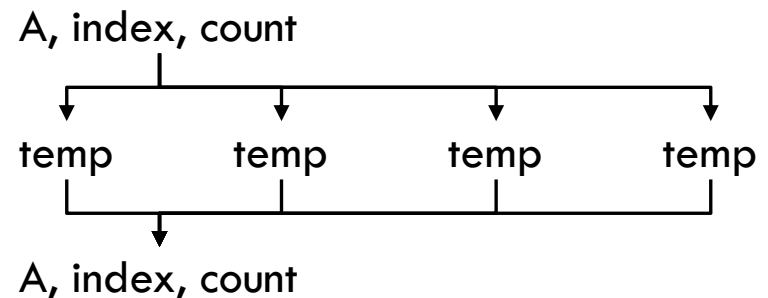
```
double A[10];  
int main() {  
    int index[10];  
    #pragma omp parallel  
    work(index);  
    printf("%d\n", index[0]);  
}
```

```
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(vars)
- PRIVATE(vars)

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

O valor global é copiado para cada variável privada:

- FIRSTPRIVATE(vars)

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

- LASTPRIVATE(vars)

Os modos padrão podem ser sobrescritos:

- DEFAULT (SHARED | **NONE**)
- DEFAULT(PRIVATE) **apenas para Fortran**

*Todas diretivas se aplicam a construções com **parallel** e de divisão de tarefa (**for**).

Porém, "**share**" se aplica apenas a construções **parallel**.

COMPARTILHAMENTO DE DADOS: PRIVATE

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

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

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

tmp não foi inicializada

tmp vale 0 aqui

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

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

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

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

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

tmp tem um valor não especificado

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

Não está especificado qual 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., quando estava $i=(n-1)$)

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?

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)
```

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

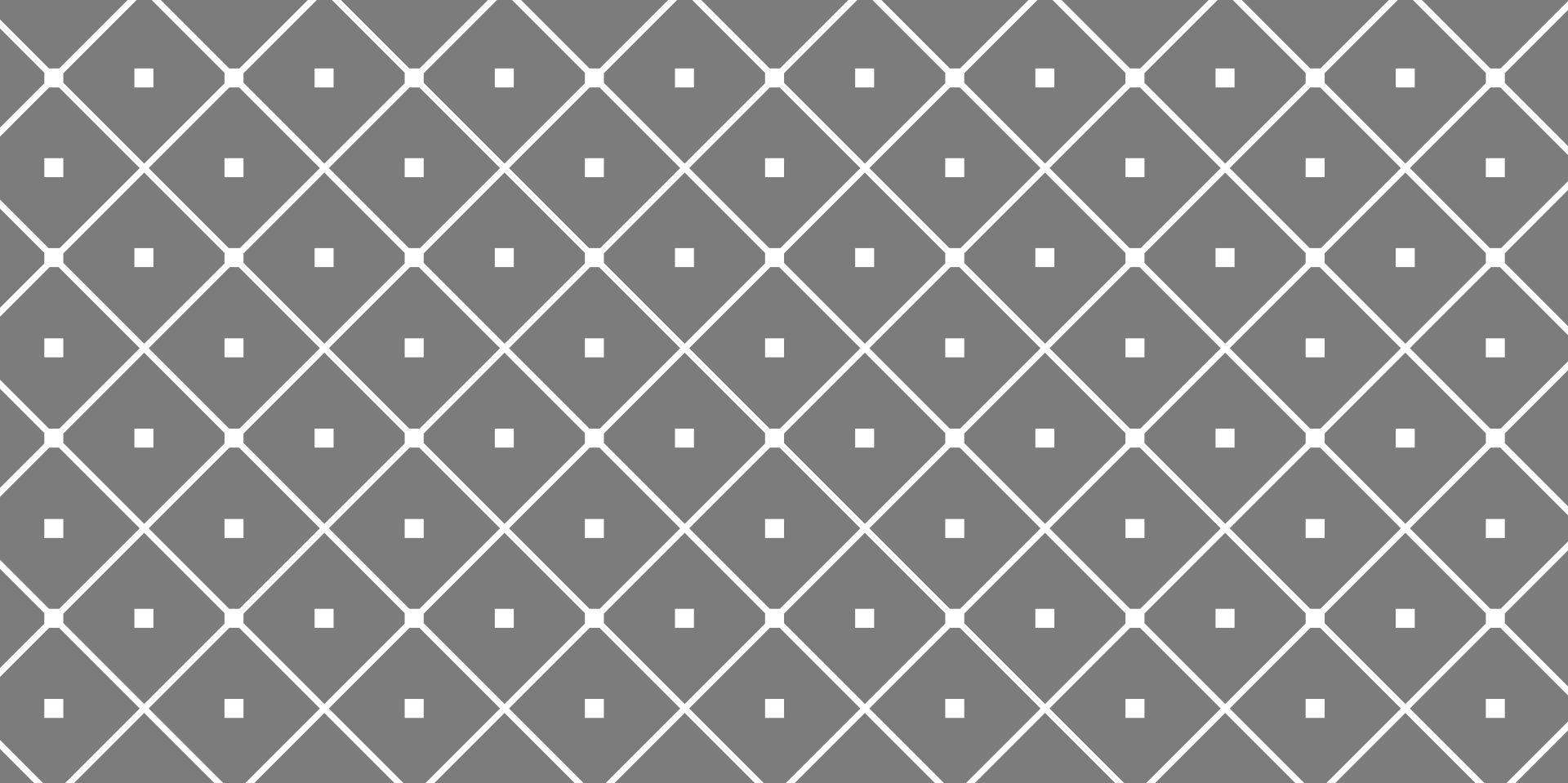
Para mudar o padrão: DEFAULT(PRIVATE)

- 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).



RETORNANDO AO PROGRAMA PI

PROGRAMA SERIAL PI

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

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

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

EXEMPLO: PROGRAMA PI ...

MUDANÇAS MÍNIMAS

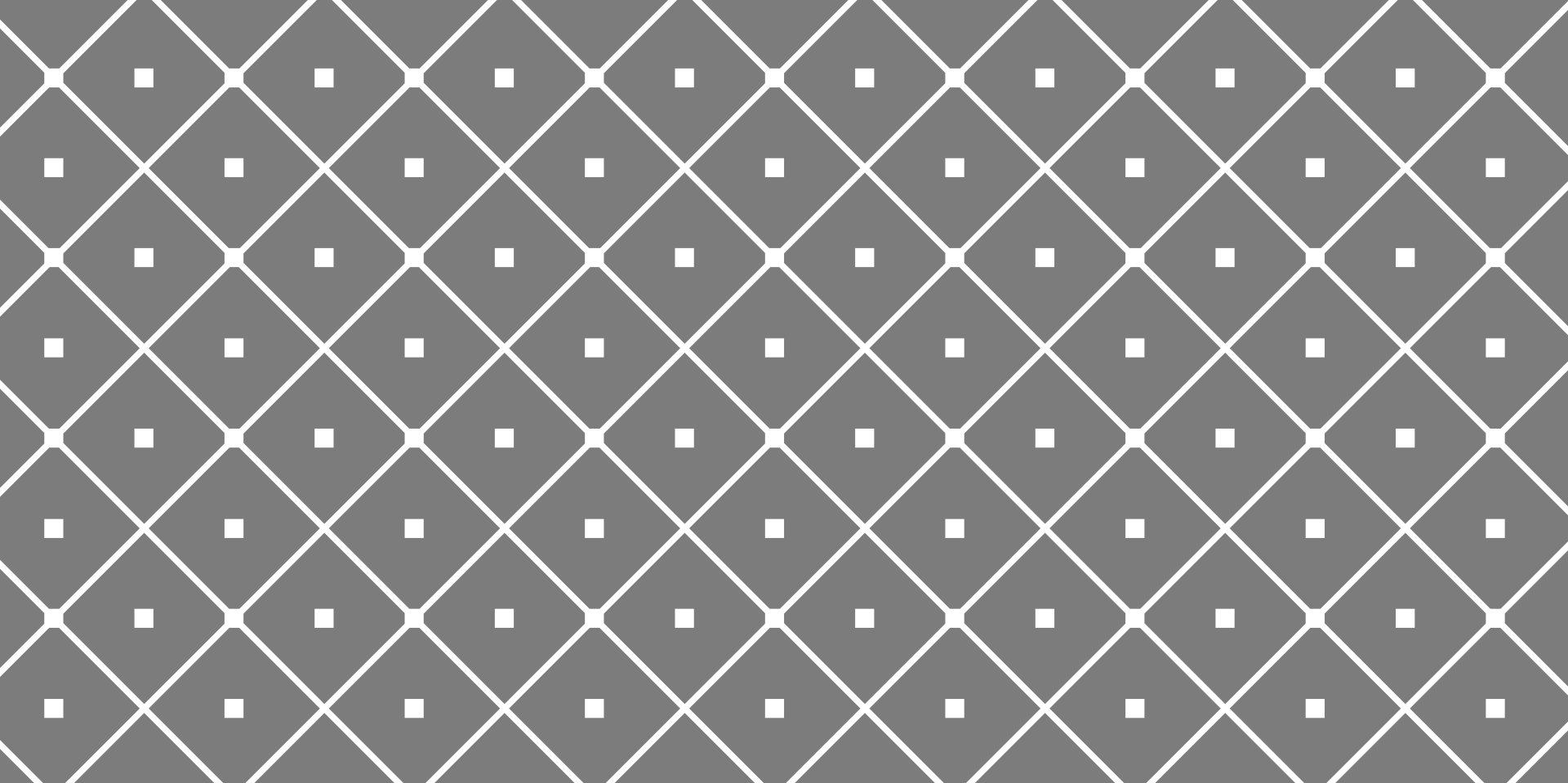
Para boas implementações OpenMP, reduction é mais escalável que o critical.

```
#include <omp.h>
static long num_steps = 100000; double step;

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

i é privada por padrão

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



DEBUGANDO PROGRAMAS OPENMP

EXERCÍCIO 6: ÁREA DO CONJUNTO MANDELBROT

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

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

Mas sua versão sequencial funciona corretamente

Procure e conserte os erros!

```

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

```

```

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

```

Quando executamos esse programa, obtemos uma resposta errada a cada execução ...

Existe uma condição de corrida!!!

EXERCÍCIO 6 (CONT.)

Ao terminar de consertar, **tente otimizar o programa**

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