

MO644/MC970

Programação em Memória Compartilhada usando OpenMP

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

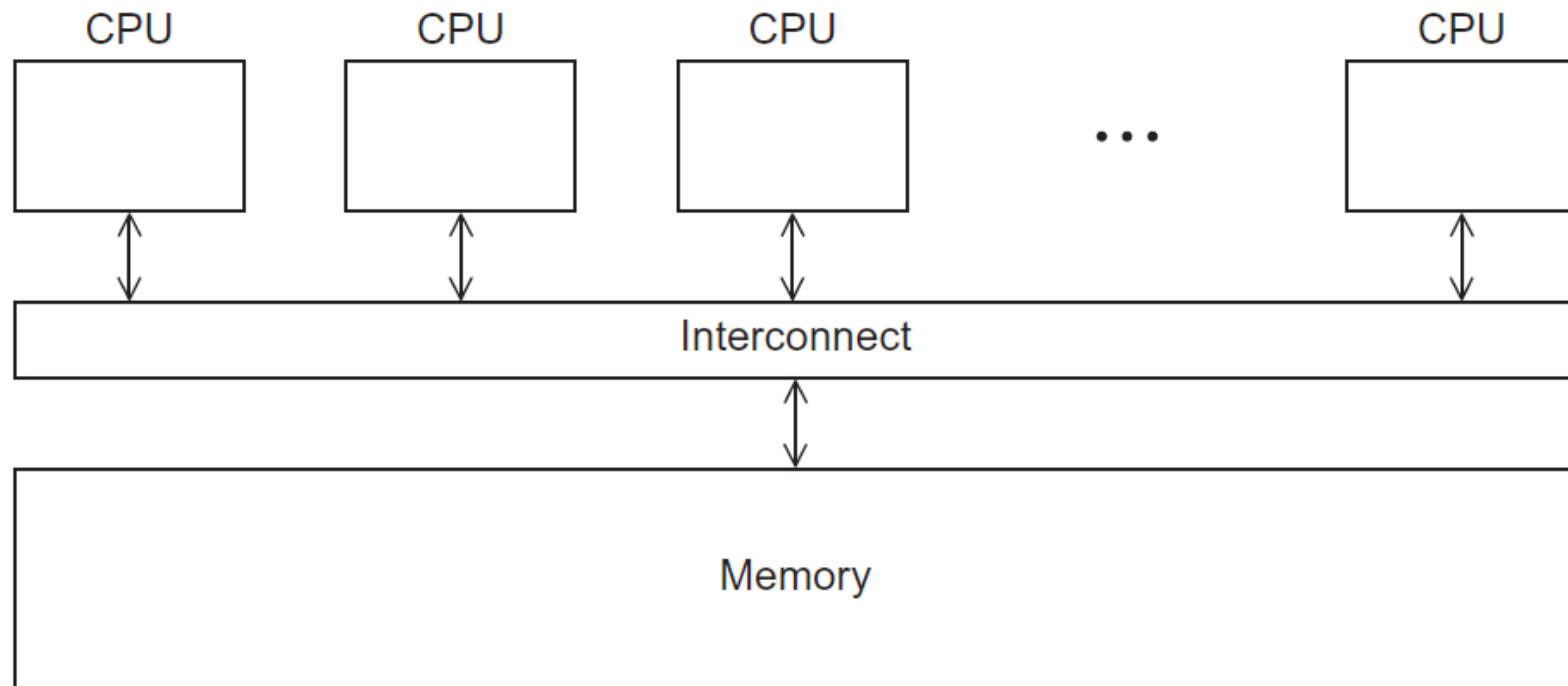
Roteiro

- 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 nos 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 compartilhada



Pragmas

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

#pragma

```

#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

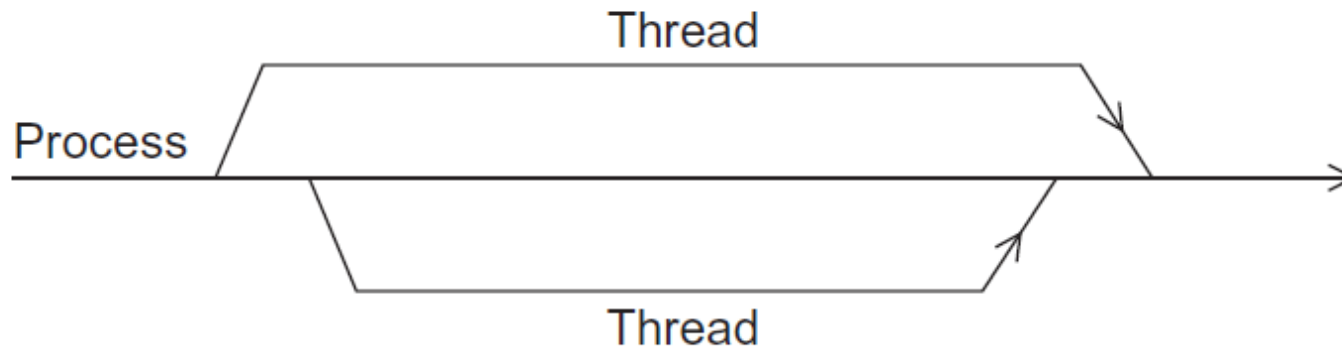
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

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 chamada de **master**, e as threads adicionais são chamadas **slaves**.



E se o compilador não aceitar OpenMP?

```
#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();

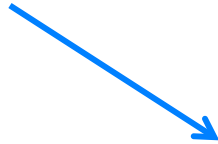
    return 0;
} /* main */

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 */
```

Caso o compilador não aceite OpenMP

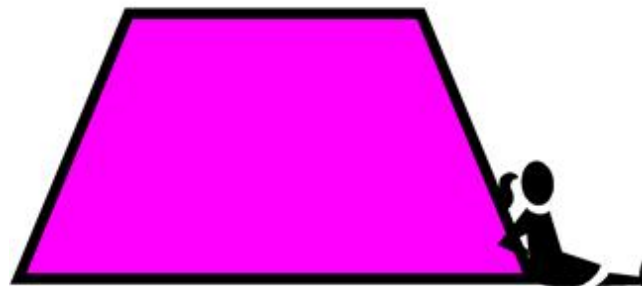
```
# include <omp.h>
```



```
#ifdef _OPENMP  
# include <omp.h>  
#endif
```

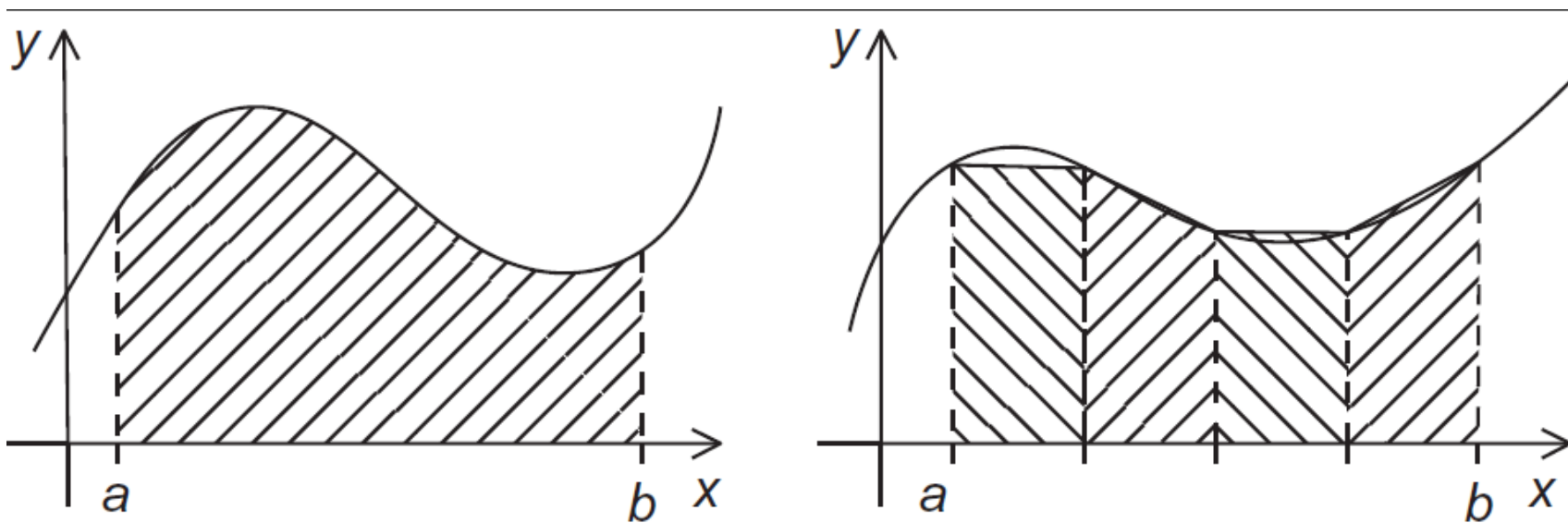
Caso o compilador não aceite OpenMP

```
# ifdef _OPENMP
    int my_rank = omp_get_thread_num( );
    int thread_count = omp_get_num_threads( );
# else
    int my_rank = 0;
    int thread_count = 1;
# endif
```



A REGRA DO TRAPÉZIO

A regra do trapézio



Algoritmo serial

```
/* Input:  a, b, n */  
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
for (i = 1; i <= n-1; i++) {  
    x_i = a + i*h;  
    approx += f(x_i);  
}  
approx = h*approx;
```

Uma primeira versão em OpenMP

- 1) Identificamos dois tipos de tarefas:
 - a) Computação das áreas dos trapézios individuais
 - b) Soma das áreas dos trapézios.
- 2) Não existe comunicação entre as tarefas do item 1a, mas cada uma delas se comunica com as tarefas do item 1b.

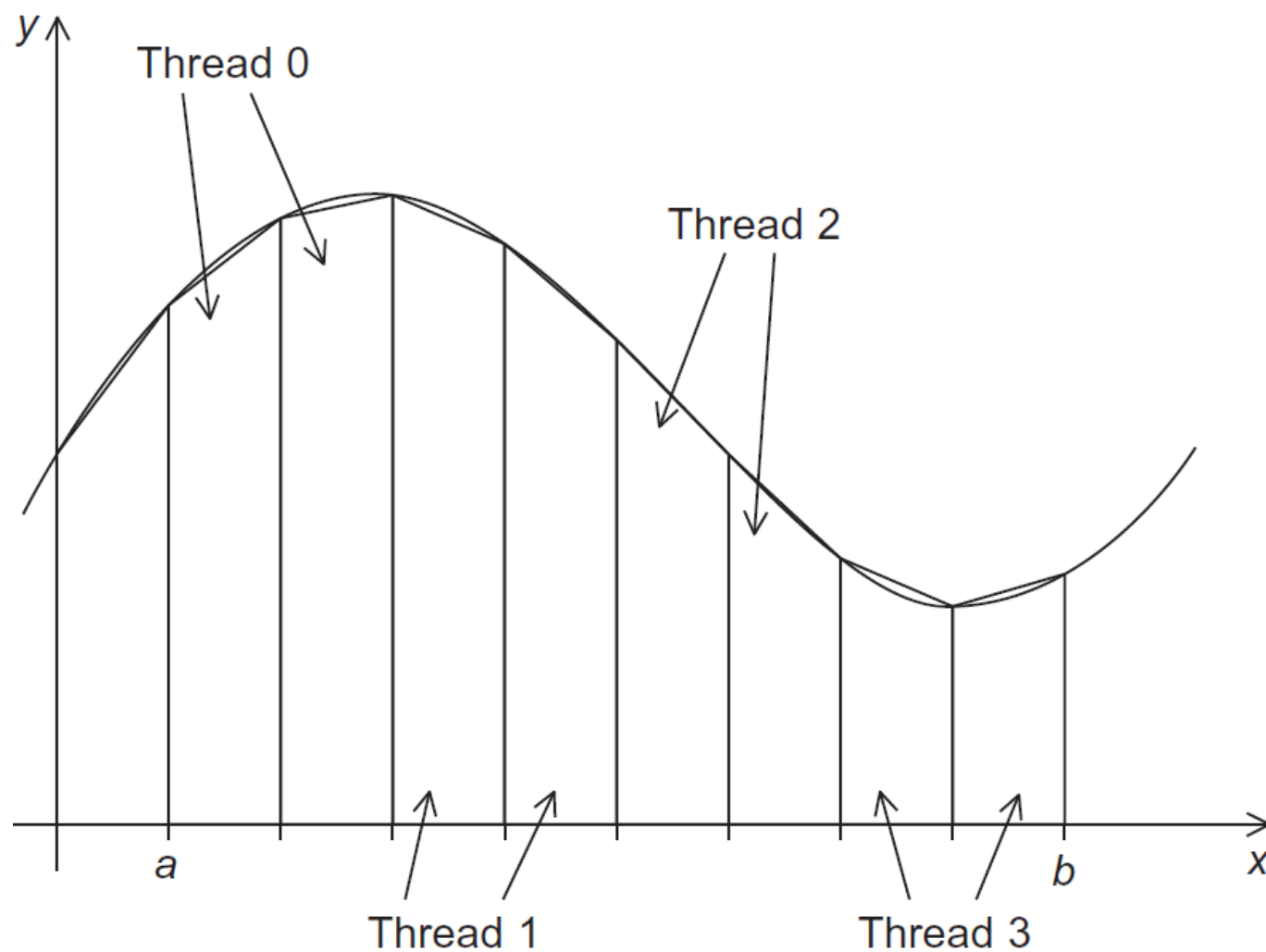


Qual o problema com isto?

Uma primeira versão em OpenMP (2)

- 3) Nós assumimos que existem muito mais trapézios que núcleos.
- 4) Agregamos tarefas atribuindo blocos de trapézios consecutivos a cada thread (e uma única thread a cada núcleo).

Atribuindo trapézios à threads



Time	Thread 0	Thread 1
0	<code>global_result = 0 to register</code>	<code>finish my_result</code>
1	<code>my_result = 1 to register</code>	<code>global_result = 0 to register</code>
2	<code>add my_result to global_result</code>	<code>my_result = 2 to register</code>
3	<code>store global_result = 1</code>	<code>add my_result to global_result</code>
4		<code>store global_result = 2</code>

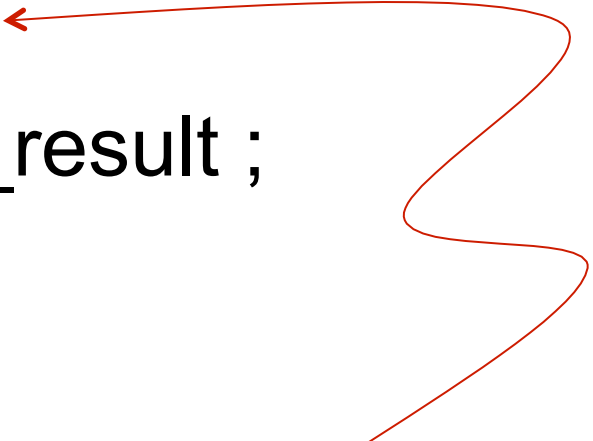
Resultados imprevisíveis podem ocorrer quando duas (ou mais) threads tentam simultaneamente executar:

`global_result += my_result ;`



Exclusão mútua

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



somente uma thread pode executar o
bloco estruturado por vez

```

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

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
    double    global_result = 0.0;  /* Store result in global_result */
    double    a, b;                 /* Left and right endpoints      */
    int       n;                     /* Total number of trapezoids    */
    int       thread_count;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);
    # pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n",
        a, b, global_result);
    return 0;
} /* main */

```



```

void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    h = (b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n-1; i++) {
        x = local_a + i*h;
        my_result += f(x);
    }
    my_result = my_result*h;

    # pragma omp critical
        *global_result_p += my_result;
}.../*..Trap. */

```



SCOPE OF VARIABLES

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**.





A CLÁUSULA REDUCTION

Precisamos desta versão mais complexa para poder somar o resultado parcial de cada thread em *global_result*.

```
void Trap(double a, double b, int n, double* global_result_p);
```

O ideal seria usar esta chamada....

```
double Trap(double a, double b, int n);
```



```
global_result = Trap(a, b, n);
```

Se usarmos esta não existe região crítica na chamada!

```
double Local_trap(double a, double b, int n);
```

Se fixarmos desta forma...

```
global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
{
#   pragma omp critical
    global_result += Local_trap(double a, double b, int n);
}
```

← qual o problema aqui?

... forçamos as threads a executar sequencialmente.

... como evitar isto?

Podemos evitar este problema declarando uma variável privada dentro do bloco paralelo e movendo a seção crítica para depois da chamada da função.

```
global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
{
    double my_result = 0.0;  /* private */

    my_result += Local_trap(double a, double b, int n);
# pragma omp critical
    global_result += my_result;
}
```




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.

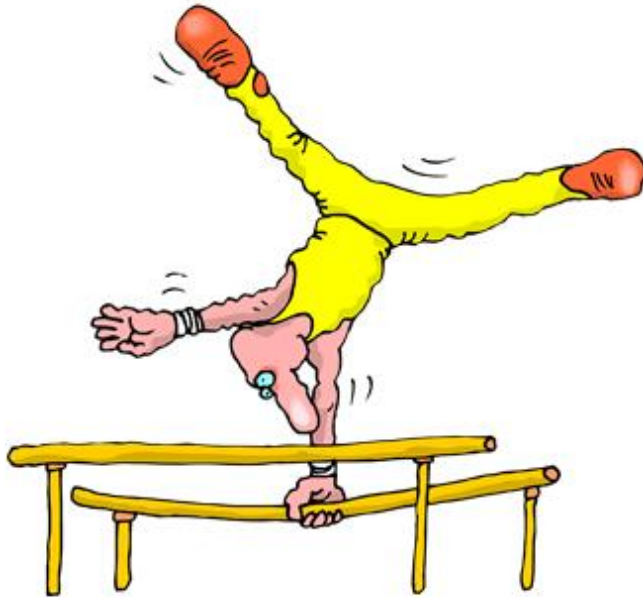
Uma cláusula de redução pode ser adicionada a uma diretiva paralela.

```
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);
```

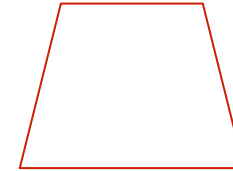


DIRETIVA “PARALLEL FOR”

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.

```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
for (i = 1; i <= n-1; i++)  
    approx += f(a + i*h);  
approx = h*approx;
```



qual a vantagem aqui?

```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
# pragma omp parallel for num_threads(thread_count) \  
    reduction(+: approx)  
for (i = 1; i <= n-1; i++)  
    approx += f(a + i*h);  
approx = h*approx;
```

Tipos de sentenças for paralelizáveis

for	{	index = start ;		index++
				++index
			index < end	index--
			index <= end	--index
			index >= end ;	index += incr
			index > end	index -= incr
				index = index + incr
				index = incr + index
				index = index - incr
				}

Cuidado...

- A variável `index` precisa ser do 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**.

Dependencia de datos

```
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



but sometimes
we get this

1 1 2 3 2 2

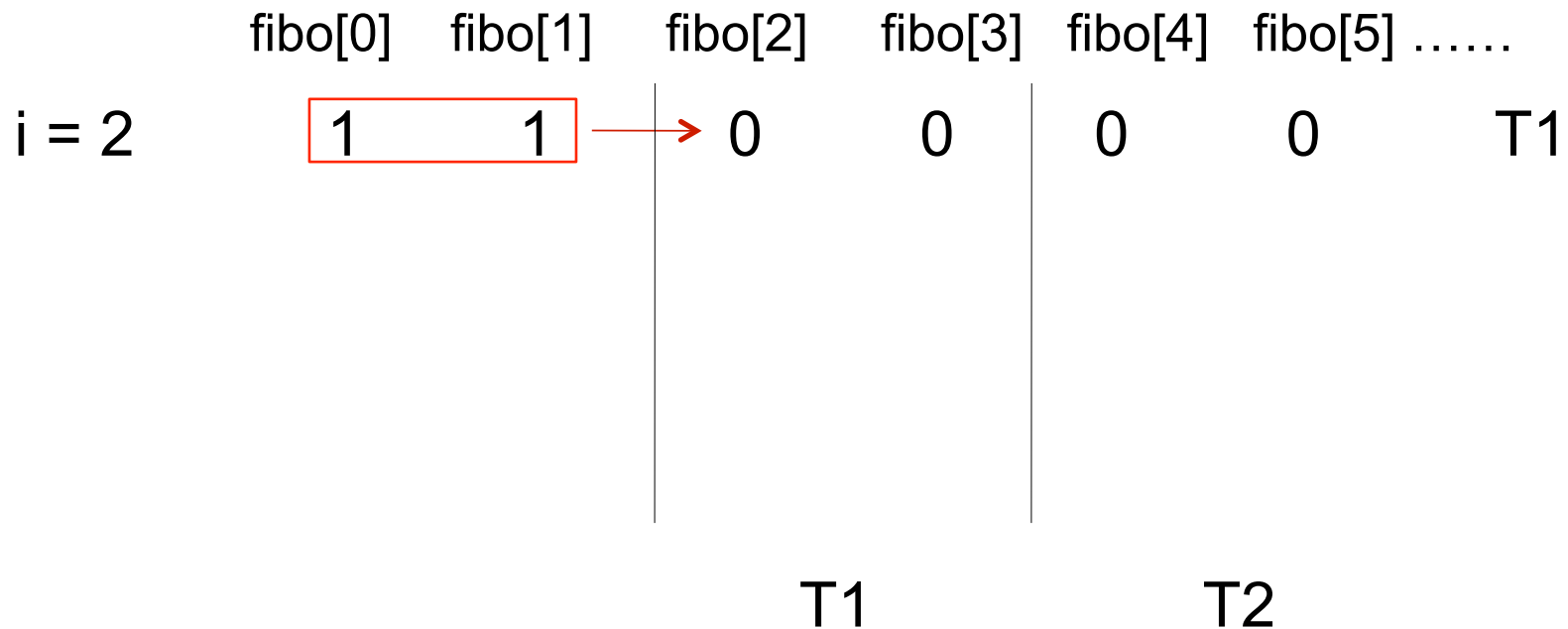
Dependencia de dados

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

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

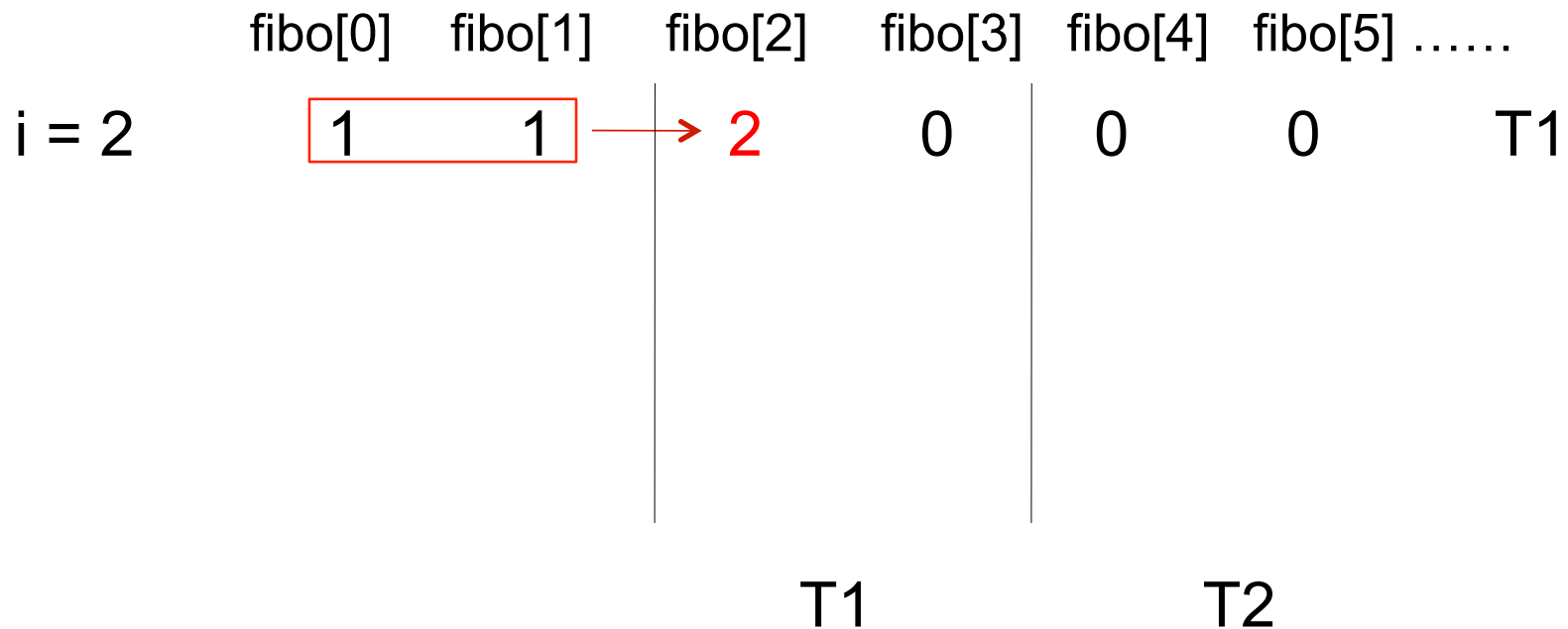
O que ocorreu?

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



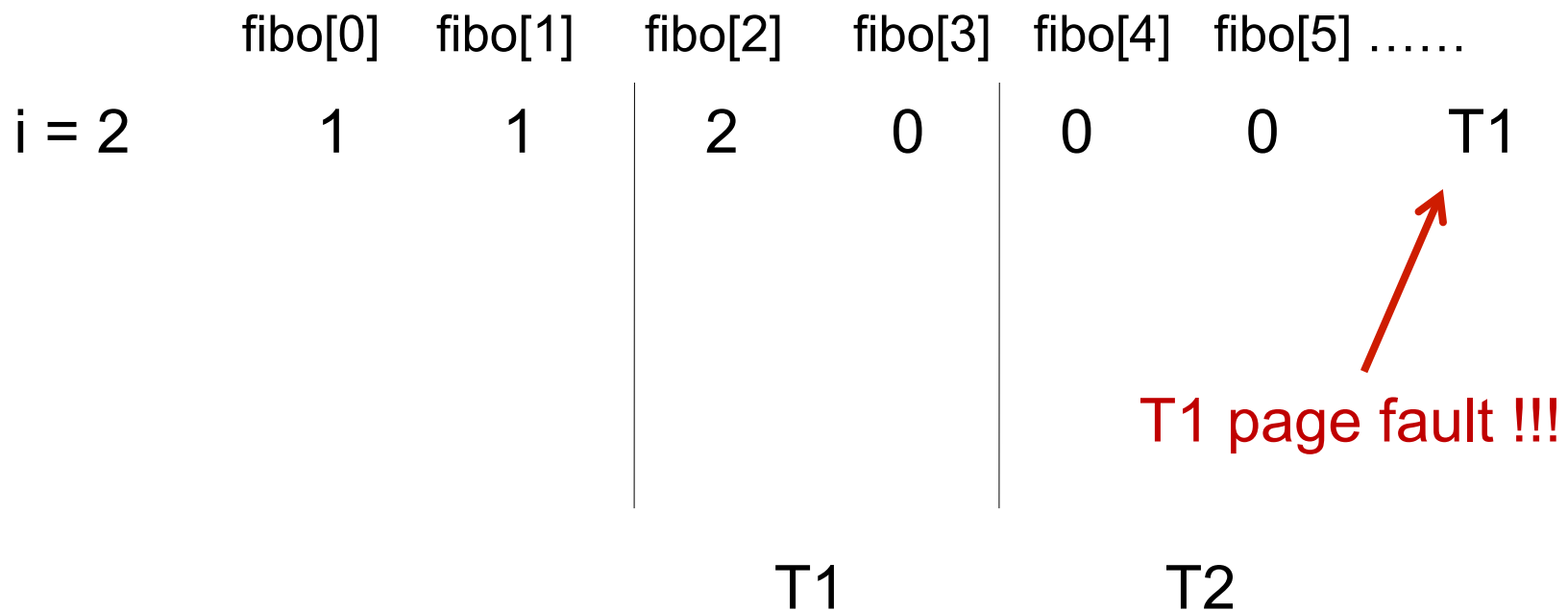
O que ocorreu?

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



O que ocorreu?

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



O que ocorreu?

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

	fibonacci[0]	fibonacci[1]	fibonacci[2]	fibonacci[3]	fibonacci[4]	fibonacci[5]
i = 2	1	1	2	0	0	0	T1
i = 3	1	1	2	0	→ 2	0	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 ];

```

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

T1 T2

O que ocorreu?

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

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

T1 retornou !!!



O que ocorreu?

```

fibonacci[0] = fibonacci[1] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < n; i++)
    fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];

```

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

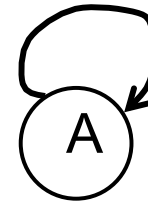
O que ocorreu?



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 ];
```



- Construa um grafo de dependencias para as instruções do laço
- Se o grafo não possuir ciclos o laço é DOALL e as iterações podem ser separadas em threads
- Do contrário é DOACROSS, e você está com um problema ;-)

Estimando π

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

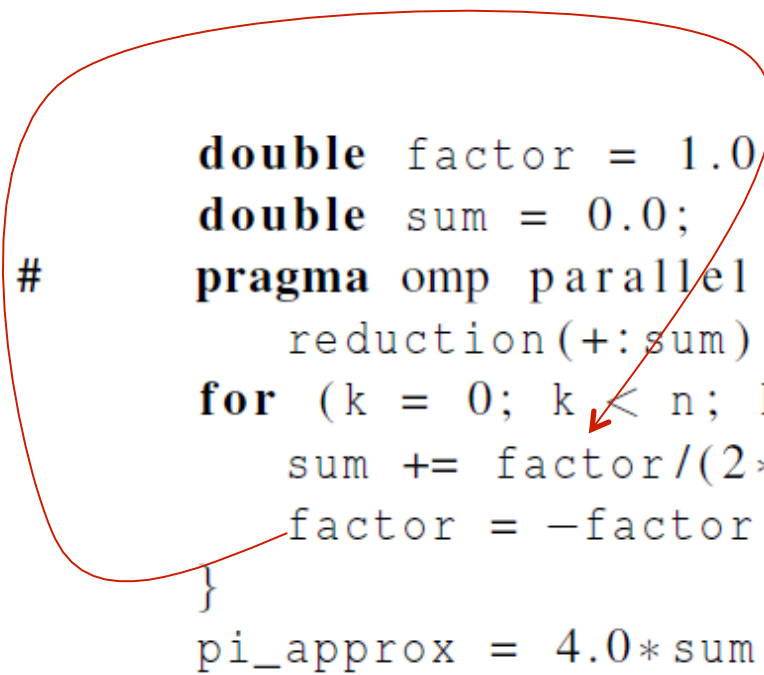
```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
    sum += factor/(2*k+1);
    factor = -factor;
}
pi_approx = 4.0*sum;
```

Qual o problema aqui?

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);
}
```

T2 (i=0)

:

factor = 1

:

:

:

:

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

T3 (i=1)


:

factor = -1

Qual a solução?

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 fora 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);
}
```



MAIS SOBRE LAÇOS EM OPENMP: ORDENAÇÃO

Bubble Sort

```
for (list_length = n; list_length >= 2; list_length--)  
    for (i = 0; i < list_length-1; i++)  
        if (a[i] > a[i+1]) {  
            tmp = a[i];  
            a[i] = a[i+1];  
            a[i+1] = tmp;  
        }
```

Outer loop

list_length = 4 : 3 4 1 2 > 3 1 2 **4**

list_length = 3 : 3 1 2 **4**

Como funciona?



Como paralelizar?

Odd-Even Transposition Sort (serial)

```
for (phase = 0; phase < n; phase++)  
    if (phase % 2 == 0)  
        for (i = 1; i < n; i += 2)  
            if (a[i-1] > a[i]) Swap(&a[i-1], &a[i]);  
    else  
        for (i = 1; i < n-1; i += 2)  
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

Como funciona?

Even: $a[i-1]$ $a[i]$ $a[i+1]$

Odd: $a[i-1]$ $a[i]$ $a[i+1]$

Odd-Even Transposition Sort (serial)

Phase	Subscript in Array			
	0	1	2	3
0	9	↔ 7	8	↔ 6
	7	9	6	8
1	7	9	↔ 6	8
	7	6	9	8
2	7	↔ 6	9	↔ 8
	6	7	8	9
3	6	7	↔ 8	9
	6	7	8	9

Outer loop

phase = 0 : 9 7 8 6 > (9,7) e (8,6)

phase = 1 : pode ser que compare (7,8)

OpenMP Odd-Even Sort # 1

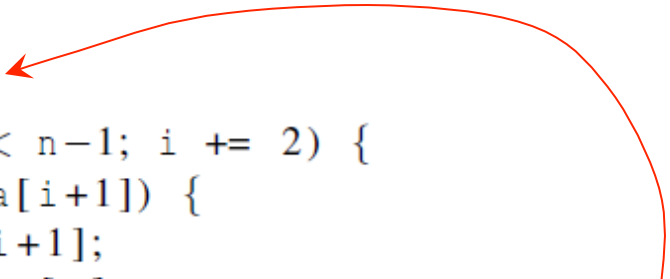
```
for (phase = 0; phase < n; phase++) {  
    if (phase % 2 == 0)  
#        pragma omp parallel for num_threads(thread_count) \  
            default(none) shared(a, n) private(i, tmp) \  
            for (i = 1; i < n; i += 2) {  
                if (a[i-1] > a[i]) {  
                    tmp = a[i-1];  
                    a[i-1] = a[i];  
                    a[i] = tmp;  
                }  
            }  
    else  
#        pragma omp parallel for num_threads(thread_count) \  
            default(none) shared(a, n) private(i, tmp) \  
            for (i = 1; i < n-1; i += 2) {  
                if (a[i] > a[i+1]) {  
                    tmp = a[i+1];  
                    a[i+1] = a[i];  
                    a[i] = tmp;  
                }  
            }  
}
```

fase inteira precisa
terminar junto

Problema: dispara novas threads
a cada iteração do laço externo

OpenMP Odd-Even Sort # 2

```
# pragma omp parallel num_threads(thread_count) \  
    default(none) shared(a, n) private(i, tmp, phase)  
for (phase = 0; phase < n; phase++) {  
    if (phase % 2 == 0)  
#        pragma omp for  
        for (i = 1; i < n; i += 2) {  
            if (a[i-1] > a[i]) {  
                tmp = a[i-1];  
                a[i-1] = a[i];  
                a[i] = tmp;  
            }  
        }  
    else  
#        pragma omp for  
        for (i = 1; i < n-1; i += 2) {  
            if (a[i] > a[i+1]) {  
                tmp = a[i+1];  
                a[i+1] = a[i];  
                a[i] = tmp;  
            }  
        }  
}
```



Threads são disparadas uma vez e
reusadas a cada iteração
laço externo

Odd-even sort com
duas diretivas *parallel for* e duas *for*
(Tempos em segundos para 20.000 elementos)

thread_count	1	2	3	4
Two parallel for directives	0.770	0.453	0.358	0.305
Two for directives	0.732	0.376	0.294	0.239





ESCALONAMENTO DE LAÇOS

```
sum = 0.0;  
for (i = 0; i <= n; i++)  
    sum += f(i);
```

Queremos paralelizar este laço

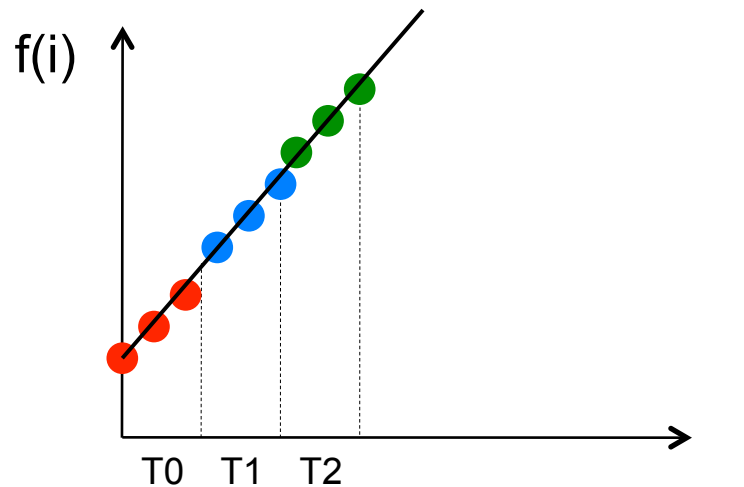
Assuma $f(i)$ leva dobro de tempo a cada iteração
 $f(2*i) \approx 2*f(i)$

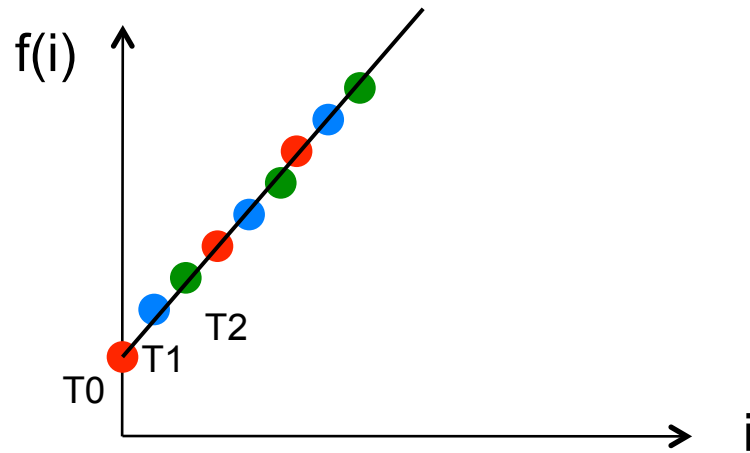
```
double f(int i) {  
    int j, start = i*(i+1)/2, finish = start + i;  
    double return_val = 0.0;  
  
    for (j = start; j <= finish; j++) {  
        return_val += sin(j);  
    }  
    return return_val;  
} /* f */
```

Nossa definição da função f.

Como alocar as iterações às threads?

```
sum = 0.0;  
for (i = 0; i <= n; i++)  
    sum += f(i);
```





```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

Thread	Iterations
0	$0, n/t, 2n/t, \dots$
1	$1, n/t + 1, 2n/t + 1, \dots$
\vdots	\vdots
$t - 1$	$t - 1, n/t + t - 1, 2n/t + t - 1, \dots$

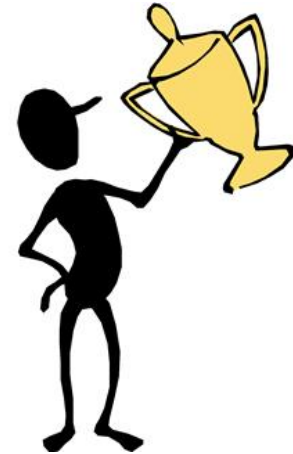
Atribuição de trabalho
usando particionamento
cíclico.

Resultados

- $f(i)$ chama a função $\sin i$ vezes.
- Assuma que o tempo para executar $f(2i)$ é aproximadamente duas vezes maior que o tempo para executar $f(i)$.
- $n = 10,000$
 - uma thread
 - tempo de execução = 3.67 seconds.

Resultados

- $n = 10,000$
 - duas threads
 - atribuições default
 - tempo de execução = 2.76 seconds
 - speedup = 1.33
- $n = 10,000$
 - duas threads
 - atribuição cíclica
 - tempo de execução= 1.84 segundos
 - speedup = 1.99



A cláusula *schedule*

■ Default schedule:

```
sum = 0.0;
#   pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum)
    for (i = 0; i <= n; i++)
        sum += f(i);
```

■ Cyclic schedule:

```
sum = 0.0;
#   pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum) schedule(static,1)
    for (i = 0; i <= n; i++)
        sum += f(i);
```

schedule (type , chunksize)

- *type* pode ser:
 - *static*: as iterações são atribuídas às threads antes que o laço seja executado.
 - *dynamic* ou *guided*: as iterações são atribuídas às threads enquanto o laço está sendo executando.
 - *auto*: o compilador e/ou o sistema de *run-time* determinam o escalonamento.
 - *runtime*: o escalonamento é determinado pelo *run-time*.
- *chunksize* é um inteiro positivo.

Escalonamento *static*

12 iterações, 0, 1, . . . , 11, e três threads

```
schedule(static, 1)
```

Thread 0 : 0, 3, 6, 9

Thread 1 : 1, 4, 7, 10

Thread 2 : 2, 5, 8, 11

Escalonamento *static*

12 iterações, 0, 1, . . . , 11, e três threads

```
schedule(static, 2)
```

Thread 0 : 0, 1, 6, 7

Thread 1 : 2, 3, 8, 9

Thread 2 : 4, 5, 10, 11

Escalonamento *static*

12 iterações, 0, 1, . . . , 11, e três threads

```
schedule(static, 4)
```

Thread 0 :	0, 1, 2, 3
Thread 1 :	4, 5, 6, 7
Thread 2 :	8, 9, 10, 11

Escalonamento *dynamic*

- As iterações são também quebradas em pedaços consecutivos de tamanho **chunksize**.
- Cada thread executa um pedaço e quando uma thread termina o seu pedaço ela pede outro pedaço ao sistema de *run-time*.
- Isto continua até que as iterações sejam concluídas.
- Se **chunksize** é omitida. Quando isto ocorre, é usado 1 para o **chunksize**.

Escalonamento *guided*

- Cada thread também executa o seu pedaço e quando a thread termina, ela requisita outro.
- No entanto, ao usar *guided*, quando os pedaços vão sendo terminados o tamanho dos novos pedaços vai reduzindo (ex. para $\text{\#iterações restantes} / \text{\#threads}$)
- Se **chunksize** não é especificado, o tamanhos dos novos pedaços vão reduzindo até 1.
- Se **chunksize** é especificado, o pedaço diminui para **chunksize**, com a exceção de que o último pedaço pode ser menor que **chunksize**.

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Atribuição das iterações da regra do trapézio (1–9999) usando um escalonamento *guided* com duas threads.

Escalonamento *runtime*

- O sistema usa a variável de ambiente **OMP_SCHEDULE** para determinar em tempo de execução como escalonar o laço.
- A variável de ambiente **OMP_SCHEDULE** pode receber qualquer valor possível para ser usado por escalonamentos *static*, *dynamic*, ou *guided*.

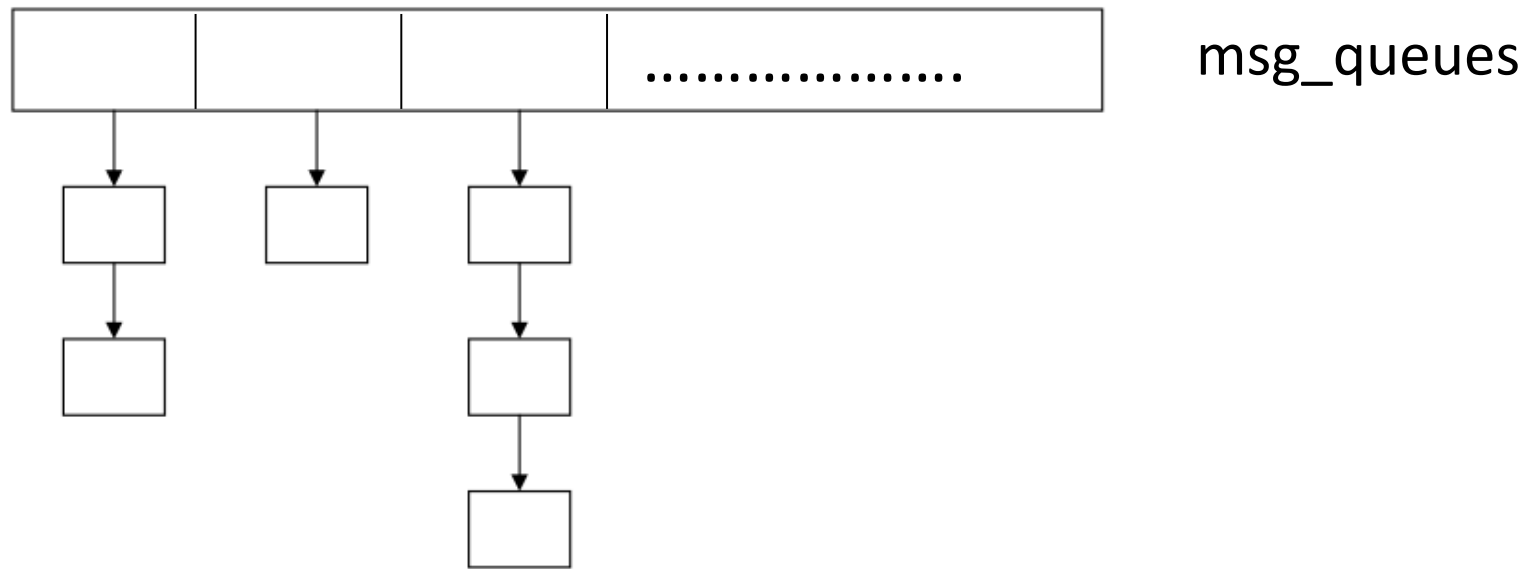


PRODUTORES AND CONSUMIDORES

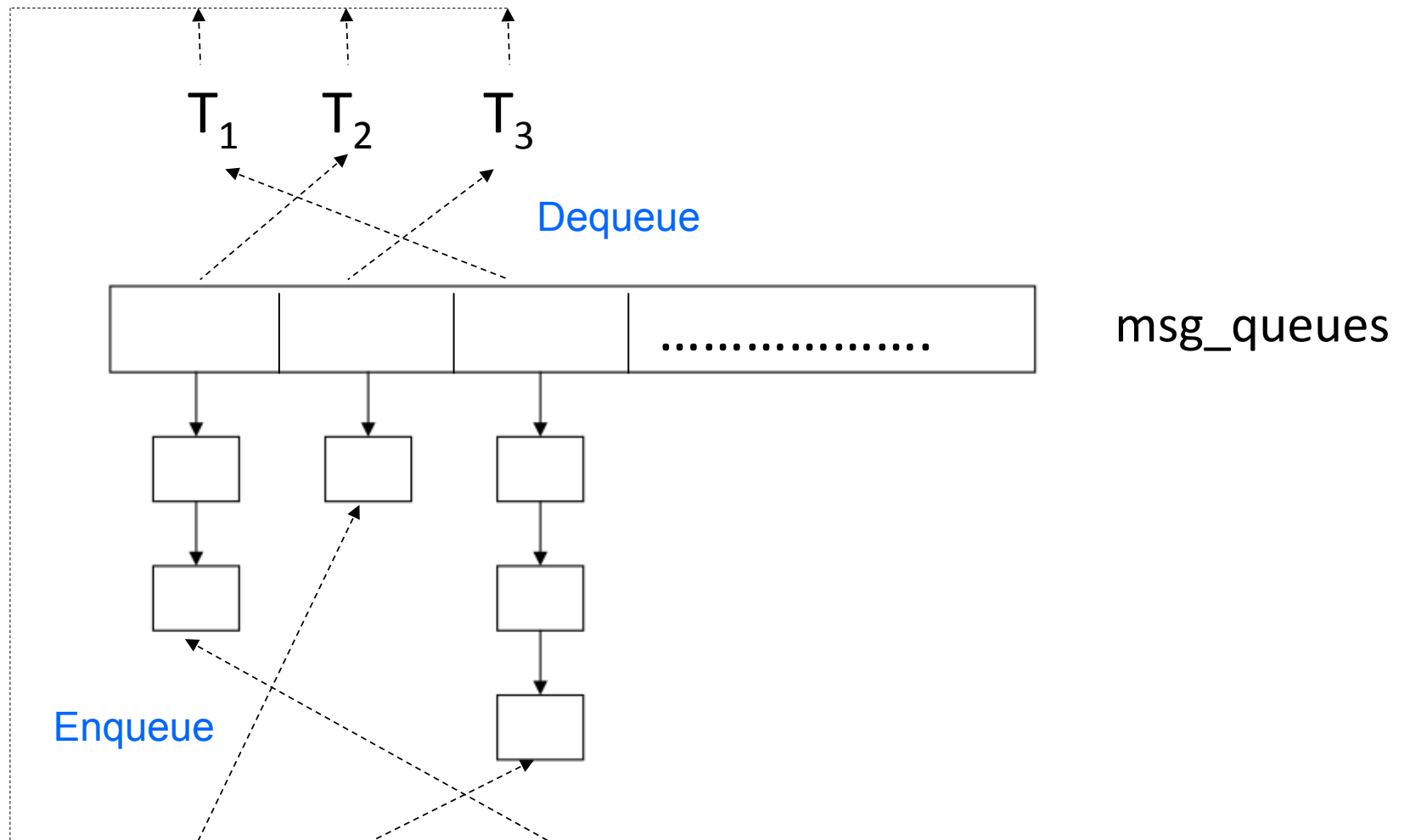
Filas

- Pode ser vista como a abstração de uma linha de clientes esperando para pagar por suas compras em um supermercado.
- Uma estrutura de dados muito usada em aplicações *multithread*.
- Por exemplo suponha que temos várias threads “produtoras” and várias threads “consumidoras”.
 - Threads produtoras podem “produzir” pedidos por dados.
 - Threads consumidoras podem “consumir” os pedidos encontrando ou gerando o dado.

Filas (estrutura)



Filas (operação)



Passagem de Mensagens (1)

- Cada thread pode ter uma fila compartilhada de mensagens, e quando uma thread quer “enviar uma mensagem” para outra thread, ela pode colocar a mensagem na fila da thread de destino.
- Uma thread pode receber uma mensagem removendo a mensagem na cabeça de sua fila.

Passagem de Mensagens (2)

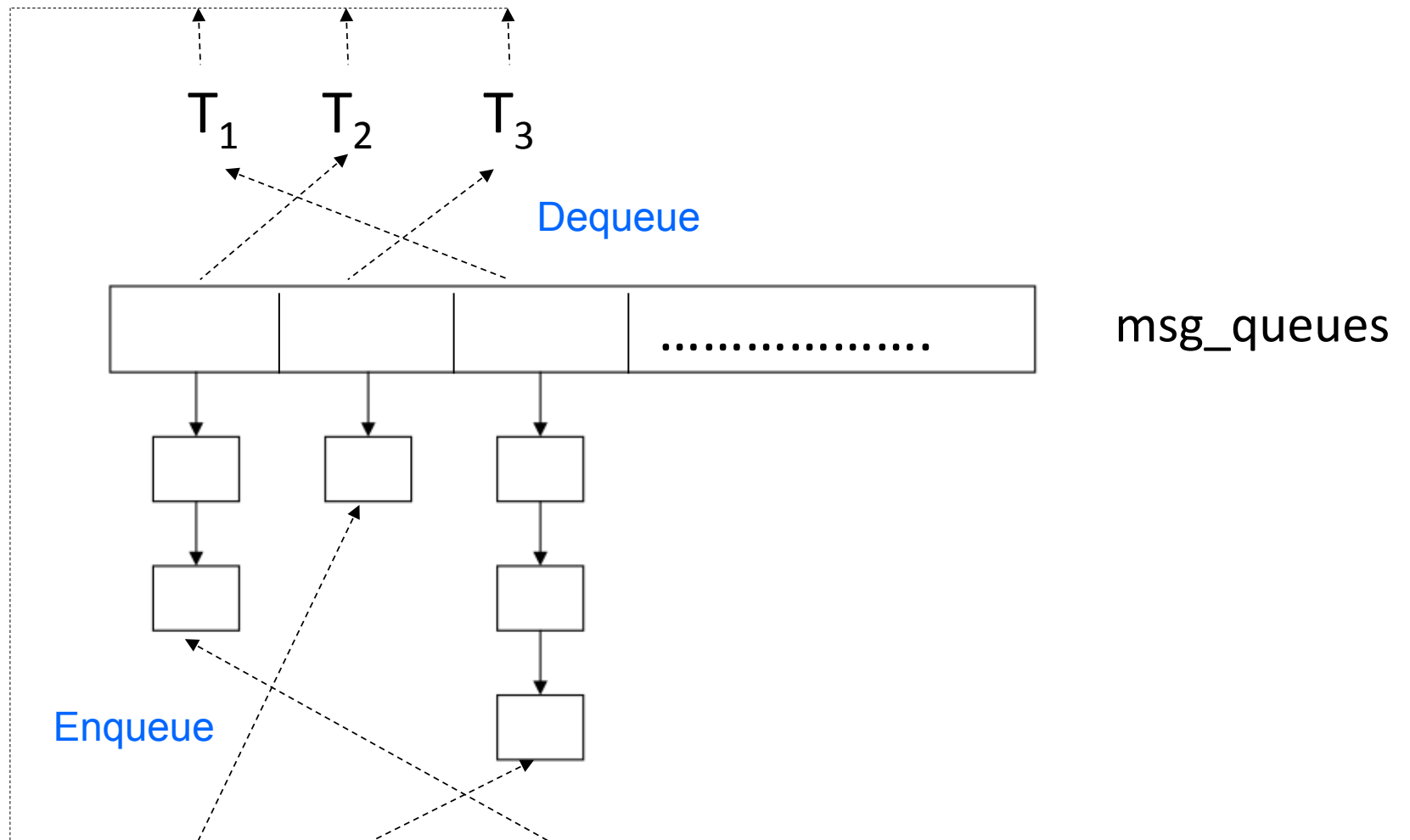
```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {  
    Send_msg();  
    Try_receive();  
}  
  
while (!Done())  
    Try_receive();
```


Enviando Mensagens (3)

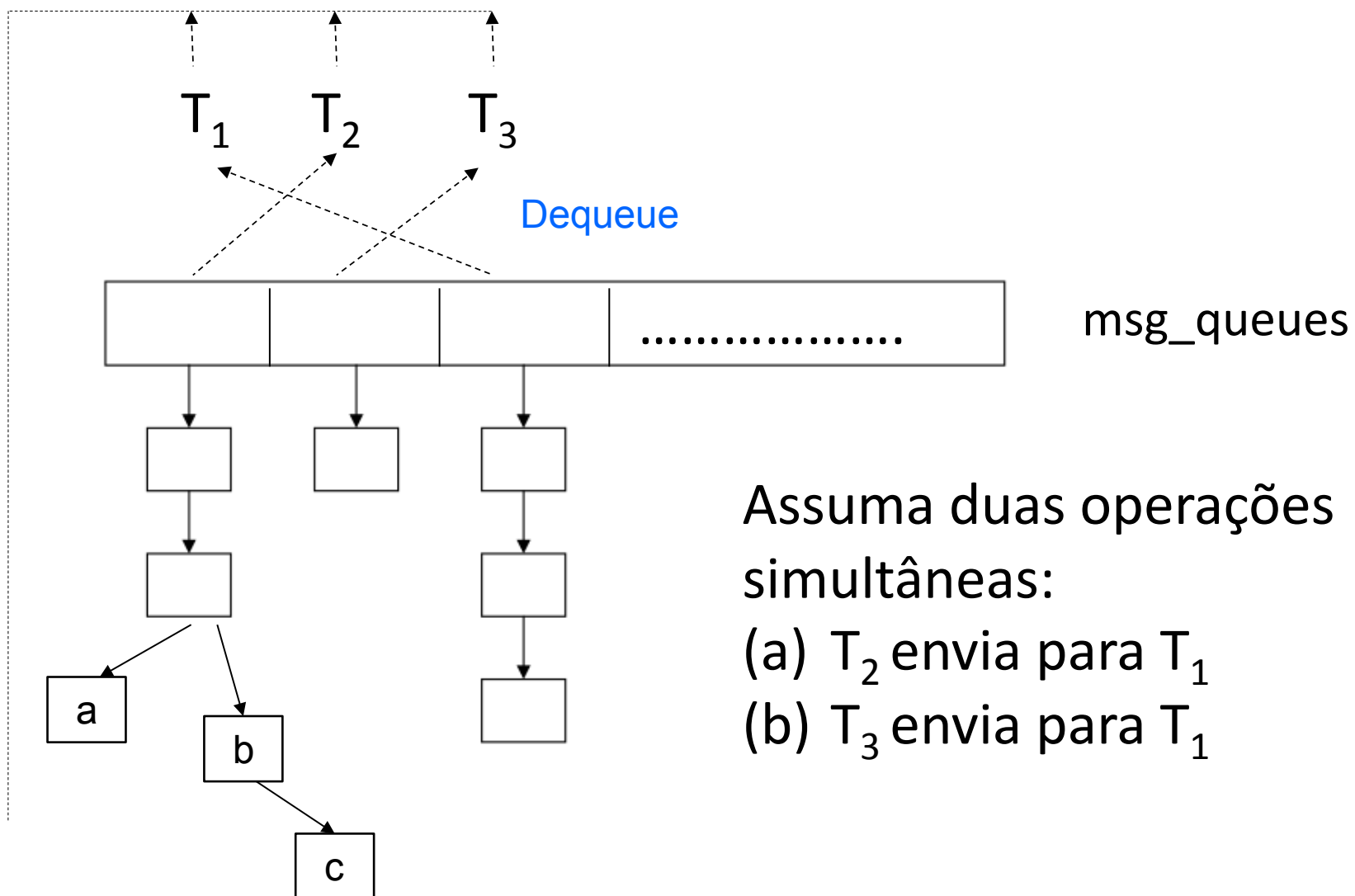
```
mesg = random();  
dest = random() % thread_count;  
  
Enqueue(queue, dest, my_rank, mesg);
```

Qual o problema com este código?

Filas (operação)



Filas (operação)



Enviando Mensagens (3)

```
msg = random();  
dest = random() % thread_count;  
# pragma omp critical  
  Enqueue(queue, dest, my_rank, msg);
```

Recebendo Mensagens (*Try_receive*)

- Como detectar que não existem mais mensagens para uma thread?

Usar um contador *queue_size* (protegido)

- Será que eu posso fazer uma operação de Enqueue e uma Dequeue simultâneas?
- Isto funcionaria sempre?

Recebendo Mensagens

```
if (queue_size == 0) return;  
else if (queue_size == 1)  
#    pragma omp critical  
    Dequeue(queue, &src, &msg);  
else  
    Dequeue(queue, &src, &msg);  
Print_message(src, msg);
```

Recebendo Mensagens

- Ainda assim eu preciso proteger `queue_size`?
- Qual o impacto disto?
- Como resolver?
- Usar um contador para *enqueue* e outro para *dequeue*


`queue_size = enqueued - dequeued`

 Quem acessa este?

Detecção de Término (1)

Qual o problema com este código?

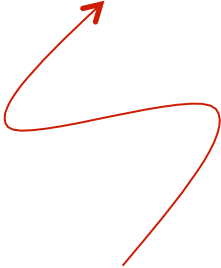
```
queue_size = enqueued - dequeued;  
if (queue_size == 0)  
    return TRUE;  
else  
    return FALSE;
```



E se outra thread muda *enqueued* logo após *queue_size* ter sido lida?

Detecção de Término (2)


```
int Done(struct queue_s* q_p, int done_sending, int thread_count) {  
    int queue_size = q_p->enqueued - q_p->dequeued;  
    if (queue_size == 0 && done_sending == thread_count)  
        return 1;  
    else  
        return 0;  
}
```



cada thread incrementa
done_sending depois de
terminar o seu laço for
(done_sending++)

Detecção de Término (2)

```
int Done(struct queue_s* q_p, int done_sending, int thread_count) {  
    int queue_size = q_p->enqueued - q_p->dequeued;  
    if (queue_size == 0 && done_sending == thread_count)  
        return 1;  
    else  
        return 0;  
}
```



Thread A muda enqueued
e done_sending aqui

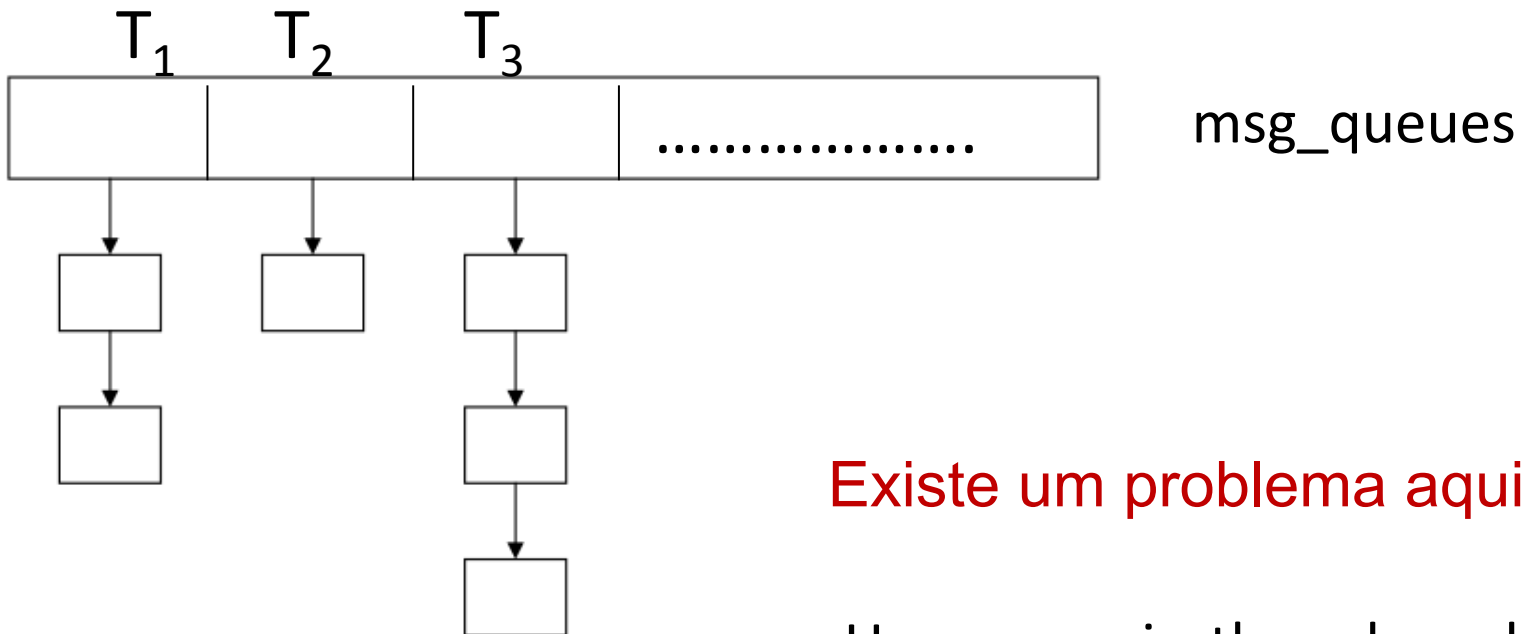
- (1) Thread A envia uma mensagem para B
- (2) Thread B não detecta nada, e acha que acabou
- (3) Mensagem de A para B foi perdida?

Usa valor antigo de done_sending
pois é passado por cópia!

Startup (1)

- Quando o programa inicia a execução, uma única thread, a thread mestre, recebe argumentos, através da linha de comando, e aloca um array de filas de mensagens, uma para cada thread.
- Este array precisa ser compartilhado, já que qualquer thread pode enviar para qualquer outra thread, podendo colocar uma mensagem em qualquer uma das filas.

Criando as filas



Existe um problema aqui?

Uma ou mais threads podem terminar de alocar as suas filas antes de outras threads.

O que fazer?

Startup (2)

- Precisamos de uma barreira explícita de modo que quando uma thread encontrar a barreira, ela bloqueia até que todas as threads no time tenham alcançado a barreira.
- Depois que todas as threads tenham alcançado a barreira, todas as threads no time podem continuar.

```
# pragma omp barrier
```

A Diretiva *Atomic* (1)

- Diferentemente da diretiva *critical* ela somente protege seções críticas que consistem de uma única sentença em C.

```
# pragma omp atomic
```

- Além disto as sentença devem possuir algum dos seguintes formatos:

```
x <op>= <expression>;  
x++;  
++x;  
x--;  
--x;
```

A Diretiva *Atomic* (2)

- Onde <op> pode ser um dos seguintes operadores binários

`+, *, -, /, &, ^, |, <<, or >>`

- Muitos processadores provêm uma instrução *load-modify-store*.
- Uma seção crítica que somente faz *load-modify-store*, pode ser protegida de maneira muitos mais eficiente usando esta instrução especial do que usando técnicas mais gerais para seções críticas.

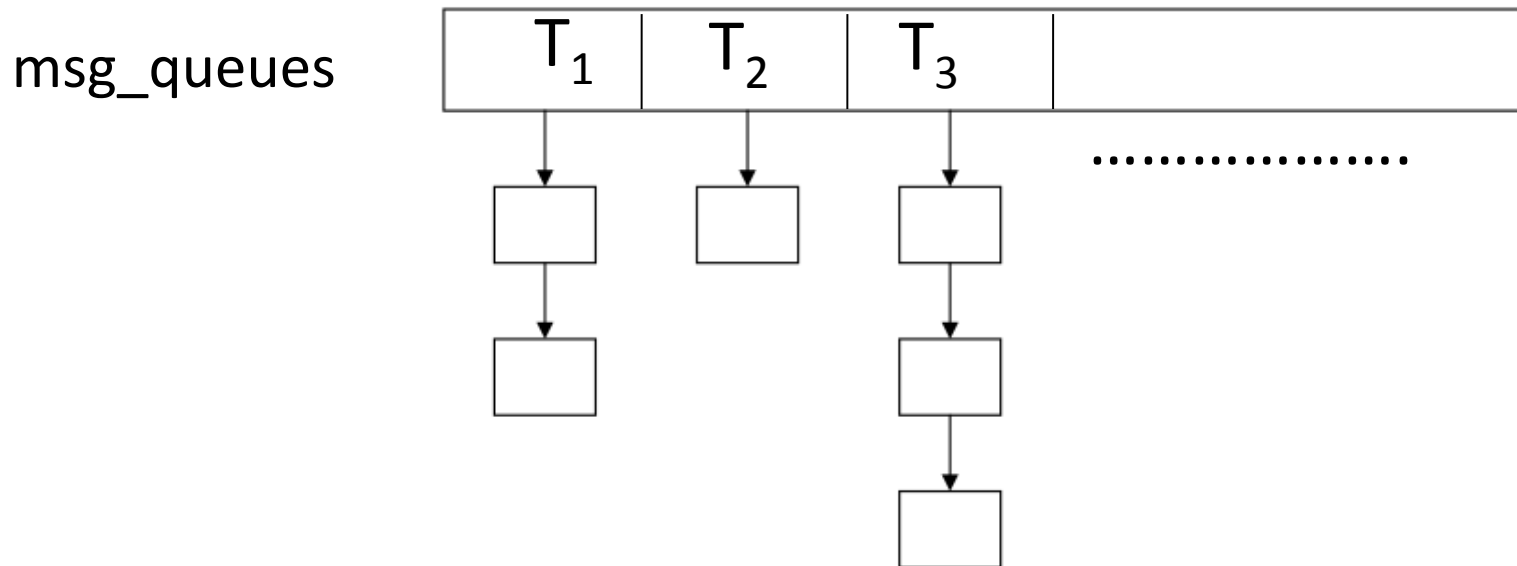
Cuidado....

- Quais as garantias que existem neste código?

```
#pragma omp atomic  
x += y++;
```


Blocos que usam *critical* ou *atomic*

```
done_sending++;  
Enqueue(q_p, my_rank, msg);  
Dequeue(q_p, &src, &msg);
```



É preciso mesmo ter estas três como *critical*?

Seções Críticas

- OpenMP permite adicionar um nome a uma seção crítica:

```
# pragma omp critical(name)
```

- Quando fazemos isto dois blocos protegidos com diretivas *critical* e nomes diferentes podem executar simultaneamente.
- No entanto, os nomes são definidos durante a compilação, e queremos uma seção crítica distinta para cada fila de uma thread.

O que fazer?

Locks

- Uma lock consiste de uma estrutura de dados e um conjunto de funções que permitem o programador forçar explicitamente exclusão mútua em uma seção crítica.



Locks

```
/* Executed by one thread */  
Initialize the lock data structure;  
. . .  
/* Executed by multiple threads */  
Attempt to lock or set the lock data structure;  
Critical section;  
Unlock or unset the lock data structure;  
. . .  
/* Executed by one thread */  
Destroy the lock data structure;
```

Funções com locks

```
void omp_init_lock(omp_lock_t*    lock_p    /* out */);  
void omp_set_lock(omp_lock_t*    lock_p    /* in/out */);  
void omp_unset_lock(omp_lock_t*  lock_p    /* in/out */);  
void omp_destroy_lock(omp_lock_t* lock_p    /* in/out */);
```

Usando Locks em um programa com passagem de mensagens

```
# pragma omp critical  
/* q_p = msg_queues[dest] */  
Enqueue(q_p, my_rank, msg);
```

```
/* q_p = msg_queues[dest] */  
omp_set_lock(&q_p->lock);  
Enqueue(q_p, my_rank, msg);  
omp_unset_lock(&q_p->lock);
```

Usando Locks em um programa com passagem de mensagens

```
# pragma omp critical  
  /* q_p = msg_queues[my_rank] */  
  Dequeue(q_p, &src, &msg);
```

```
  /* q_p = msg_queues[my_rank] */  
  omp_set_lock(&q_p->lock);  
  Dequeue(q_p, &src, &msg);  
  omp_unset_lock(&q_p->lock);
```

Alguns desafios

- Não misture os diferentes tipos de exclusão mútua para uma única seção crítica.

```
# pragma omp atomic  
x += f(y);
```

```
# pragma omp critical  
x = g(x);
```

Não existem garantias de que a exclusão será respeitada!!

Alguns desafios

- Não existe garantias de justiça em construções que envolvem exclusão mútua.

```
while(1) {  
    . . .  
    #pragma omp critical  
    x = g(my_rank);  
    . . .  
}
```

Alguns desafios

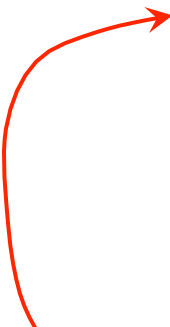
- Pode ser perigoso aninhar construções de exclusão mútua.

Por quê?

```
1  # pragma omp critical
2  y = f(x);
   . . .
4  double f(double x) {
5  # pragma omp critical
6  z = g(x);  /* z is shared */
   . . .
}
```

Deadlock !!

```
1  # pragma omp critical
2  y = f(x);
   . . .
4  double f(double x) {
5  # pragma omp critical
   z = g(x);  /* z is shared */
   . . .
}
```



Fica bloqueado em 5 e não consegue voltar para 2!!

Self Deadlock !!

Thread u

block all other critical
and calls f

```
# pragma omp critical
1 y = f(x);
. . .
double f(double x) {
# 2 pragma omp critical
  z = g(x); /* z is shared */
  . . .
}
```

2 waits for 1, but 1
cannot continue
because of 2

Deadlock !!

Thread u

0 Critical section 1

1 Critical section 2

Thread v

0 Critical section 2

1 Critical section 1

Time	Thread u	Thread v
0	Enter crit. sect. one	Enter crit. sect. two
1	Attempt to enter two	Attempt to enter one
2	Block	Block

Multiplicação Matrix-vetor (1)

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

a_{00}	a_{01}	\cdots	$a_{0,n-1}$	$\begin{matrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{matrix}$	$=$	y_0
a_{10}	a_{11}	\cdots	$a_{1,n-1}$			y_1
\vdots	\vdots		\vdots			\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$			$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$
\vdots	\vdots		\vdots			\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$			y_{m-1}

```

for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}

```

Pode-se usar *parallel for* no laço externo?

Multiplicação Matrix-vetor (2)

```
# pragma omp parallel for num_threads(thread_count) \
    default(none) private(i, j) shared(A, x, y, m, n)
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

Tempos de execução e eficiência
na multiplicação matriz-vetor
(tempo em segundos)

Threads	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Multiplicação Matrix-vetor (2)

E quem são os vilões?

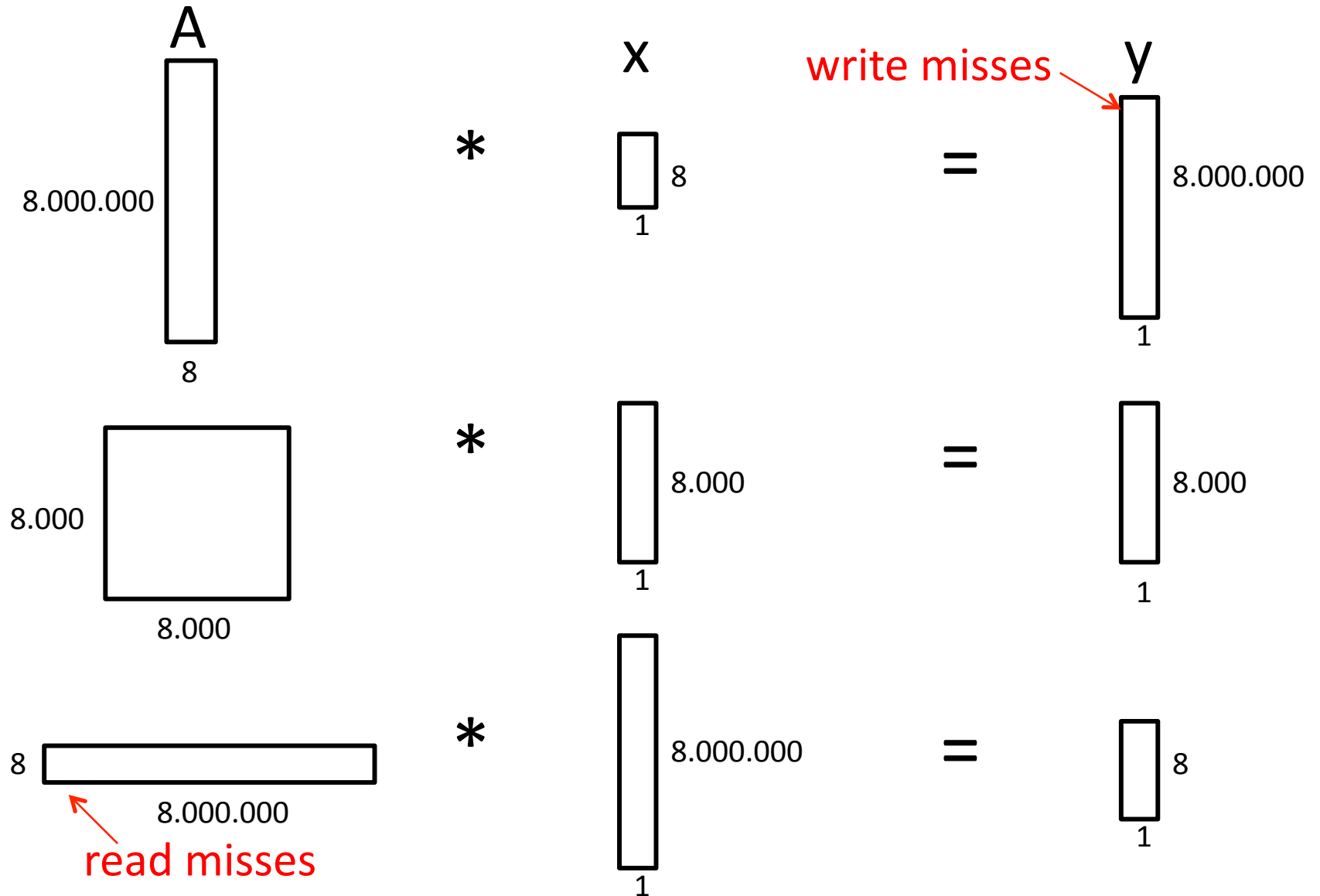
```
# pragma omp parallel for num_threads(thread_count) \
    default(none) private(i, j) shared(A, x, y, m, n)
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```


Multiplicação Matrix-vetor (2)

E quem são os vilões?

```
# pragma omp parallel for num_threads(thread_count) \
    default(none) private(i, j) shared(A, x, y, m, n)
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

Multiplicação Matrix-vetor (2)



Thread-Safety

```
void Tokenize(  
    char* lines[] /* in/out */,  
    int line_count /* in */,  
    int thread_count /* in */) {  
    int my_rank, i, j;  
    char *my_token;  
  
    # pragma omp parallel num_threads(thread_count) \  
        default(none) private(my_rank, i, j, my_token) \  
        shared(lines, line_count)  
    {  
        my_rank = omp_get_thread_num();  
    # pragma omp for schedule(static, 1)  
        for (i = 0; i < line_count; i++) {  
            printf("Thread %d > line %d = %s", my_rank, i, lines[i]);  
            j = 0;  
            my_token = strtok(lines[i], " \t\n");  
            while ( my_token != NULL ) {  
                printf("Thread %d > token %d = %s\n", my_rank, j, my_token);  
                my_token = strtok(NULL, " \t\n");  
                j++;  
            }  
        } /* for i */  
    } /* omp parallel */  
  
} /* Tokenize */
```

Comentários finais (1)

- OpenMP é um padrão para programação de sistemas de memória compartilhada.
- OpenMP usa funções especiais e diretivas de pre-processamento chamadas *pragmas*.
- Programas em OpenMP iniciam múltiplas threads e não múltiplos processos.
- Muitas diretivas OpenMP podem ser modificadas por cláusulas.

Comentários finais (2)

- O maior problema no projeto de programas de memória compartilhada é a possibilidade de corridas.
- OpenMP provê vários mecanismos para garantir exclusão mútua em seções críticas.
 - Diretivas *critical*
 - Diretivas *critical* para nomes
 - Diretivas *Atomic*
 - *Locks* simples

Comentários finais (3)

- A maior parte dos sistemas usam por *default* uma partição por blocos das iterações do laço paralelizado.
- OpenMP oferece uma variedade de opções de escalonamento.
- Em OpenMP o escopo de uma variável é uma coleção de *threads* que podem acessar esta variável.

Comentários finais (4)

- Uma redução é uma operação que repetidamente aplica o mesmo operador de redução a uma sequência de operados para obter um único resultado.