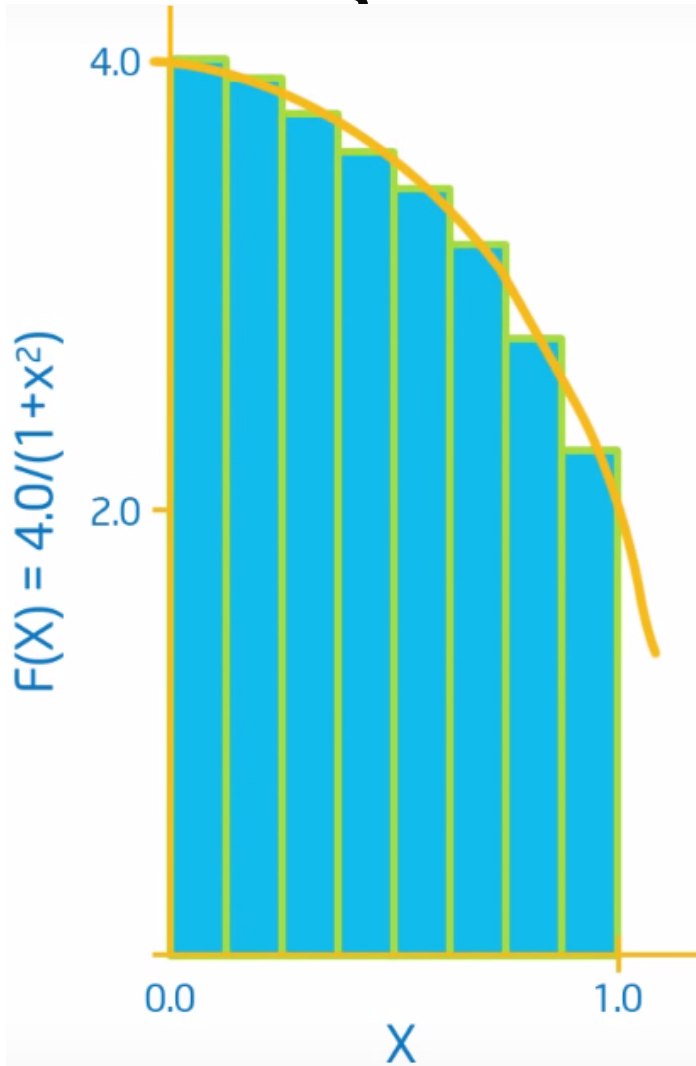




# PROGRAMAÇÃO PARALELA OPENMP — AULA 02

Marco A. Zanata Alves

# EXERCÍCIOS 2 A 4: INTEGRAÇÃO NUMÉRICA



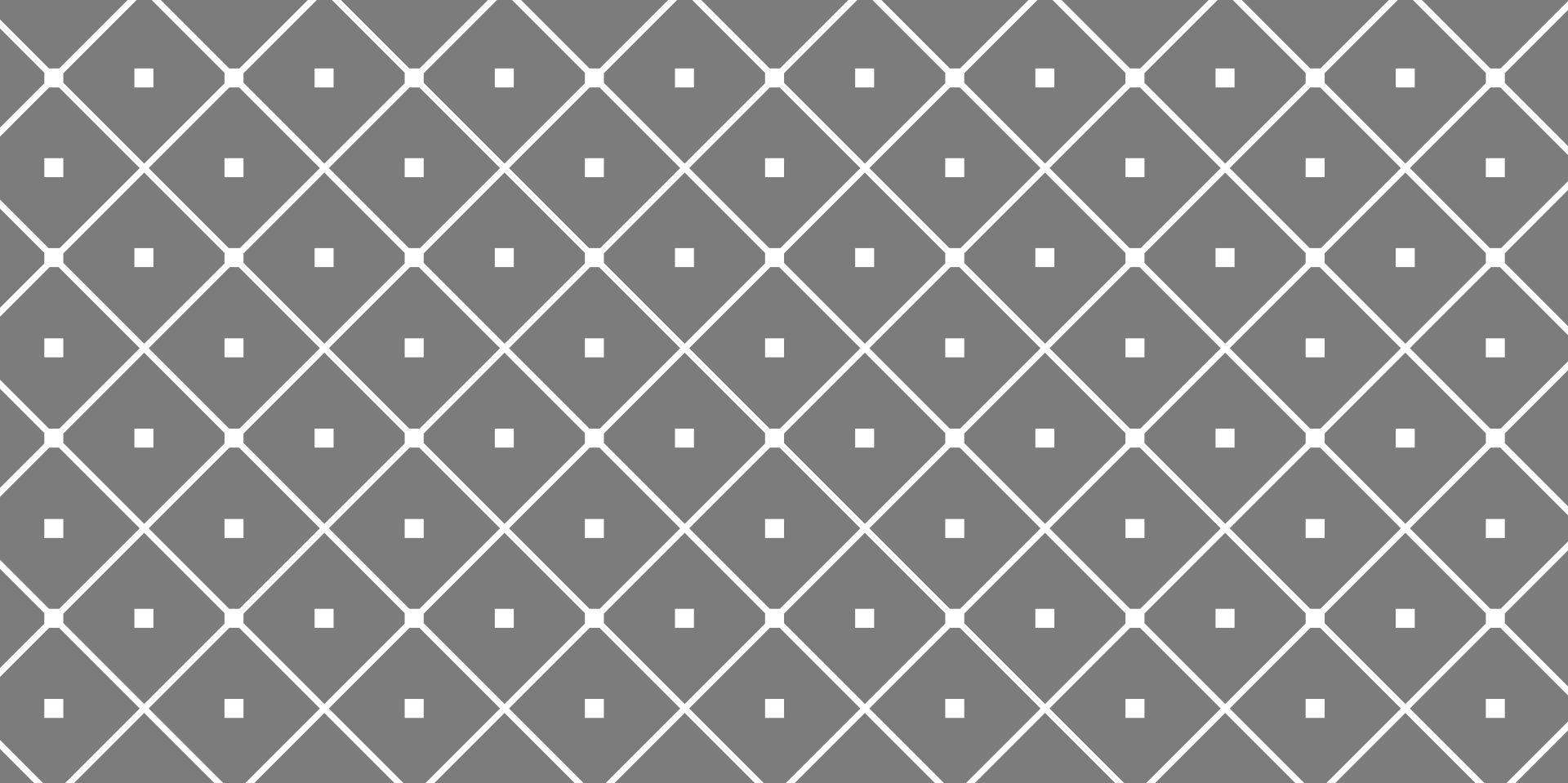
Matematicamente, sabemos que:

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

Podemos aproximar essa integral como a soma de retângulos:

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

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



# UM SIMPLES PROGRAMA PI E PORQUE ELE NÃO PRESTA

## EXERCÍCIOS 2 A 4: PROGRAMA PI SERIAL

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

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

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

Usamos uma variável global para evitar perder dados

```

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

```


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

Sempre verifique o # de threads

```

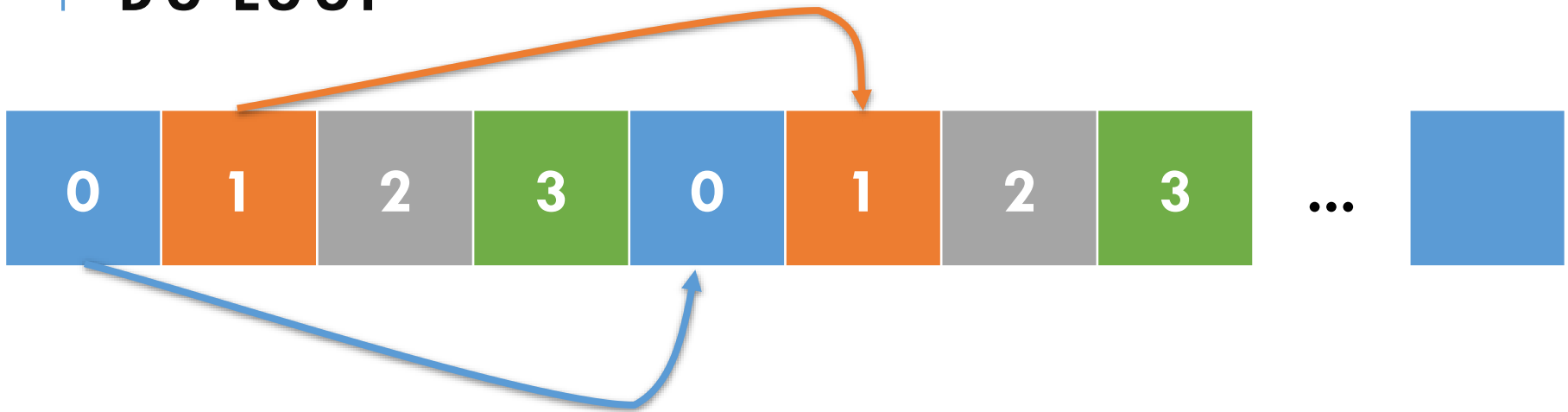
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main () {
    int i, nthreads; double pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        int i, id, nthrds; double x;
        id = omp_get_thread_num();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

```



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

# 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 nos  $P$  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).**

# RESULTADOS\*

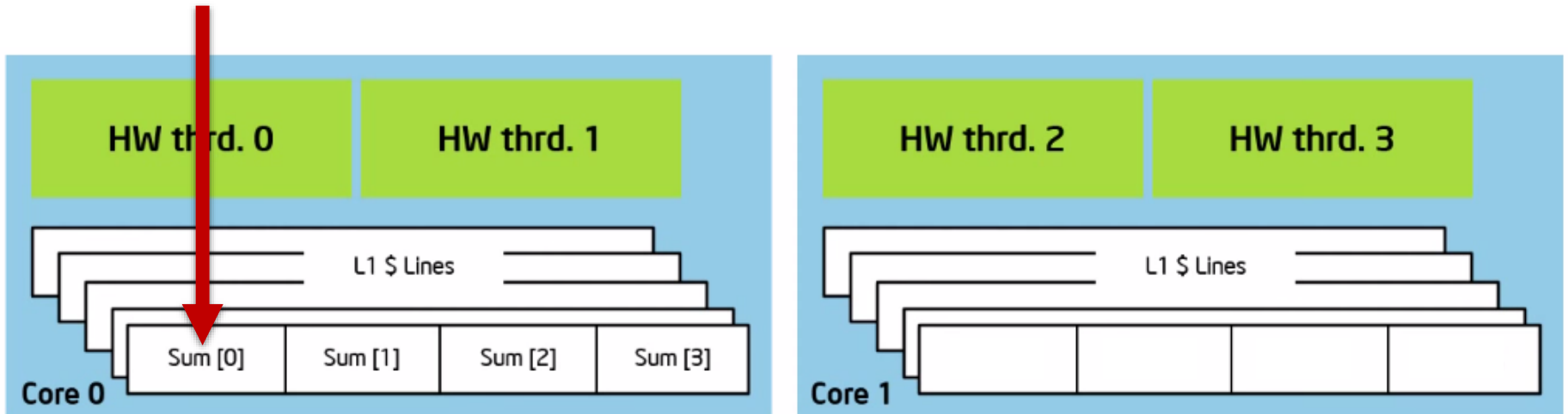
O Pi original sequencial com 100mi passos, executou em **1.83** seg.

\*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	S(p)
1	1.86	1,00
2	1.03	1,80
3	1.08	1,72
4	0.97	1,91

# O MOTIVO DESSA FALTA DE DESEMPENHO? FALSO COMPARTILHAMENTO

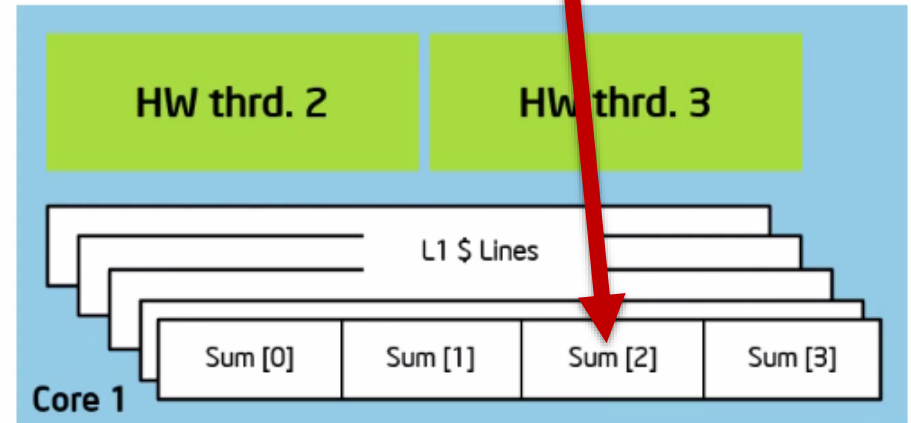
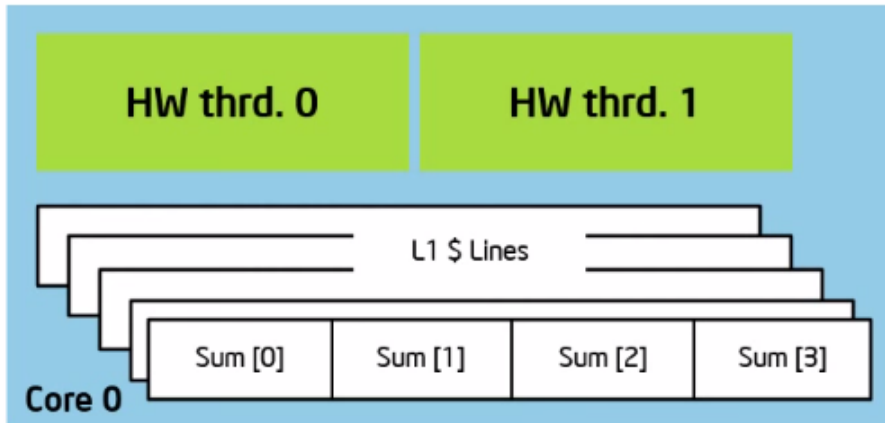
`sum[0] += 4.0/(1.0+x*x);`



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

# O MOTIVO DESSA FALTA DE DESEMPENHO? FALSO COMPARTILHAMENTO

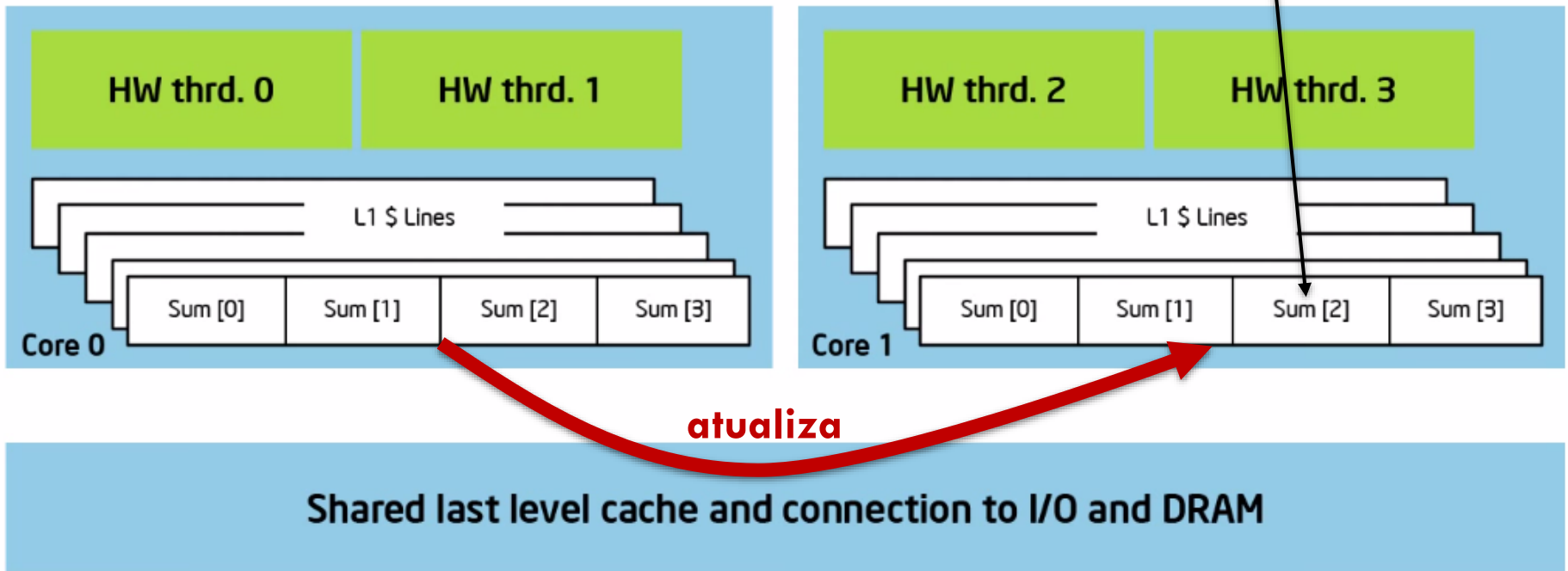
`sum[2] += 4.0/(1.0+x*x);`



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

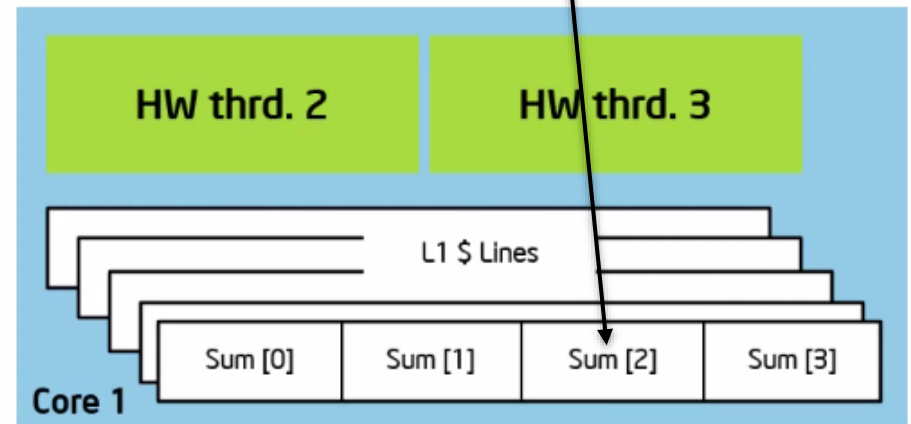
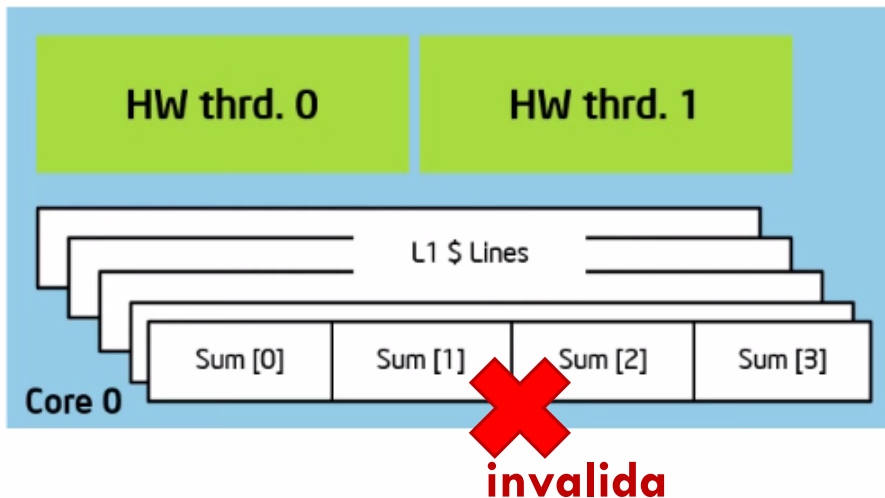
# O MOTIVO DESSA FALTA DE DESEMPENHO? FALSO COMPARTILHAMENTO

`sum[2] += 4.0/(1.0+x*x);`



# O MOTIVO DESSA FALTA DE DESEMPENHO? FALSO COMPARTILHAMENTO

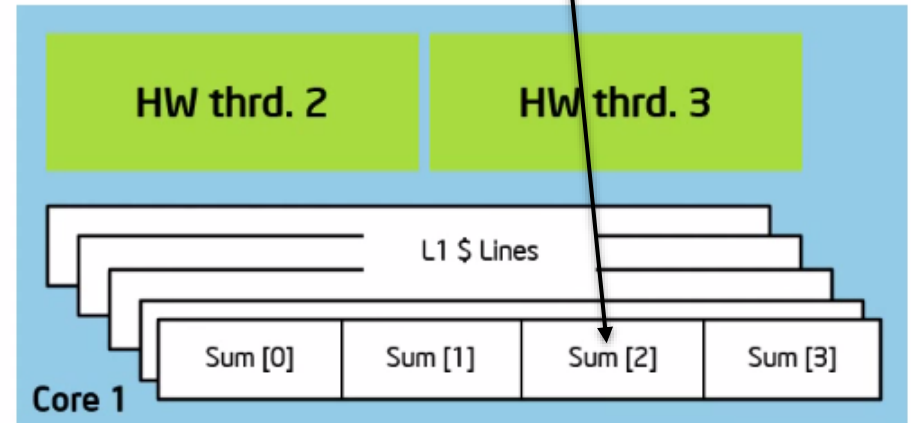
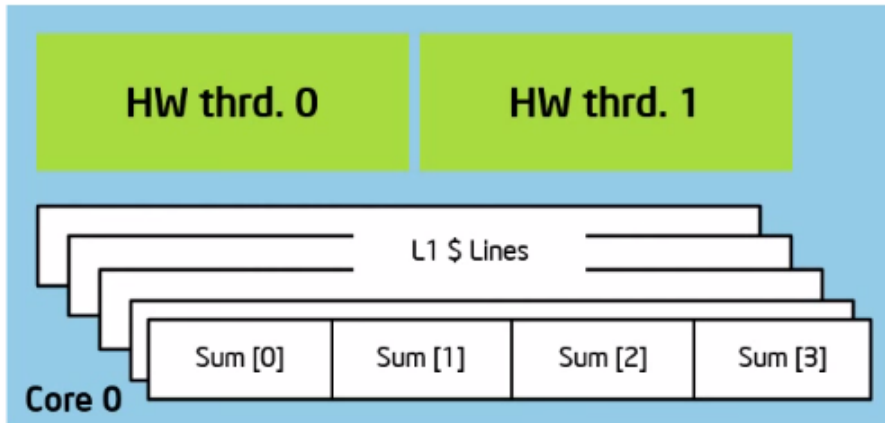
`sum[2] += 4.0/(1.0+x*x);`



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

# O MOTIVO DESSA FALTA DE DESEMPENHO? FALSO COMPARTILHAMENTO

`sum[2] += 4.0/(1.0+x*x);`

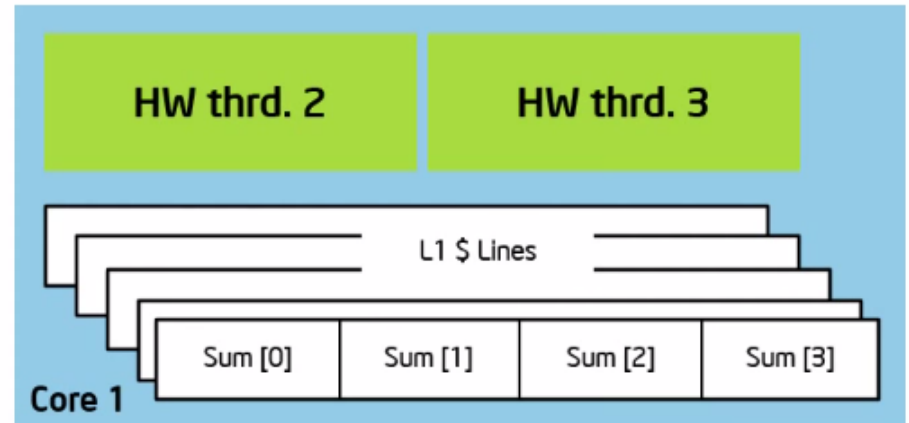
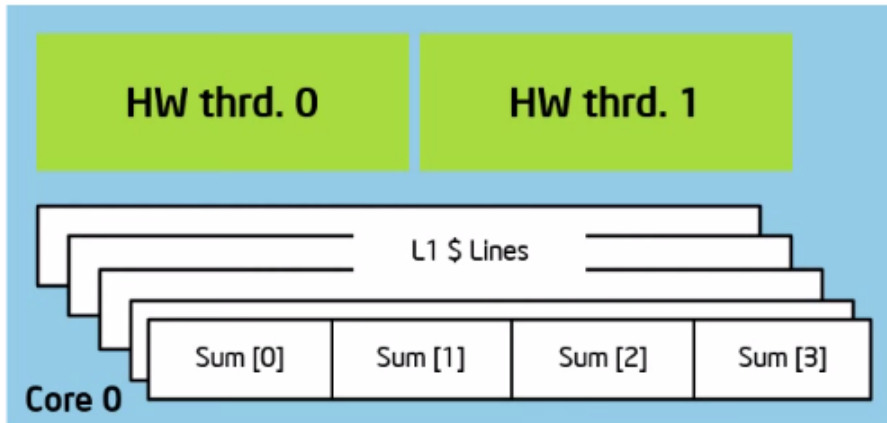


**escreve**

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

# O MOTIVO DESSA FALTA DE DESEMPENHO? FALSO COMPARTILHAMENTO

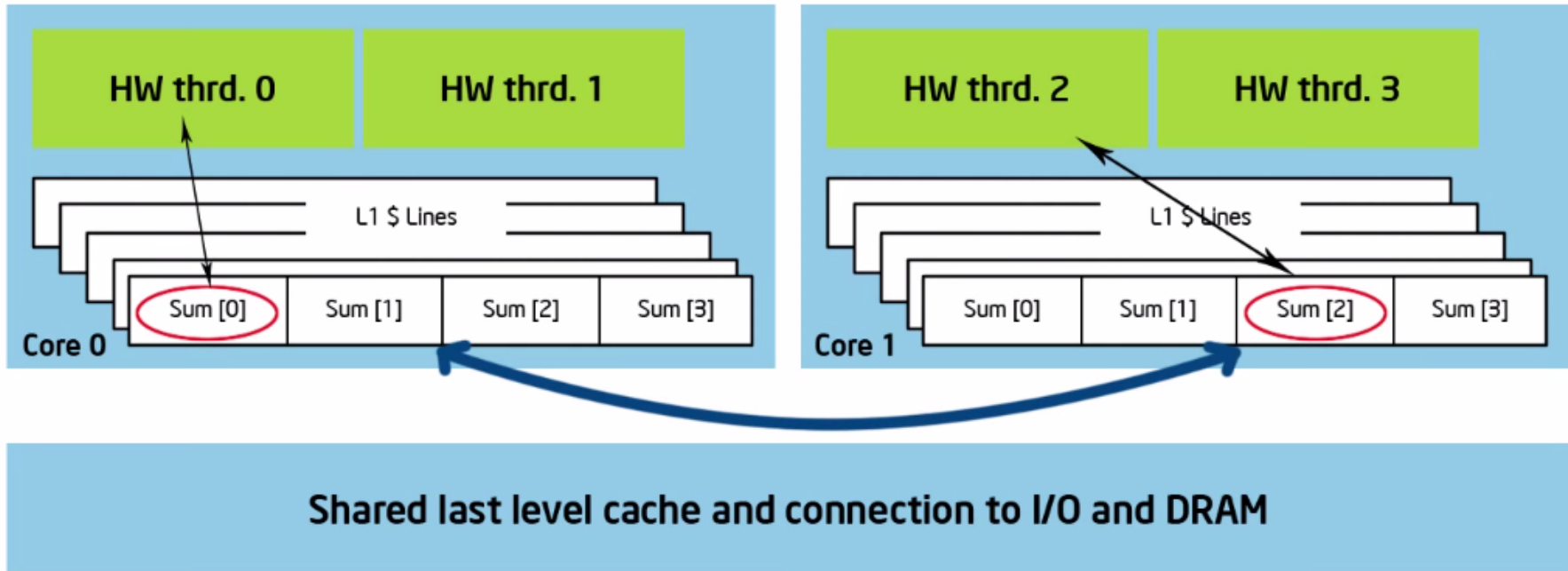
Se acontecer de elementos de dados independentes serem alocados em uma mesma linha de cache, cada atualização irá causar que a linha de cache fique em **ping-pong entre as caches...** Isso é chamado de "falso compartilhamento".



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



# O MOTIVO DESSA FALTA DE DESEMPENHO? FALSO COMPARTILHAMENTO



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

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

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

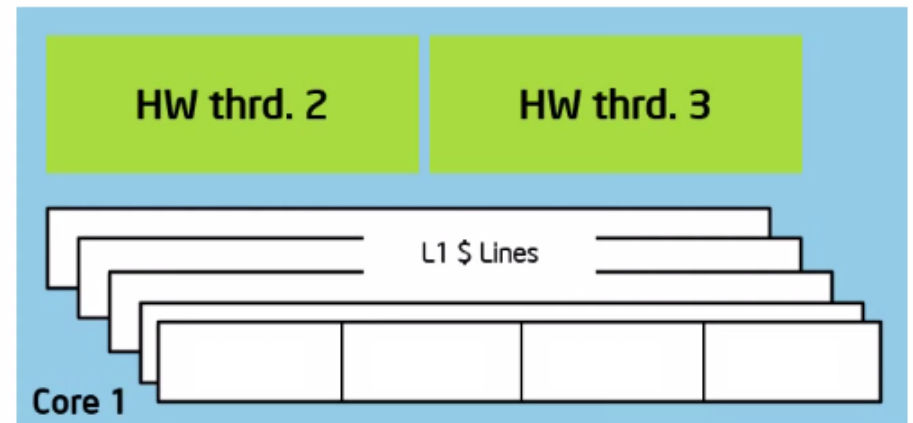
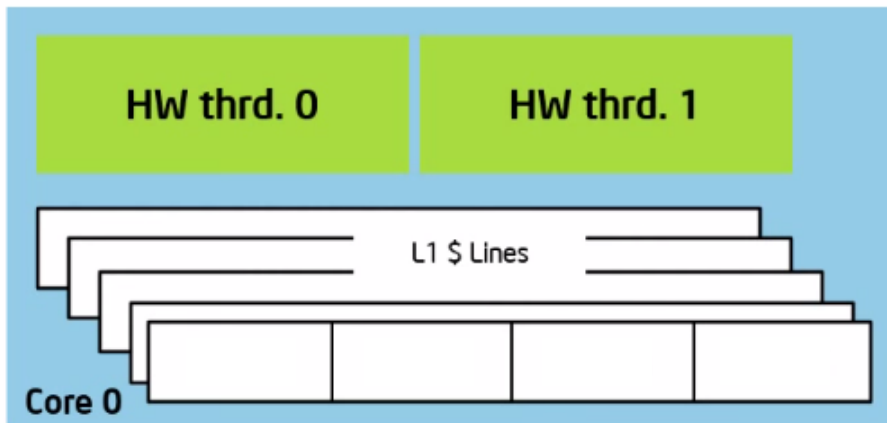
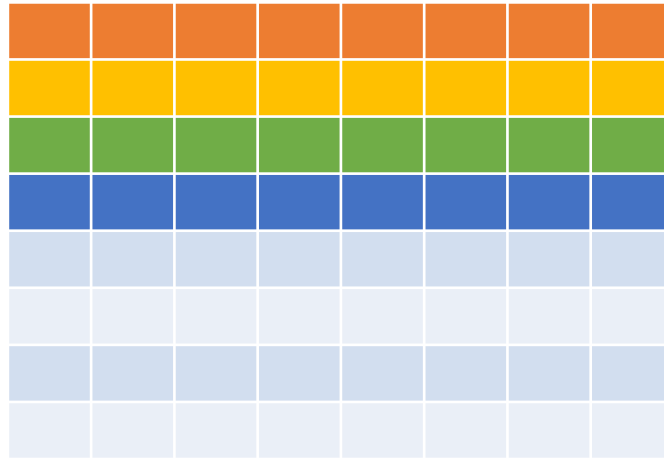
```
#include <omp.h>
static long num_steps = 100000; double step;
#define PAD 8 // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main () {
int i, nthreads; double pi, sum[NUM_THREADS][PAD];
step = 1.0/(double) num_steps;
#pragma omp parallel num_threads(NUM_THREADS)
{
int i, id, nthrds;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

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

Precisamos saber qual o tamanho da linha de cache de nosso processador... Usualmente 64 bytes

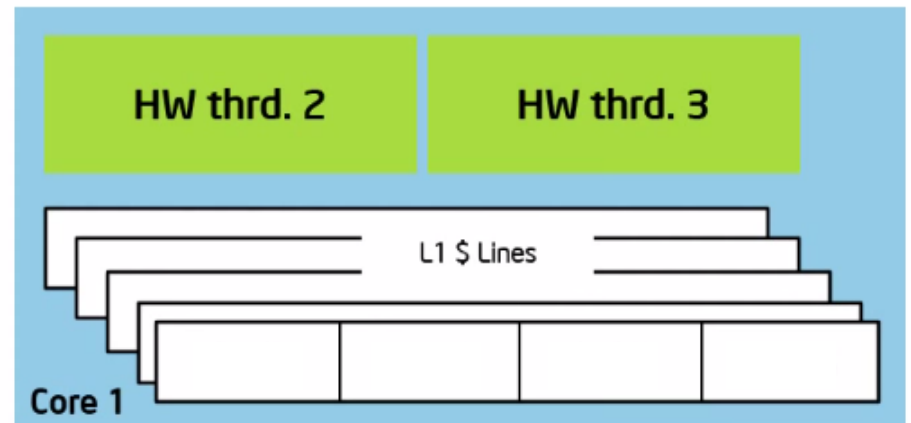
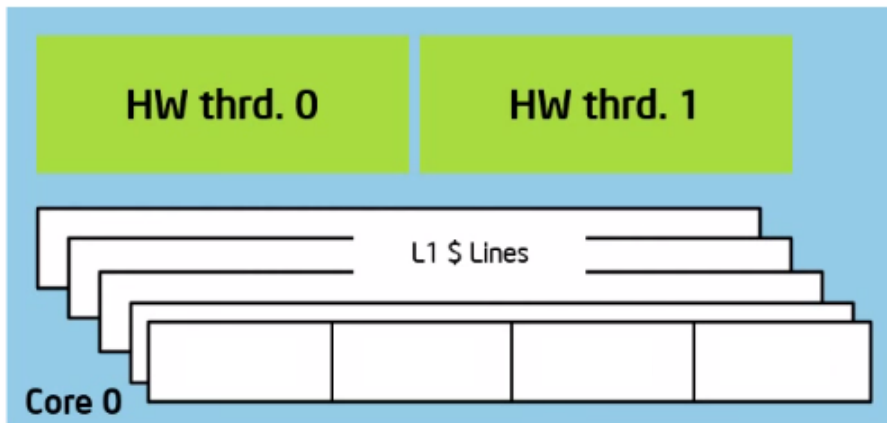
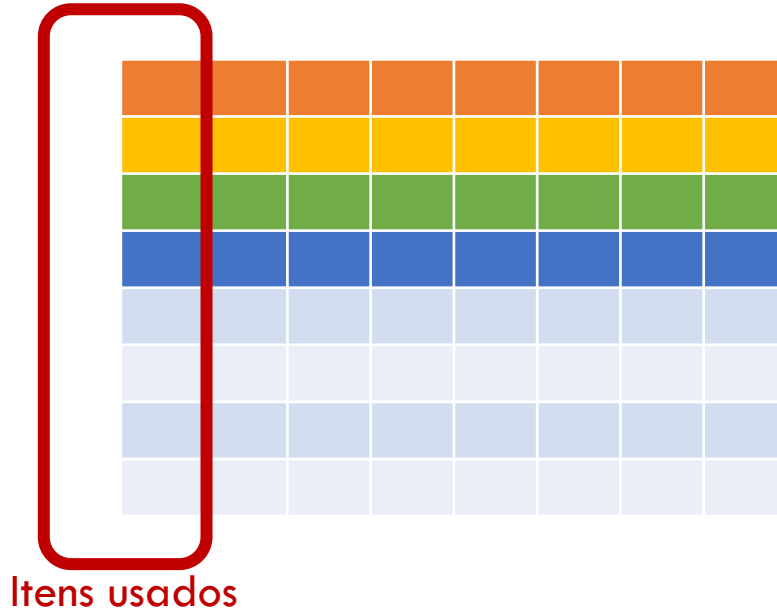
```
double sum[NUM_THREADS][PAD];
```

# PADDING



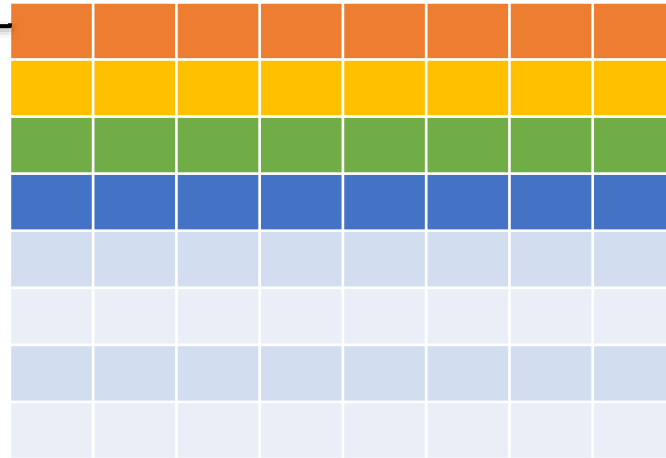
```
double sum[NUM_THREADS][PAD];
```

# PADDING

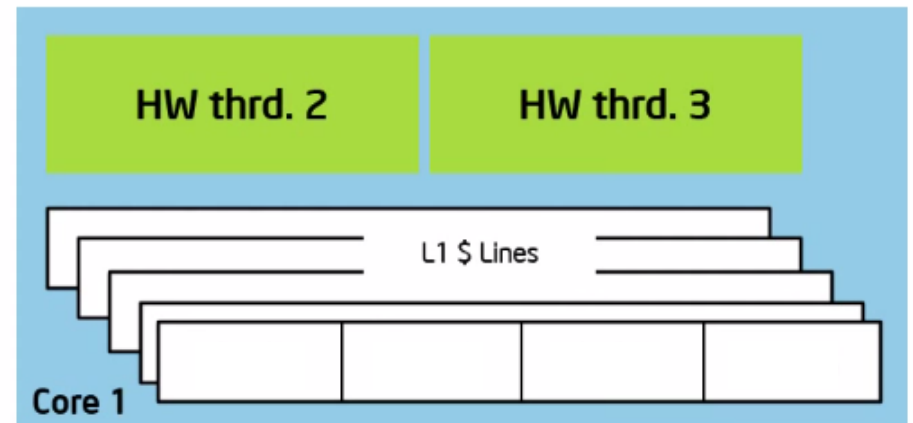
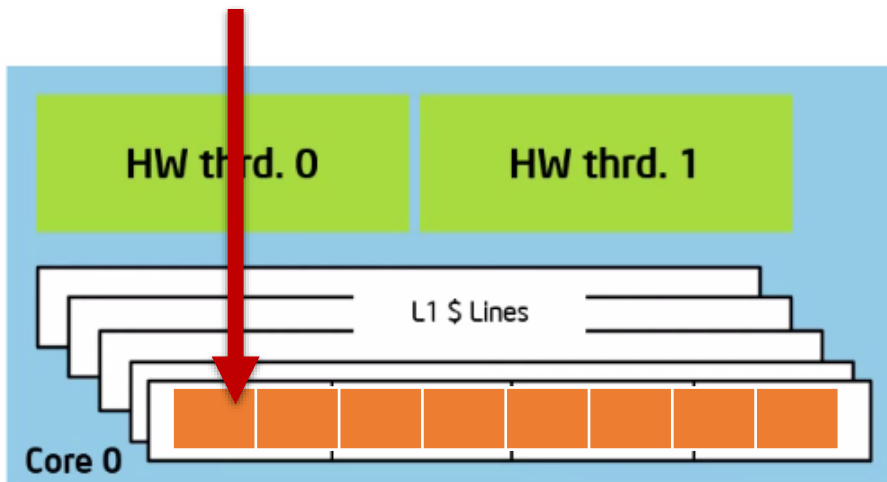


```
double sum[NUM_THREADS][PAD];
```

PADDING



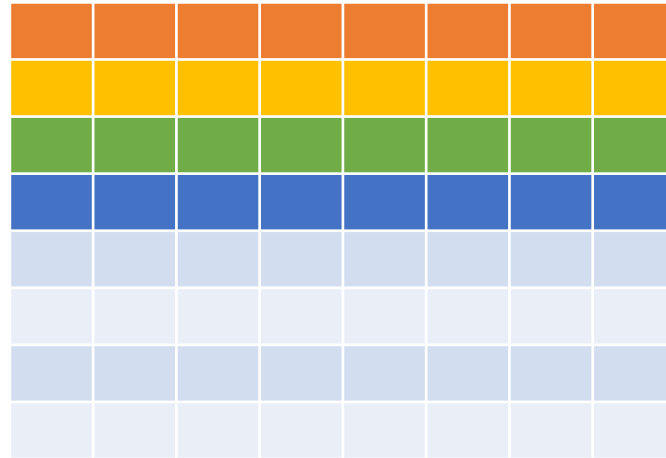
```
sum[0][0] += 4.0/(1.0+x*x);
```



```
double sum[NUM_THREADS][PAD];
```

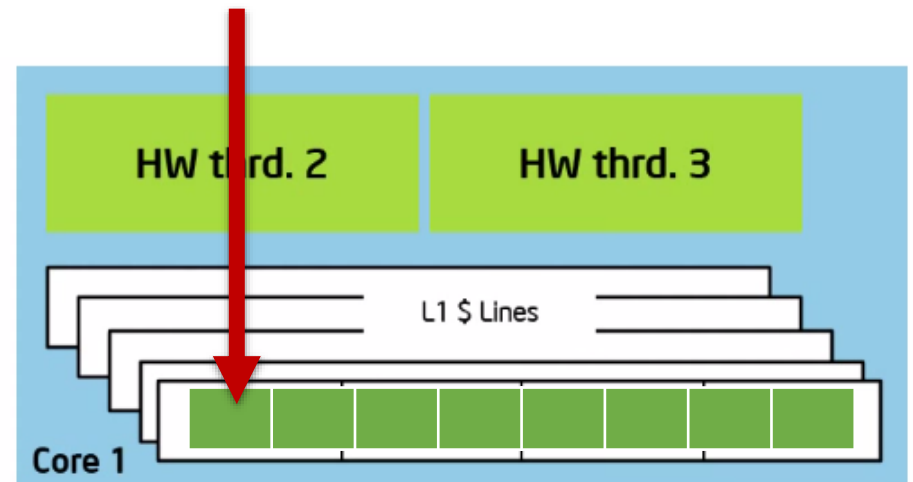
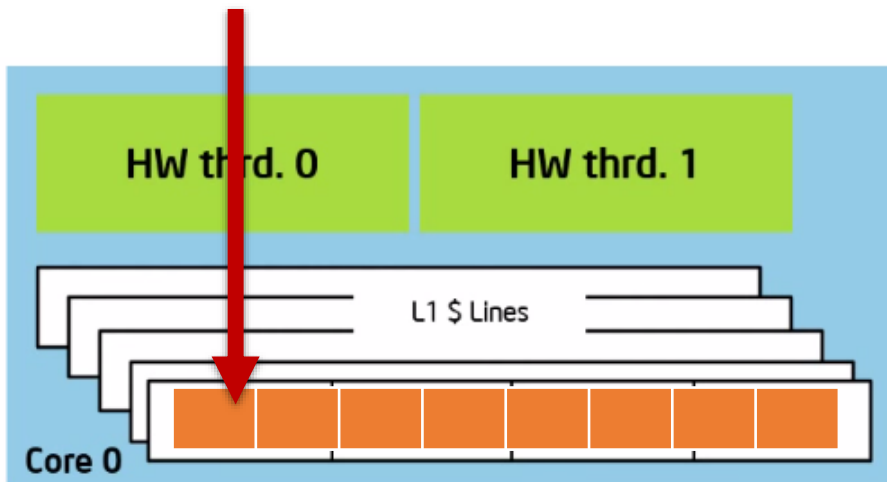
# PADDING

Eliminamos o falso compartilhamento!



```
sum[0][0] += 4.0/(1.0+x*x);
```

```
sum[2][0] += 4.0/(1.0+x*x);
```



# RESULTADOS\*

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

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

Threads	1 SPMD	1 SPMD padding	S(p)
1	1.86	1.86	1,00
2	1.03	1.01	1,84
3	1.08	0.69	2,69
4	0.97	0.53	3,50

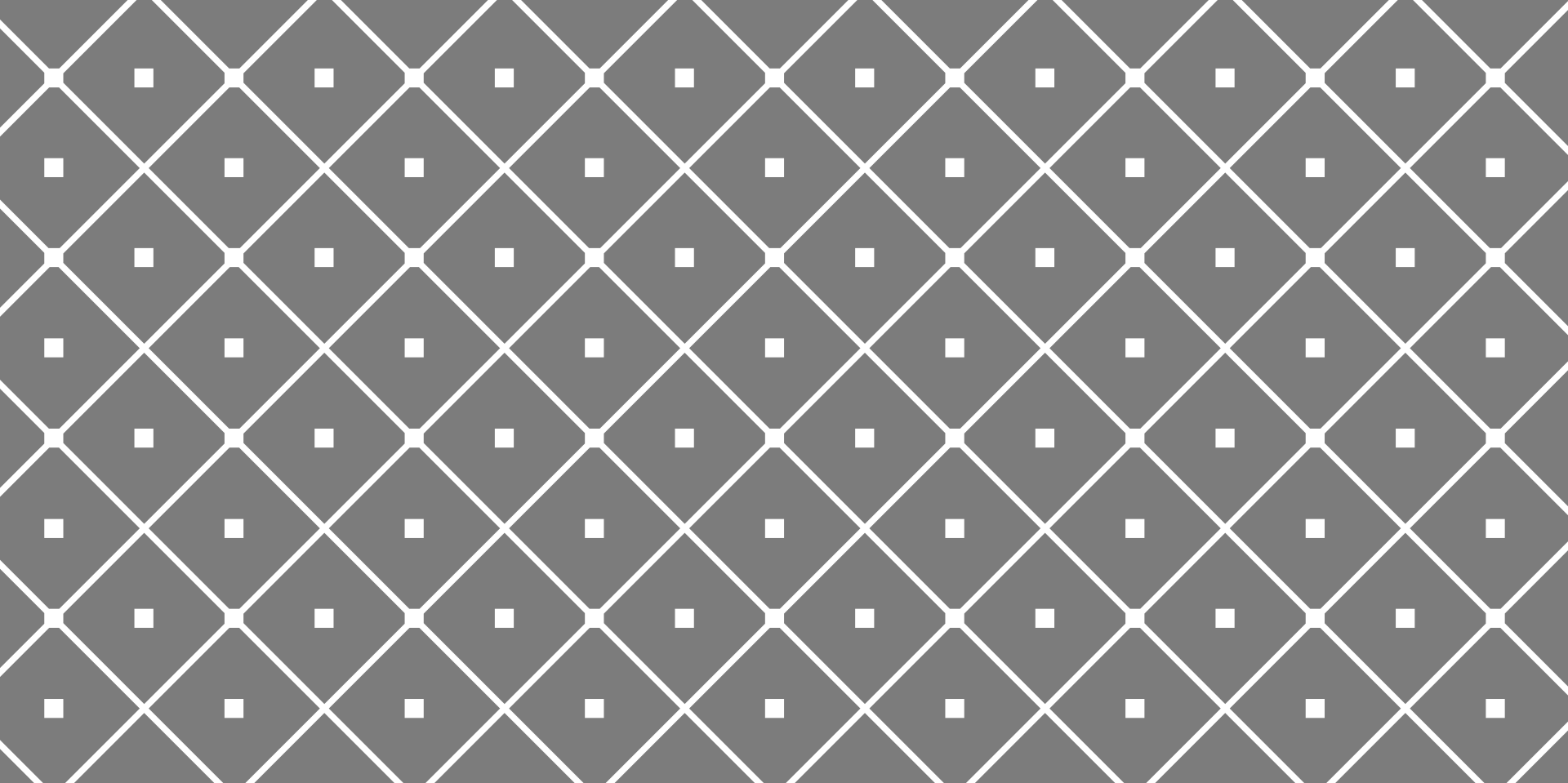
# REALMENTE PRECISAMOS ESPAÇAR NOSSOS VETORES?

## **Aquilo foi feio!**

- Espaçar vetores requer conhecimento profundo da arquitetura de cache.
- Mova seu programa para uma máquina com tamanho de linhas de cache diferente, e o desempenho desaparece.
- Existe desperdício de espaço de armazenamento.

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





# SINCRONIZAÇÃO (REVISITANDO O PROGRAMA PI)

# VISÃO GERAL DO OPENMP: COMO AS THREADS INTERAGEM?

Threads comunicam-se através de variáveis compartilhadas.

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

(exemplo do saldo bancário, aula passada)

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

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

# SINCRONIZAÇÃO

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

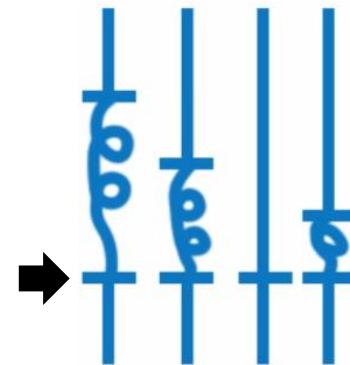
As duas formas mais comuns de sincronização são:

# SINCRONIZAÇÃO

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

As duas formas mais comuns de sincronização são:

**Barreira:** Cada thread espera na barreira até a chegada de todas as demais



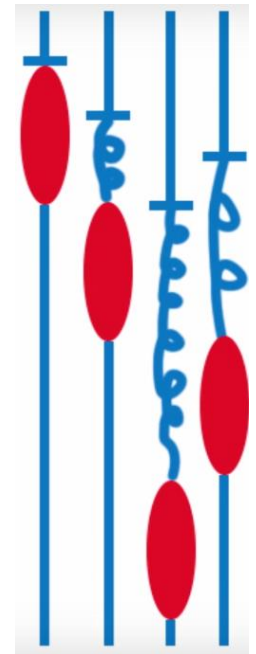
# SINCRONIZAÇÃO

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

As duas formas mais comuns de sincronização são:

**Barreira:** Cada thread espera na barreira até a chegada de todas as demais

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



# SINCRONIZAÇÃO

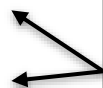
## Sincronização de alto nível:

- critical
- atomic
- barrier
- ordered

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

## Sincronização de baixo nível:

- flush
- locks (both simple and nested)



Vamos falar sobre esses mais tarde!

# SINCRONIZAÇÃO: BARRIER

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

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

    #pragma omp barrier

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

# SINCRONIZAÇÃO: CRITICAL

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

```
float res;  
#pragma omp parallel  
{ float B; int i, id, nthrds;  
  id = omp_get_thread_num();  
  nthrds = omp_get_num_threads();  
  for(i=id; i<niters; i+=nthrds){  
    B = big_job(i); // Se for pequeno, muito overhead  
    #pragma omp critical  
      res += consume (B);  
  }  
}
```

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



# SINCRONIZAÇÃO : ATOMIC (FORMA BÁSICA)

Formas adicionais foram incluídas no OpenMP 3.1.

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

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

Usará uma  
instrução  
especial se  
disponível

A declaração dentro de atomic deve ser uma das seguintes:

**x Op= expr**

**x++**

**++x**

**x--**

**--x**

**x** é um valor escalar

**Op** é um operador não sobrecarregado.

# EXERCÍCIO 3

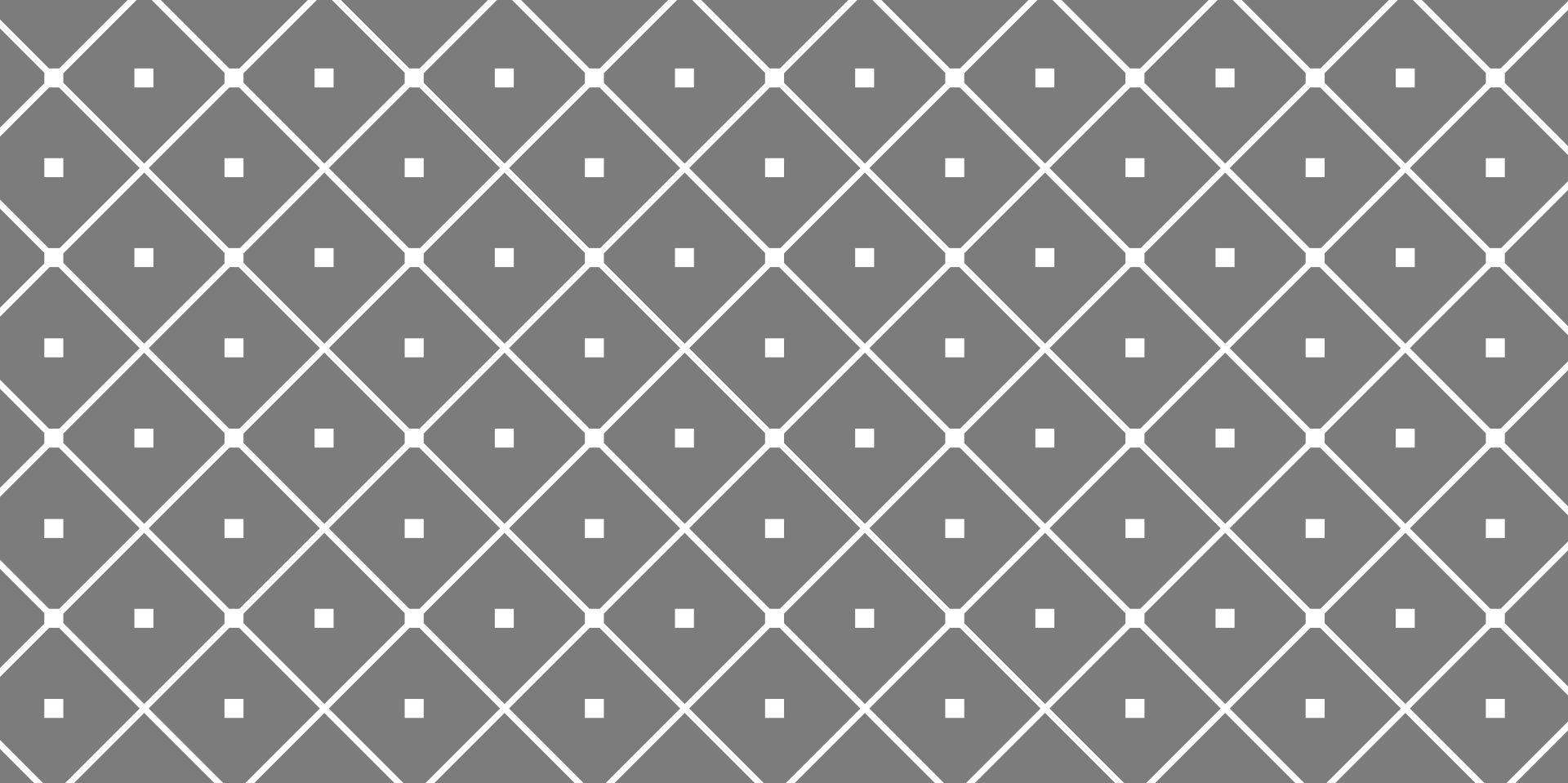
No exercício 2, provavelmente foi usado um vetor para cada thread armazenar o valor de sua soma parcial.

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

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

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

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



# OVERHEAD DE SINCRONIZAÇÃO E ELIMINAÇÃO DE FALSO COMPARTILHAMENTO

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

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

Cria um escalar local para cada thread acumular a soma parcial

Sem vetor, logo sem falso compartilhamento

A variável soma estará fora de escopo quando sairmos da região paralela. Logo, devemos somar aqui. Vamos proteger a soma com uma região crítica para que as atualizações não conflitem

# RESULTADOS\*

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

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

Threads	1. SPMD	1. SPMD padding	SPMD critical	S(p)
1	1.86	1.86	1.87	1,00
2	1.03	1.01	1.00	1,86
3	1.08	0.69	0.68	2,73
4	0.97	0.53	0.53	3,50

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

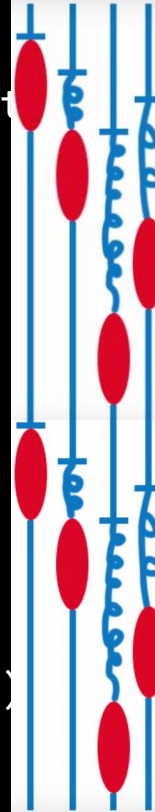
```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main () {
    double pi = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    { int i, id, nthrds; double x;
      id = omp_get_thread_num();
      nthrds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

Atenção onde você irá colocar a seção crítica

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

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

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



Atenção onde você irá colocar a seção crítica

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

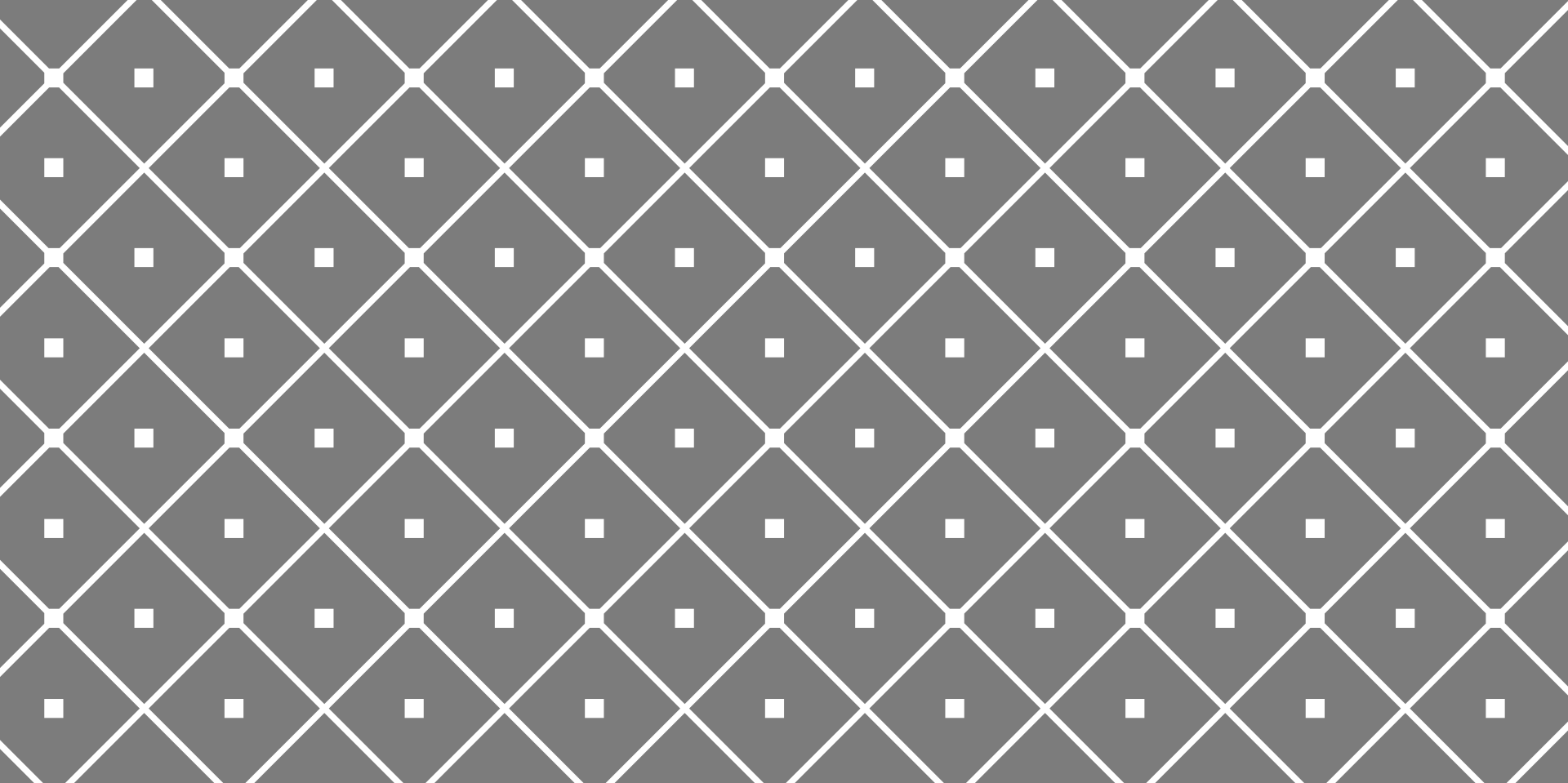
Tempo execução sequencial + overhead

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

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

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





# LAÇOS PARALELOS (SIMPLIFICANDO O PROGRAMA PI)

# SPMD VS. WORKSHARING

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

**Como dividir** os caminhos dentro do código entre as threads?

Isso é chamado de **worksharing** (divisão de trabalho)

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

Veremos mais tarde

# CONSTRUÇÕES DE DIVISÃO DE LAÇOS

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

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

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

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

# CONSTRUÇÕES DE DIVISÃO DE LAÇOS

## UM EXEMPLO MOTIVADOR

Código sequencial

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

Região OpenMP parallel

???

# CONSTRUÇÕES DE DIVISÃO DE LAÇOS

## UM EXEMPLO MOTIVADOR

Código sequencial

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

Região OpenMP parallel

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

# CONSTRUÇÕES DE DIVISÃO DE LAÇOS

## UM EXEMPLO MOTIVADOR

Código sequencial

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

Região OpenMP parallel

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

**Região paralela OpenMP  
com uma construção de  
divisão de trabalho**

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

# CONSTRUÇÕES PARALELA E DIVISÃO DE LAÇOS COMBINADAS

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


```
double res[MAX]; int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i< MAX; i++) {
        res[i] = huge();
    }
}
```

=

```
double res[MAX]; int i;
#pragma omp parallel for
    for (i=0; i< MAX; i++) {
        res[i] = huge();
    }
```

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

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



Como o laço  
será mapeado  
para as  
threads?



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

`schedule(static [,chunk])`

- Distribui iterações de tamanho "chunk" para cada thread

`schedule(dynamic[,chunk])`

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

`schedule(guided[,chunk])`

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

`schedule(runtime)`

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

`schedule(auto)` ← "Novo"

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

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

Tipo de Schedule	Quando usar
STATIC	Pré-determinado e previsível pelo programador
DYNAMIC	Imprevisível, quantidade de trabalho por iteração altamente variável
GUIDED	Caso especial do dinâmico para reduzir o overhead dinâmico
AUTO	Quando o tempo de execução pode "aprender" com as iterações anteriores do mesmo laço

Menos trabalho durante a execução (mais tempo durante a compilação)

Mais trabalho durante a execução (lógica complexa de controle)

# TRABALHANDO COM LAÇOS

## Abordagem básica:

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

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

Onde está a  
dependência  
aqui?

# TRABALHANDO COM LAÇOS

## Abordagem básica:

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

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

Note que o índice  
"i" será privado  
por padrão

Remove a  
dependência  
dentro do laço

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

# LAÇOS ANINHADOS

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

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

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

# EXERCÍCIO 4: PI COM LAÇOS

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

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

# REDUÇÃO

Como podemos proceder nesse caso?

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

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

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

# REDUÇÃO

A diretiva OpenMP reduction: `reduction (op : list)`

Dentro de uma região paralela ou de divisão de trabalho:

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

A variável na "lista" deve ser compartilhada entre as threads.

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



# REDUÇÃO

## OPERANDOS E VALORES INICIAIS

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

Valores iniciais são os que fazer sentido (elemento nulo)

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

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

Apenas para  
C e C++

# EXERCÍCIO 5: PI COM LAÇOS

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

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