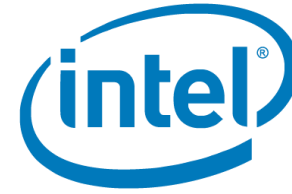




INTEL MODERN CODE PARTNER

EQUIPE



Philippe O. A. Navaux – GPPD – UFRGS

Marco A. Z. Alves – LSE – UFPR

Matthias Diener – GPPD – UFRGS

Eduardo H. M. Cruz – GPPD – UFRGS

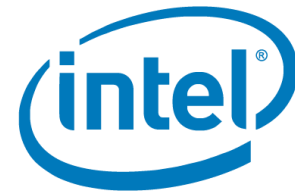
Jean Bez – GPPD – UFRGS

Matheus S. Serpa – GPPD – UFRGS

Demais membros do GPPD.



INTRODUÇÃO AO OPENMP



INTRODUÇÃO

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

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

Observações:

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

SINTAXE BÁSICA - OPENMP

Tipos e protótipos de funções no arquivo:

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

A maioria das construções OpenMP são diretivas de compilação.

```
#pragma omp construct [clause [clause]...]
```

- Exemplo:

```
#pragma omp parallel private(var1, var2) shared(var3, var4)
```

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

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

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

NOTAS DE COMPILAÇÃO

Linux e OS X com gcc or intel icc:

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

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

```
export OMP_NUM_THREADS=32
```

```
./a.out
```

Para shell bash

Por padrão é o nº
de proc. virtuais.

Também
funciona
no
Windows!

Até
mesmo no
Visual
Studio!

Mas
vamos
usar
Linux 😊

EXERCÍCIO 1: HELLO WORLD

Programa sequencial.

```
#include <stdio.h>

int main(){
    int myid, nthreads;
```

```
    myid = 0;
```

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

```
    return 0;
```

```
}
```

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

FUNÇÕES

Funções da biblioteca OpenMP.

```
include <omp.h> // runtime library routines for C/C++
```

```
int omp_get_thread_num(); // the thread number (0 to T-1)  
// set / get number of threads  
void omp_set_num_threads(int num_threads);  
int omp_get_num_threads();
```

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


DIRETIVAS

Diretivas do OpenMP.

```
#pragma omp parallel private(...) shared(...)
{
}
```

```
#pragma omp single // block is executed by only one thread
```

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

RESUMO

Diretivas e funções do OpenMP.

```
include <omp.h> // runtime library routines for C/C++

#pragma omp parallel private(...) shared(...)
{
}

#pragma omp single // block is executed by only one thread

int omp_get_thread_num(); // the thread number (0 to T-1)
// set / get number of threads
void omp_set_num_threads(int num_threads);
int omp_get_num_threads();

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

EXERCÍCIO 1: HELLO WORLD

Torne o programa abaixo multithreaded.

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

```
int main(){  
    int myid, nthreads;
```

```
    myid = 0;
```

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

```
    return 0;
```

```
}
```

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

SOLUÇÃO 1.1: HELLO WORLD

Variáveis privadas.

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

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

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

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

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

```
}
```

```
    return 0;
```

```
}
```

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

```
./hello
```

```
0 of 2 - hello world!
```

```
1 of 2 - hello world!
```

SOLUÇÃO 1.2: HELLO WORLD

Variáveis privadas e compartilhadas.

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

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

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

SOLUÇÃO 1.3: HELLO WORLD

NUM_THREADS fora da região paralela.

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

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

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

~~SOLUÇÃO 1.3~~: HELLO WORLD

NUM_THREADS fora da região paralela.

Não funciona.

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

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

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

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

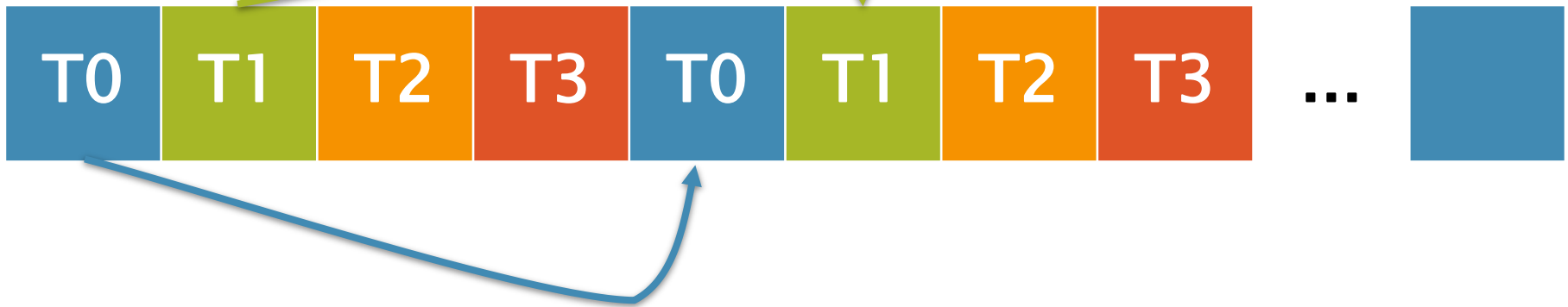
```
1 of 1 - hello world!
```

EXERCÍCIO 2, PARTE A: VECTOR SUM

2-vsum.c make ./2-vsum.exec 15

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


DISTRIBUIÇÃO CÍCLICA DE ITERAÇÕES DO LAÇO



```
// Distribuição cíclica  
for(i = myid; i < n; i += nthreads)
```

RESUMO

Diretivas e funções do OpenMP.

```
include <omp.h> // runtime library routines for C/C++

#pragma omp parallel default(shared) private(...)
{
}

#pragma omp single // block is executed by only one thread

int omp_get_thread_num(); // the thread number (0 to T-1)
// set / get number of threads
void omp_set_num_threads(int num_threads);
int omp_get_num_threads();

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

EXERCÍCIO 2, PARTE A: VECTOR SUM

2-vsum.c make ./2-vsum.exec 15

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

SOLUÇÃO 2.1, PARTE B: VECTOR SUM

2.1-vsum-wrong.c make ./2.1-vsum-wrong.exec 15

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

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

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

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

~~SOLUÇÃO 2.1~~, PARTE B: VECTOR SUM

2.1-vsum-wrong.c make ./2.1-vsum-wrong.exec 15

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

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

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

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

~~SOLUÇÃO 2.1~~, PARTE B: VECTOR SUM

2.1-vsum-wrong.c make ./2.1-vsum-wrong.exec 15

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


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

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

COMO AS THREADS INTERAGEM?

OpenMP é um modelo de *multithreading* de memória compartilhada.

- Threads se comunicam através de variáveis compartilhadas.



Compartilhamento não intencional de dados causa **condições de corrida**.

- Condições de corrida: quando a saída do programa muda quando a threads são escalonadas de forma diferente.

Apesar de este ser um aspectos mais poderosos da utilização de threads, também pode ser um dos mais problemáticos.



O problema existe quando dois ou mais *threads* tentam acessar/alterar as mesmas estruturas de dados (condições de corrida).

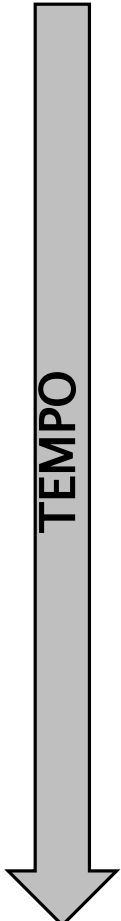
Para controlar condições de corrida:

- Usar sincronização para proteger os conflitos por dados

Sincronização é cara, por isso:

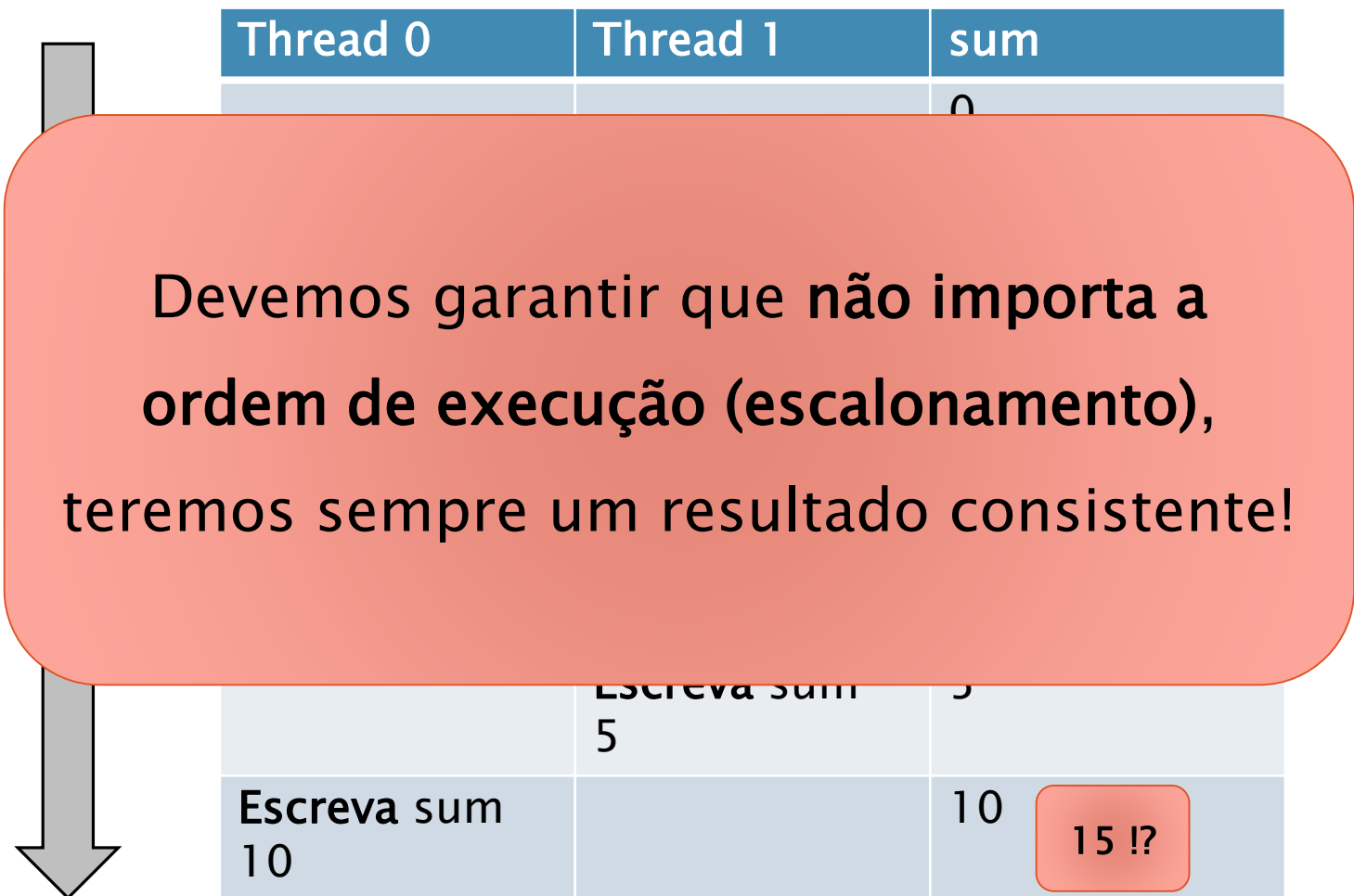
- Tentaremos mudar a forma de acesso aos dados para minimizar a necessidade de sincronizações.

CONDIÇÕES DE CORRIDA: EXEMPLO



Thread 0	Thread 1	sum
		0
Leia sum 0		0
	Leia sum 0	0
	Some 5 5	0
Some 10 10		0
	Escreva sum 5	5
Escreva sum 10		10
		15 !?

CONDIÇÕES DE CORRIDA: EXEMPLO



The diagram illustrates a race condition scenario. A vertical grey bar on the left has an upward arrow at the top and a downward arrow at the bottom, representing the execution timeline. A large red rounded rectangle is overlaid on the table, containing text about execution order. The table below shows the state of two threads and a shared sum variable.

Thread 0	Thread 1	sum
		0
Devemos garantir que não importa a ordem de execução (escalonamento), teremos sempre um resultado consistente!		
	Escreva sum 5	5
Escreva sum 10		10
		15 !?

SINCRONIZAÇÃO

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

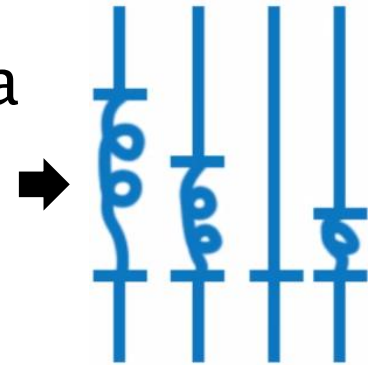
As duas formas mais comuns de sincronização são:

SINCRONIZAÇÃO

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

As duas formas mais comuns de sincronização são:

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



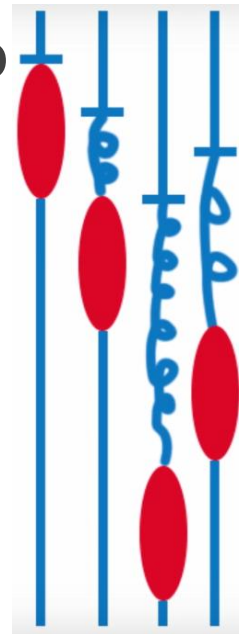
SINCRONIZAÇÃO

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

As duas formas mais comuns de sincronização são:

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

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



SINCRONIZAÇÃO: BARRIER

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

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

    #pragma omp barrier

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

SINCRONIZAÇÃO: CRITICAL

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

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

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

SINCRONIZAÇÃO: ATOMIC

`atomic` prove exclusão mútua para operações específicas.

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

Instruções especiais
da arquitetura (se
disponível)

Algumas operações aceitáveis:

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

EXERCÍCIO 2, PARTE C: VECTOR SUM

2.1-vsum-wrong.c make ./2.1-vsum-wrong.exec 15

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

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

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

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


SOLUÇÃO 2.2, PARTE C: VECTÖR SUM

2.2-vsum-critical.c make ./2.2-vsum-critical.exec 15

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

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

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

SOLUÇÃO 2.3, PARTE C: VECTÖR SUM

2.3-vsum-atomic.c make ./2.3-vsum-atomic.exec 15

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

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

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

EXERCÍCIO 2, PARTE D: VECTOR SUM

2.3-vsum-atomic.c make ./2.3-vsum-atomic.exec 15

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

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

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

Qual o problema da seção crítica dentro do loop?

EXERCÍCIO 2, PARTE D: VECTOR SUM

2.3-vsum-atomic.c make ./2.3-vsum-atomic.exec 15

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

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

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

Qual o problema da seção crítica dentro do loop?

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

SOLUÇÃO 2.4, PARTE D: VECTOR SUM

2.4-vsum-atomic-improved.c make ./2.4-vsum-atomic-improved.exec 15

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

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

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

SOLUÇÃO 2.4, PARTE D: VECTOR SUM

2.4-vsum-atomic-improved.c make ./2.4-vsum-atomic-improved.exec 15

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

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

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

Regiões atomic – nthreads vezes. Ex. 32

EXERCÍCIO 2, PARTE E: VECTOR SUM

2.4-vsum-atomic-improved.c make ./2.4-vsum-atomic-improved.exec 15

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

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

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

Existe uma solução melhor?

EXERCÍCIO 2, PARTE E: VECTOR SUM

2.4-vsum-atomic-improved.c make ./2.4-vsum-atomic-improved.exec 15

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

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

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

Existe uma solução melhor?

Estamos auxiliando a *cache*?

PRINCÍPIO DA LOCALIDADE

Programas repetem trechos de código e acessam repetidamente dados próximos.

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

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

ACESSOS INTERCALADOS

TEMPO

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

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

25% do conteúdo trazido para a cache é utilizado.

ACESSOS CONSECUTIVOS

TEMPO

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

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

100% do conteúdo trazido para a cache é utilizado.

EXERCÍCIO 2, PARTE E: VECTOR SUM

2.4-vsum-atomic-improved.c make ./2.4-vsum-atomic-improved.exec 15

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

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

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

SOLUÇÃO 2.5, PARTE E: VECTOR SUM

2.5-vsum-atomic-improved-cache.c make ./2.5-vsum-atomic-improved-cache.exec 15

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

EXERCÍCIO 2, PARTE F: VECTOR SUM

2.5-vsum-atomic-improved-cache.c make ./2.5-vsum-atomic-improved-cache.exec 15

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

OpenMP é um modelo relativamente fácil de usar

CONSTRUÇÕES DE DIVISÃO DE LAÇOS

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

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

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

CONSTRUÇÕES DE DIVISÃO DE LAÇOS

UM EXEMPLO MOTIVADOR

Código sequencial

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


CONSTRUÇÕES DE DIVISÃO DE LAÇOS UM EXEMPLO MOTIVADOR

Código sequencial

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

Região OpenMP parallel

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

CONSTRUÇÕES DE DIVISÃO DE LAÇOS

UM EXEMPLO MOTIVADOR

Código sequencial

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

Região OpenMP parallel

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

Região paralela OpenMP
com uma construção de
divisão de laço

```
#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

Algumas cláusulas podem ser combinadas.

```
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();
```

EXERCÍCIO 2, PARTE F: VECTOR SUM

2.5-vsum-atomic-improved-cache.c make ./2.5-vsum-atomic-improved-cache.exec 15

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

SOLUÇÃO 2.6, PARTE F: VECTÖR SUM

2.6-vsum-for.c make ./2.6-vsum-for.exec 15

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

{
                                sum_local = 0;

                                ...

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

EXERCÍCIO 2, PARTE G: VECTOR SUM

2.6-vsum-for.c make ./2.6-vsum-for.exec 15

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

{
    sum_local = 0;

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

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

REDUÇÃO

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

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

DIRETIVA REDUCTION

`reduction(op : list_vars)`

Dentro de uma região paralela ou de divisão de trabalho:

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

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

EXERCÍCIO 2, PARTE G: VECTOR SUM

2.6-vsum-for.c make ./2.6-vsum-for.exec 15

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

{
    sum_local = 0;

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

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

SOLUÇÃO 2.7, PARTE G: VECTOR SUM

2.7-vsum-for-reduction.c make ./2.7-vsum-for-reduction.exec 15

```
sum = 0;
```

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

VECTOR SUM

Sequencial vs. Paralelo

```
sum = 0;

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

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

RESULTADOS*

./2-vsum 50.exec, executou em 0.45 seg.

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

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

#	2.2 critical	2.3 atomic	2.4 atomic improved	2.5 atomic improved cache	2.6 for	2.7 for Reduction
1	0.45					
8	167.61	14.31	0.50	0.12	0.12	0.12
16	190.11	16.51	0.47	0.14	0.14	0.14
32	246.70	17.35	0.87	0.16	0.16	0.16

SPEEDUP

O speedup é uma medida do grau de desempenho. O speedup mede a razão entre o tempo de execução de duas soluções que resolvem o mesmo problema.

$$Speedup = \frac{T_{antigo}}{T_{novo}}$$

T_{antigo} é o tempo de execução de uma solução base

T_{novo} é o tempo de execução da nova solução proposta

SPEEDUP EM COMPUTAÇÃO PARALELA

O speedup é uma medida do grau de desempenho. O speedup mede a razão entre o tempo de execução sequencial e o tempo de execução em paralelo.

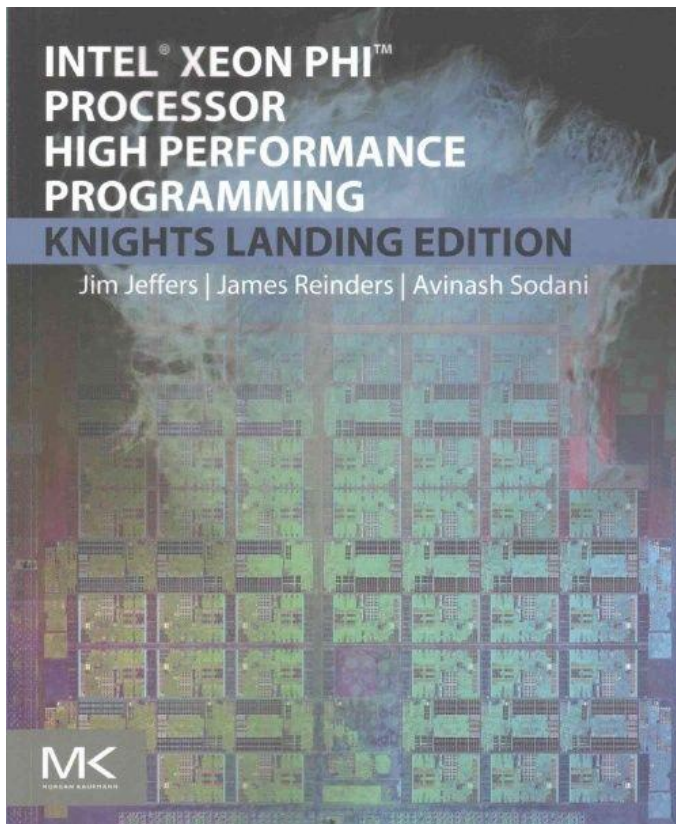
$$S(p) = \frac{T_p(1)}{T_p(p)}$$

$T_p(1)$ é o tempo de execução com um processador

$T_p(p)$ é o tempo de execução com p processadores

	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
T(p)	1000	520	280	160	100
S(p)	1,00	1,92	3,57	6,25	10,00
Ideal	1	2	4	8	16

BIBLIOGRAFIA XEON PHI



**Intel Xeon Phi Processor
High Performance
Programming**

Knights Landing Edition

Autores: Jim Jeffers, James
Reinders and Avinash Sodani

Ano: 2016

EXERCÍCIO 3

SELECTION SORT

3-selection-sort.c make ./3-selection-sort.exec 5

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

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

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

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

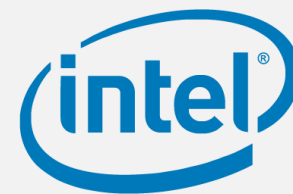
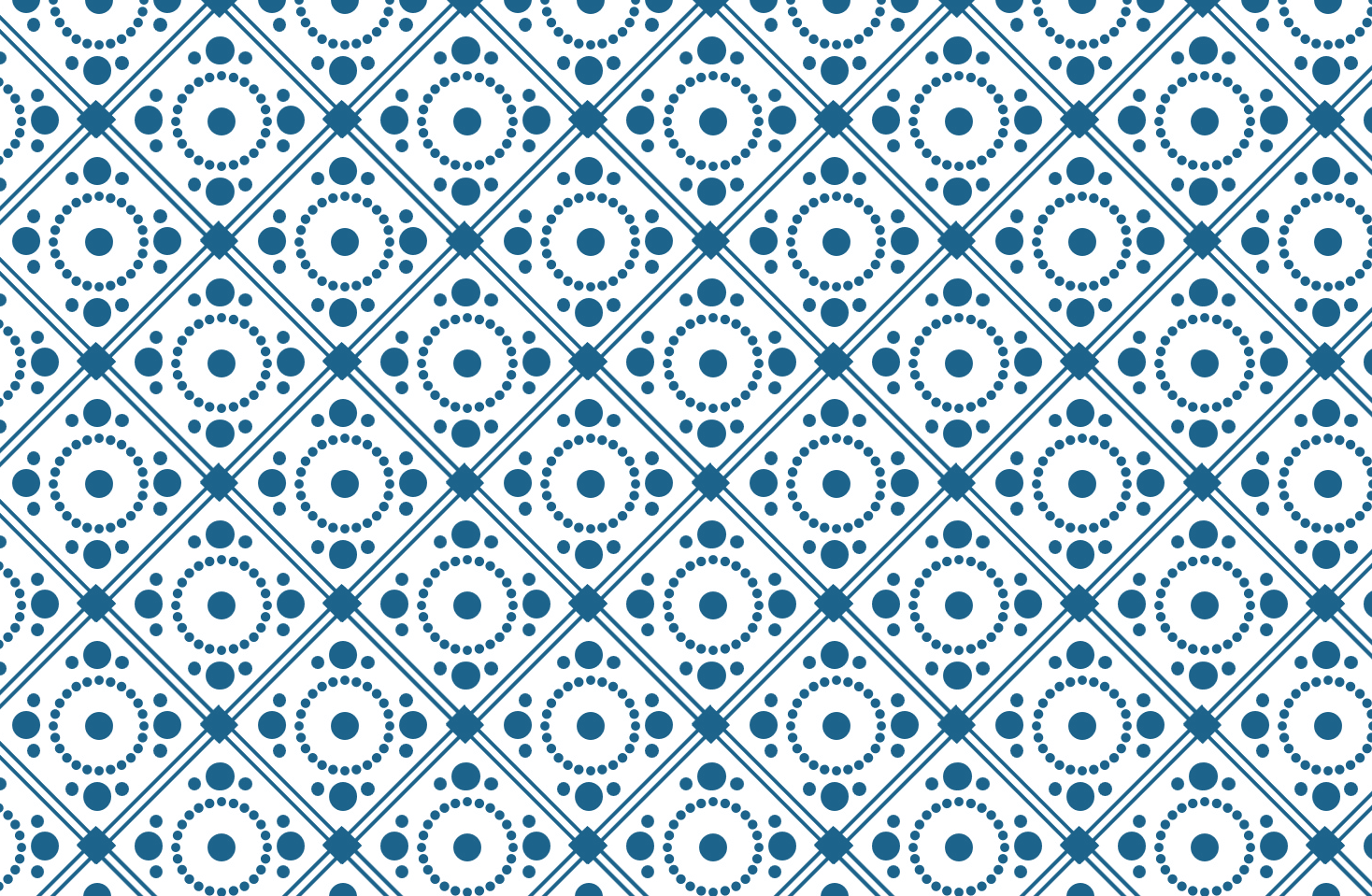

SOLUÇÃO 3.1

SELECTION SORT

3.1-selection-sort-parallel.c make ./3.1-selection-sort-parallel.exec 5

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

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



INTEL MODERN CODE PARTNER