Introdução à OpenMP (Dia 1)

Prof. Guido Araujo
www.ic.unicamp.br/~guido

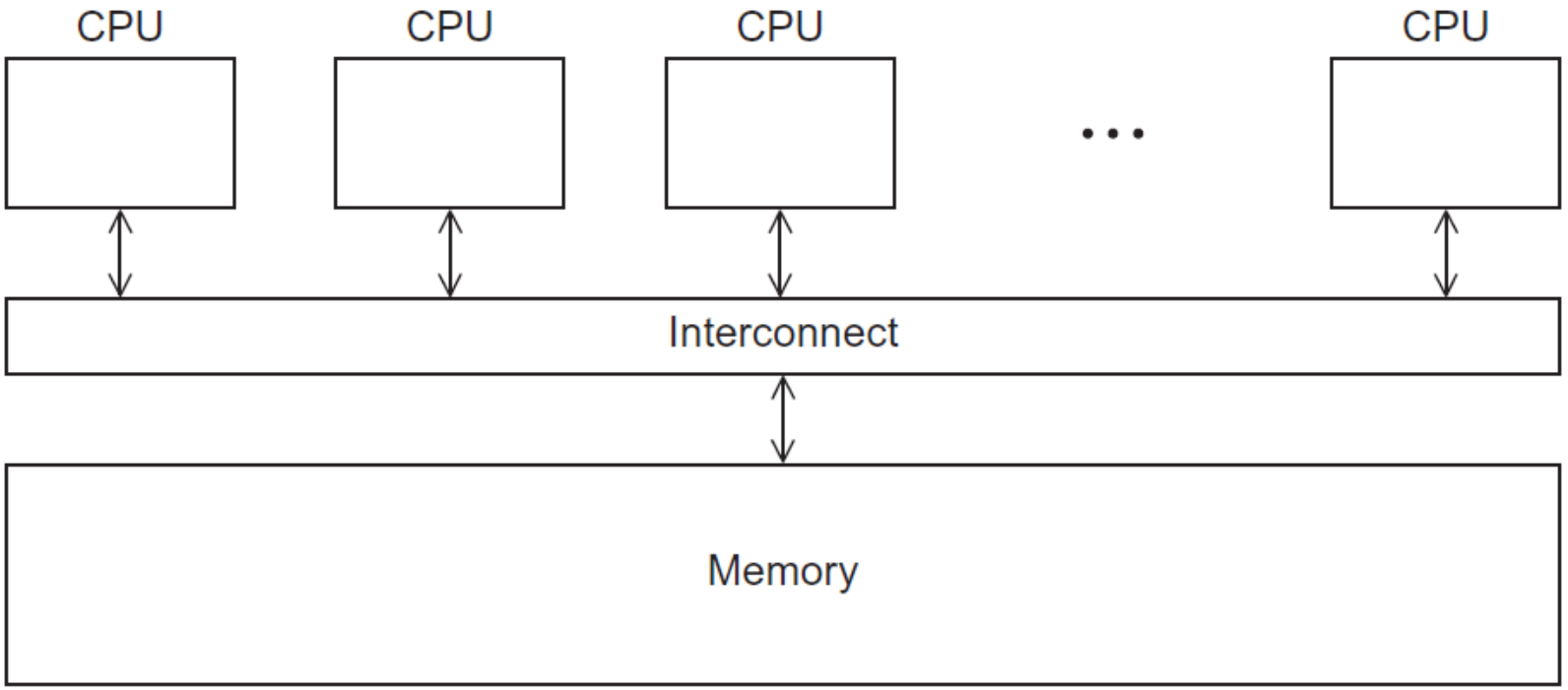
Roteiro

- Escrevendo programas usando OpenMP
- Usando OpenMP para paralelizar laços seriais com pequenas mudanças no código fonte
- Explorar paralelismo de tarefas
- Sincronização explícita de threads
- Problemas típicos em programação para memórias compartilhadas

OpenMP

- Uma API para programação paralela em memória compartilhada.
- MP = multiprocessing
- Projetada para sistemas no quais todas as threads ou processos podem, potencialmente, ter acesso à toda memória disponível.
- O sistema é visto como uma coleção de núcleos ou CPUs, no qual todos eles têm acesso à memória principal.

Um sistema de memória compatilhada



Pragmas

- Instruções especiais para pre-processamento.
- Tipicamente adicionadas ao sistema para permitir comportamentos que não são parte do especificação básica de C.
- Compiladores que não suportam pragmas ignoram-nos.

#pragma

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

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

```
gcc −g −Wall −fopenmp −o omp_hello omp_hello.c

./omp_hello 4
```

compilando

executando com 4 threads

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

resultados possíveis

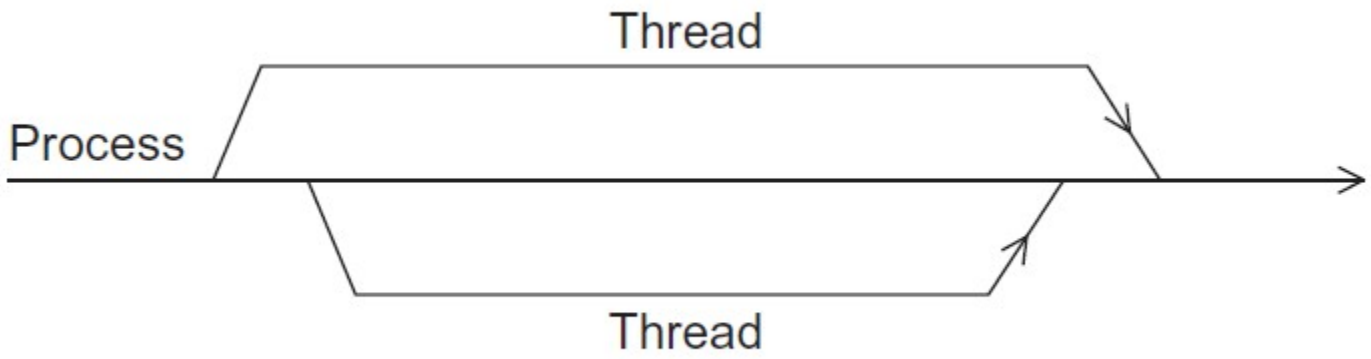
```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

OpenMP pragmas

- # pragma omp parallel
  - Diretiva paralela mais básica.
  - O número de threads que executam o bloco que segue o pragma é determinado pelo sistema de *runtime*.

Um processo de duas threads fazendo *fork* e *join*



Cláusula

- Texto que modifica uma diretiva.
- A cláusula *num_threads* pode ser adicionada a uma diretiva paralela.
- Permite o programador especificar o número de threads que devem executar no bloco que segue o pragma.

```
# pragma omp parallel num_threads ( thread_count )
```

Notas...

- Alguns sistemas podem limitar o número de threads que podem ser executadas.
- O padrão OpenMP não garante que serão iniciadas *thread_count* threads.
- A maioria dos sistemas podem iniciar centenas, ou até mesmo, milhares de threads.
- A não ser que desejemos iniciar um número muito grande de threads, quase sempre conseguiremos o número de threads desejado.

Terminologia

- Em OpenMP, o conjunto de threads formado pela thread original e pelas novas threads é chamado de team.
- A thread original é chamda de master, e as threads adicionais são chamadas slaves.

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

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

Medindo tempos

função teórica

```
double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

MPI_Wtime

omp_get_wtime

Speedup

- Número de núcleos = p
- Tempo de execução serial = T_{serial}
- Tempo de execução paralelo = $T_{parallel}$

$$T_{parallel} = T_{serial} / p$$

speedup linear

$$S = \frac{T_{serial}}{T_{parallel}}$$

Lei de Amdahl

- Ao não ser que todo um programa serial possa ser paralelizado, o speedup possível será bem limitado – independente do número de núcleos disponíveis.

Exemplo

- Podemos paralelizar 90% de um programa serial.
- A paralelização é “perfeita” independente do número *p* de núcleos que usarmos.
- Assuma um programa em que $T_{serial} = 20$ seconds

Exemplo (cont.)

- Tempo de execução da parte não “paralelizável” é

$$0.1 \times T_{serial} = 2$$

- Tempo de execução da parte paralelizável é:

$$0.9 \times T_{serial} / p = 18 / p$$

- Tempo de execução total é:

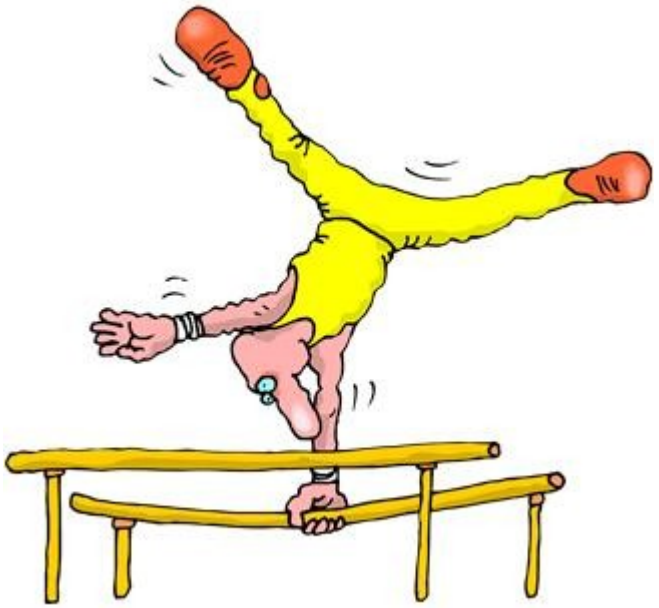
$$T_{parallel} = 0.9 \times T_{serial} / p + 0.1 \times T_{serial} = 18 / p + 2$$

Exemplo (cont.)

- Speed up

$$S = \frac{T_{serial}}{0.9 \times T_{serial} / p + 0.1 \times T_{serial}} = \frac{20}{18 / p + 2}$$

$$S = \frac{1}{f / p + (1 - f)}, \text{ onde } f = \frac{T_{parallelizable}}{T_{serial}}$$



DIRETIVA “PARALLEL FOR”

Parallel for

- Dispara um time de threads para executar o bloco lógico que segue.
- O bloco lógico que segue a diretiva precisa ser um laço for.
- A diretiva aloca cada iteração do laço que a segue a uma thread.

Tipos de sentenças for paralelizáveis

```
for ( index = start ; index < end  ; index++
                      index <= end    ++index
                      index >= end    index--
                      index > end     --index
                                      index += incr
                                      index -= incr
                                      index = index + incr
                                      index = incr + index
                                      index = index - incr )
```

Cuidado...

- A variável index precisa ser to tipo inteiro ou apontador (e.x., não pode ser float).

- As expressões start, end, and incr precisam ter tipos compatíveis. Por exemplo, se index é um apontador, então incr precisa ser do tipo inteiro.

Cuidado....

- As expressões start, end, and incr não podem mudar durante a execução do laço.

- Durante a execução do laço, a variável index somente pode ser modificada pela "expressão de incrementar" dentro da sentença for.

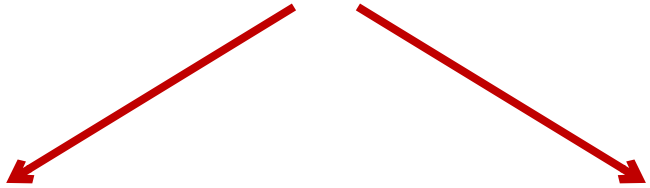
Tarefa 1

- Paralelize o programa serial que calcula a soma de dois vetores
- Meça o tempo serial
- Meça o tempo paralelo
- Meça o speed-up

Dependencia de dados

```
fibo[ 0 ] = fibo[ 1 ] = 1;
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

```
fibo[ 0 ] = fibo[ 1 ] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```



1 1 2 3 5 8

this is correct

1 1 2 3 2 2

but sometimes we get this (wrong)

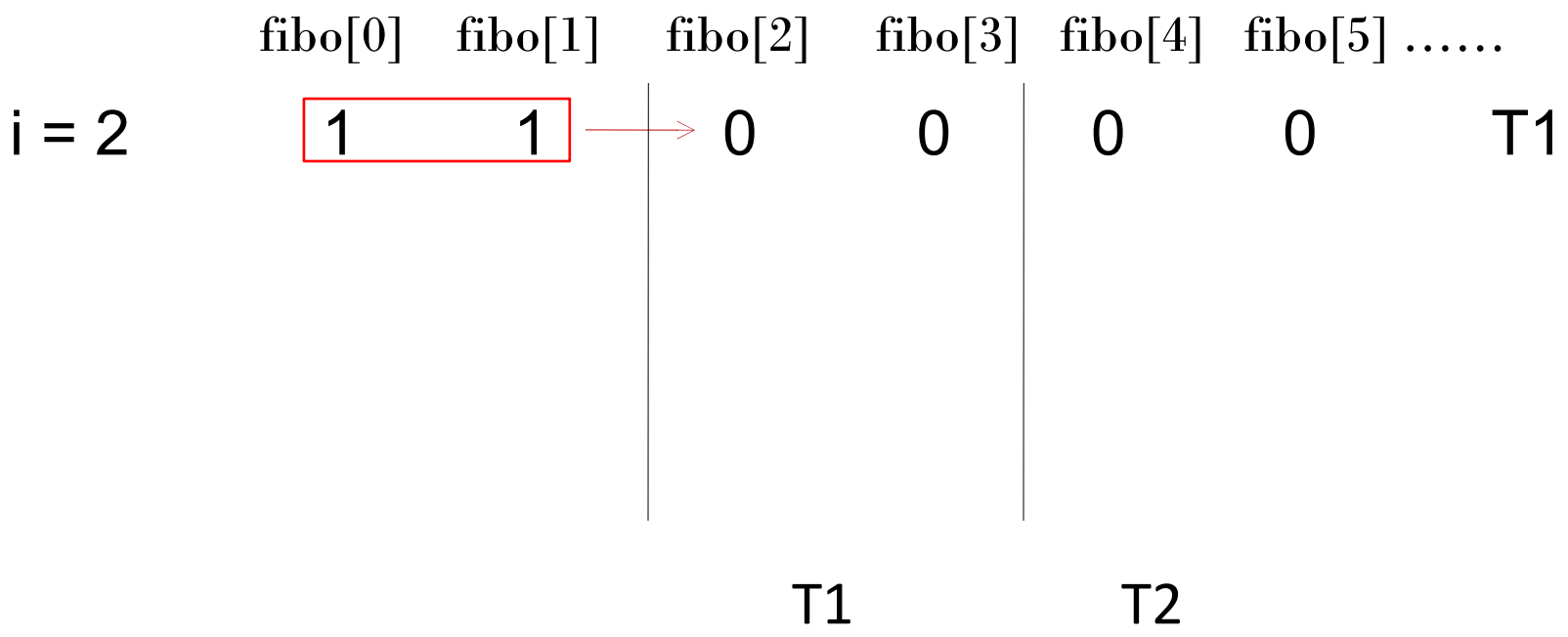
Dependencia de dados

- Assuma:
  - Duas threads (T1 e T2)
  - n = 6, ou seja cada thread faz duas iterações
  - T1 (i = 2,3) e T2 (i = 4, 5)

```
   fibo[ 0 ] = fibo[ 1 ] = 1;
#  pragma omp parallel for num_threads(2)
   for (i = 2; i < n; i++)
      fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

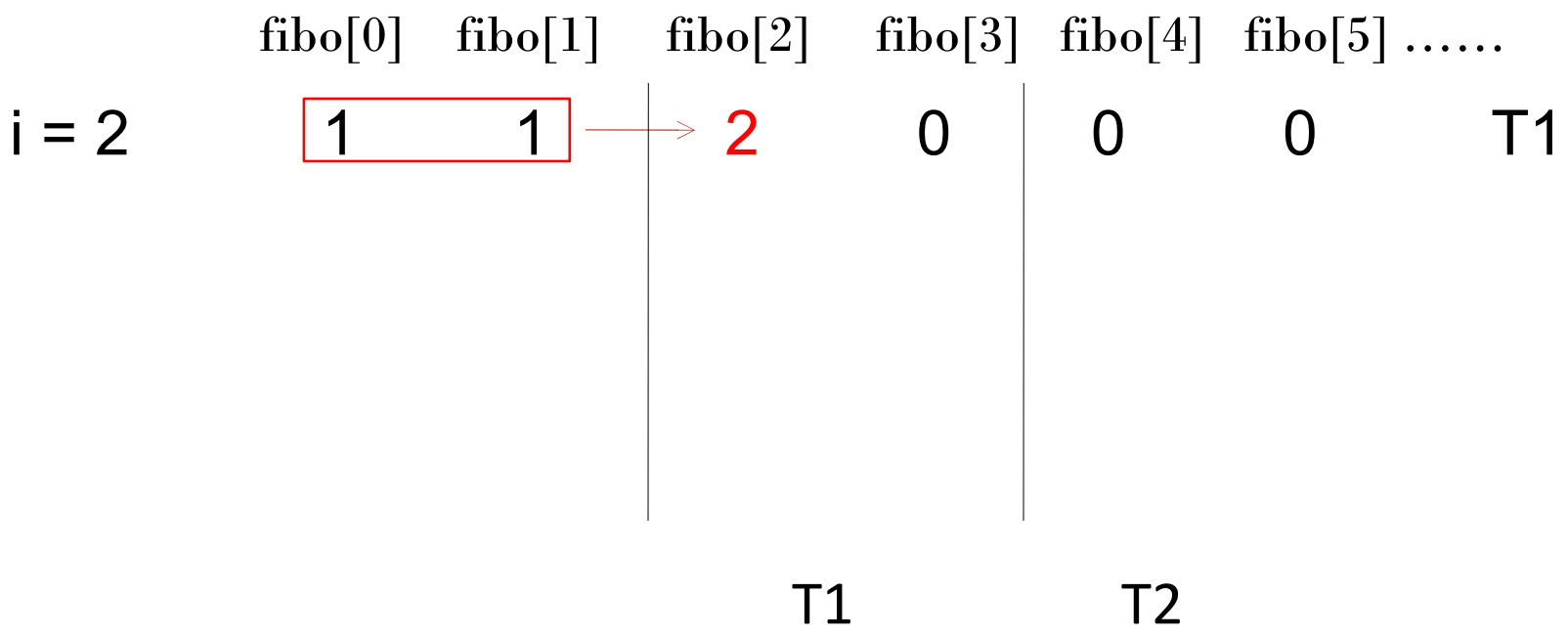
O que ocorreu?

```
fibo[ 0 ] = fibo[ 1 ] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```



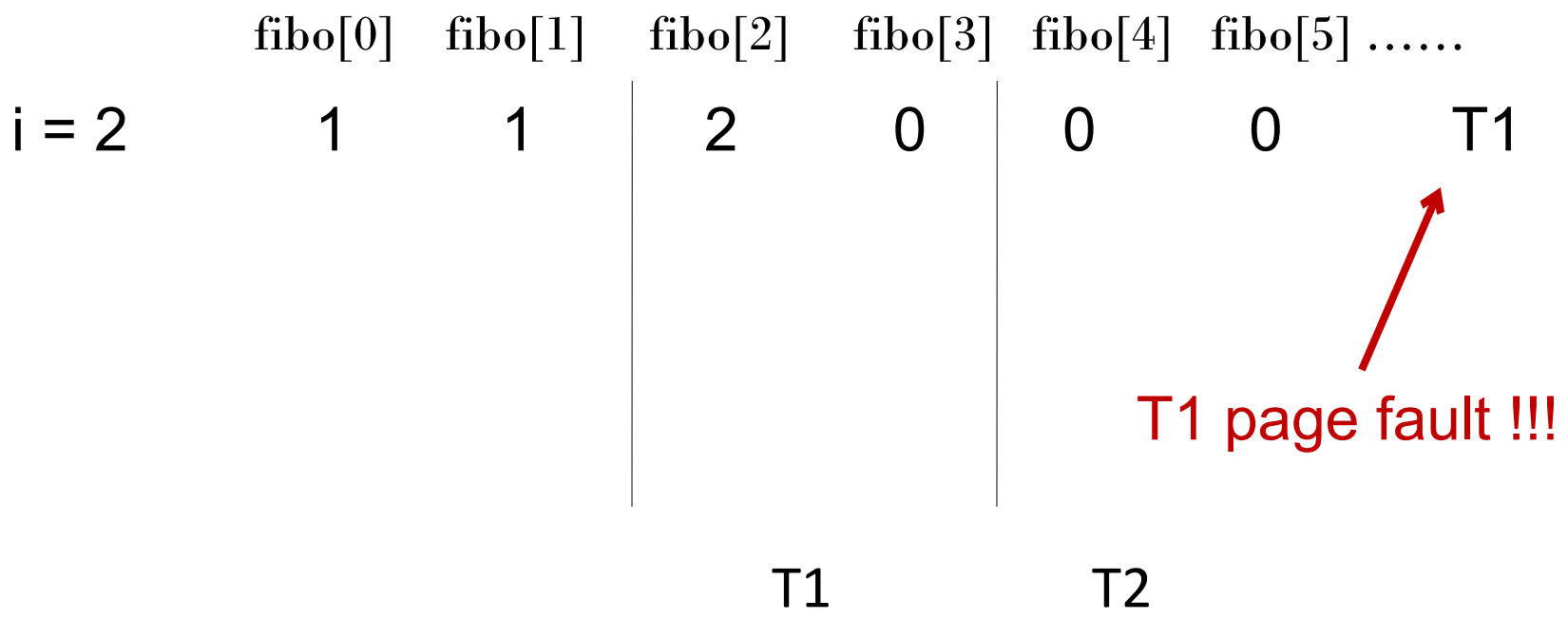
O que ocorreu?

```
fibo[ 0 ] = fibo[ 1 ] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```



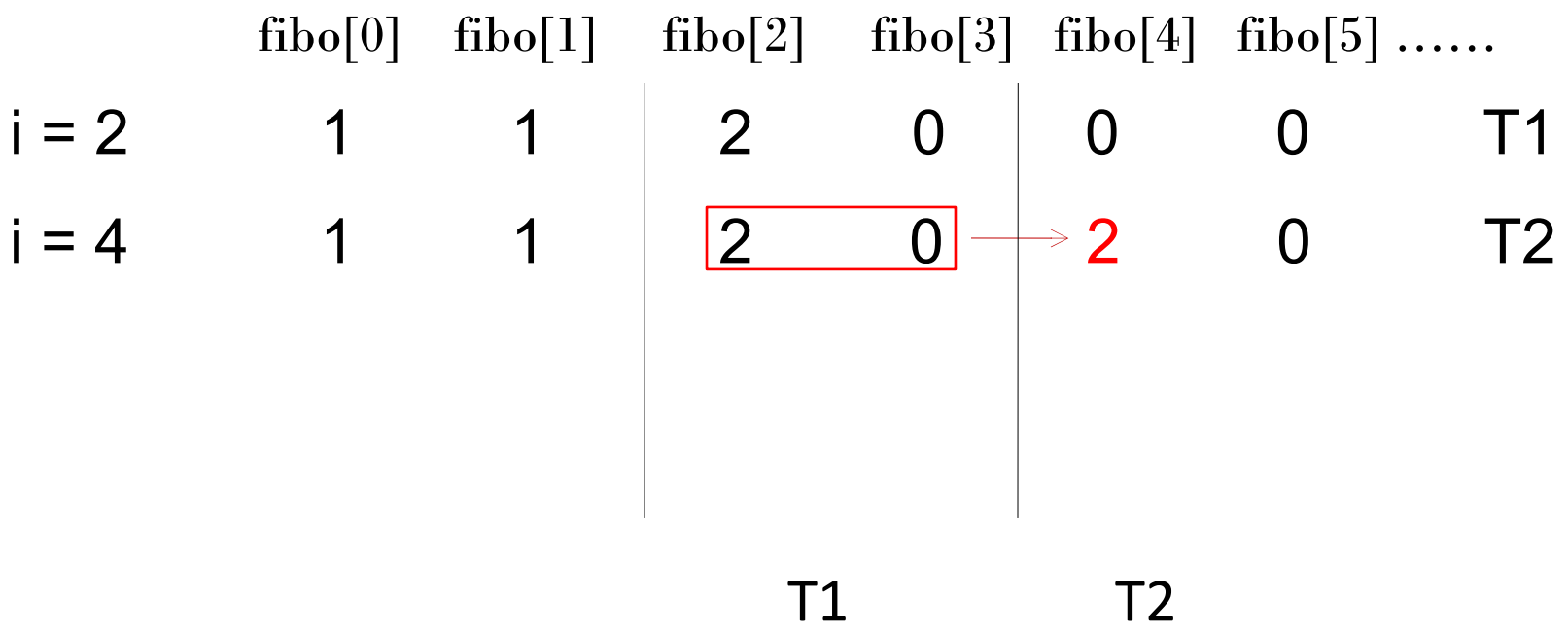
O que ocorreu?

```
fibo[ 0 ] = fibo[ 1 ] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```



O que ocorreu?

```
fibo[ 0 ] = fibo[ 1 ] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```



O que ocorreu?

```
   fibo[ 0 ] = fibo[ 1 ] = 1;
#  pragma omp parallel for num_threads(2)
   for (i = 2; i < n; i++)
      fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

	fibo[0]	fibo[1]	fibo[2]	fibo[3]	fibo[4]	fibo[5]	
i = 2	1	1	2	0	0	0	T1
i = 4	1	1	2	0	2	2	T2
i = 5	1	1	2	0	2 →	2	T2

T1 T2

O que ocorreu?

```
   fibo[ 0 ] = fibo[ 1 ] = 1;
#  pragma omp parallel for num_threads(2)
   for (i = 2; i < n; i++)
      fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

	fibo[0]	fibo[1]	fibo[2]	fibo[3]	fibo[4]	fibo[5]	
i = 2	1	1	2	0	0	0	T1
i = 4	1	1	2	0	2	2	T2
i = 5	1	1	2	0	2 →	2	T2
							T1

T1 retornou !!!

O que ocorreu?

```
   fibo[ 0 ] = fibo[ 1 ] = 1;
#  pragma omp parallel for num_threads(2)
   for (i = 2; i < n; i++)
      fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

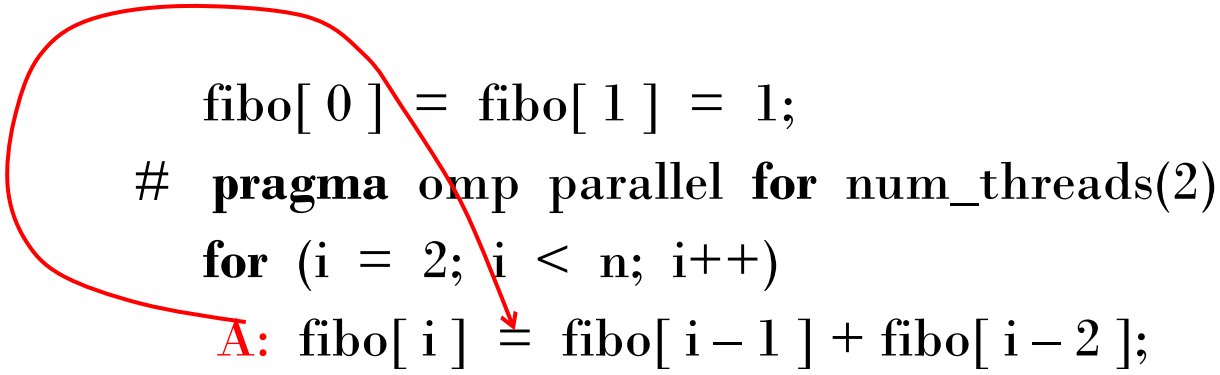
	fibo[0]	fibo[1]	fibo[2]	fibo[3]	fibo[4]	fibo[5]	
i = 2	1	1	2	0	0	0	T1
i = 4	1	1	2	0	2	2	T2
i = 5	1	1	2	0	2	0	T2
i = 3	1	1	2	→ 3	2	2	T1

T1 T2

O que ocorreu?

1. Compiladores OpenMP não checam dependências entre iterações do laço que está sendo paralelizado com a diretiva *parallel for*.
2. Um laço cujos resultados de uma ou mais iterações dependem de outras iterações não pode, no geral, ser corretamente paralelizado por OpenMP.

Como detectar?



```
    fibo[ 0 ] = fibo[ 1 ] = 1;
#   pragma omp parallel for num_threads(2)
    for (i = 2; i < n; i++)
A:    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

- Se o grafo não possuir dependências *loop-carried* o laço é DOALL e as iterações podem ser separadas em threads
- Do contrário é DOACROSS, e você não pode paralelizá-lo

Tarefa 2

- Faça um programa serial que gere e imprima a sequência de Fibonacci

- Paralelize o programa, execute várias vezes e veja se a saída permanece a mesma.

Estimando π

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
    sum += factor/(2*k+1);
    factor = -factor;
}
pi_approx = 4.0*sum;
```

Qual o problema aqui?

Problemas de Corrida

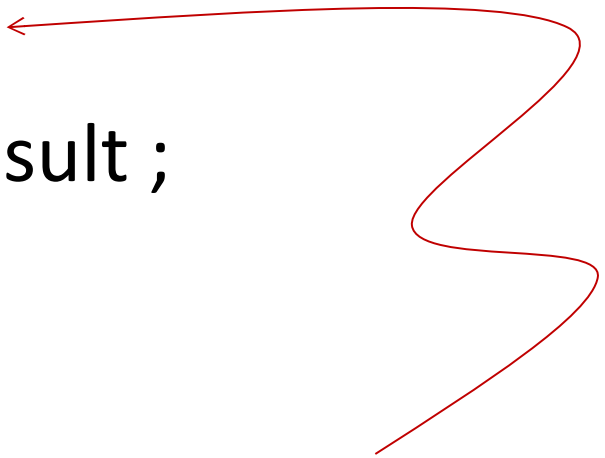
Time	Thread 0	Thread 1
0	`global_result = 0` to register	finish `my_result`
1	`my_result = 1` to register	`global_result = 0` to register
2	add `my_result` to `global_result`	`my_result = 2` to register
3	store `global_result = 1`	add `my_result` to `global_result`
4		store `global_result = 2`

Resultados imprevisíveis podem ocorrer quando duas (ou mais) threads tentam simultaneamente executar:

global_result += my_result ;

Exclusão mútua

```
# pragma omp critical
   global_result += my_result ;
```



somente uma thread pode executar o bloco estruturado por vez

Estimando π

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
    sum += factor/(2*k+1);
    factor = -factor;
}
pi_approx = 4.0*sum;
```

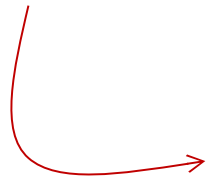
pragma omp critical

Reduction operators

- Um operador de reduction é um operador binário (tal como adição e multiplicação).
- Uma reduction é uma computação que repetidamente aplica o mesmo operador de redução a uma sequência de operandos visando obter um único resultado.
- Todos os resultados intermediários da operação devem ser armazenadas na mesma variável: a variável de redução.

Uma cláusula de redução pode ser adicionada a uma diretiva paralela.

```
reduction(<operator>: <variable list>)
```



+, *, &, |, ^, &&, ||

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

Solução OpenMP #1

```
        double factor = 1.0;
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (k = 0; k < n; k++) {
           sum += factor/(2*k+1);
           factor = -factor;
        }
        pi_approx = 4.0*sum;
```

Tarefa 3

- Paralelize um program serial que calcula o produto interno de dois vetores usando reduction
- Meça o tempo serial
- Meça o tempo paralelo
- Meça o speed-up

Solução OpenMP #1

```
        double factor = 1.0;
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (k = 0; k < n; k++) {
           sum += factor/(2*k+1);
           factor = -factor;
        }
        pi_approx = 4.0*sum;
```

Qual o problema aqui?

Solução OpenMP #1

Loop-carried dependency

```
        double factor = 1.0;
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (k = 0; k < n; k++) {
           sum += factor/(2*k+1);
           factor = -factor;
        }
        pi_approx = 4.0*sum;
```

Como resolver isto?

Solução OpenMP #2

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

Qual o problema aqui?

Solução OpenMP #2

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            reduction(+:sum)
        for (k = 0; k < n; k++) {
            if (k % 2 == 0)
               factor = 1.0;
            else
               factor = -1.0;
            sum += factor/(2*k+1);
        }
```

Qual a solução?

T0 (i=0)

:

factor = 1

:

:

:

sum += factor / (2*k + 1);

T1 (i=17)

:

factor = -1

Escopo

- Em linguagens de programação, o escopo de uma variável são aquelas partes do programa nas quais as variáveis podem ser usadas.

- Em OpenMP, o escopo de uma variável se refere ao conjunto de threads que podem acessar a variável em um bloco paralelo.

Escopo em OpenMP

- Uma variável que pode ser acessada por todas as threads de um *team* possui um escopo shared.

- Uma variável que é acessada por apenas uma thread tem escopo private.

- O escopo das variáveis declaradas antes de um bloco paralelo é shared.

Solução OpenMP #2

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) private(factor)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

garante que factor tem escopo privado.

A cláusula *default* (1)

- Deixa o programador definir o escopo de cada variável em um bloco.

```
default(none)
```

- Com esta cláusula o compilador vai requerer que definamos o escopo de cada variável usada em um bloco e que foi declarada for a do bloco.

A cláusula *default* (2)

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           default(none) reduction(+:sum) private(k, factor) \
           shared(n)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```