

MO644/MC970

Programação em Memória Compartilhada usando OpenMP

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

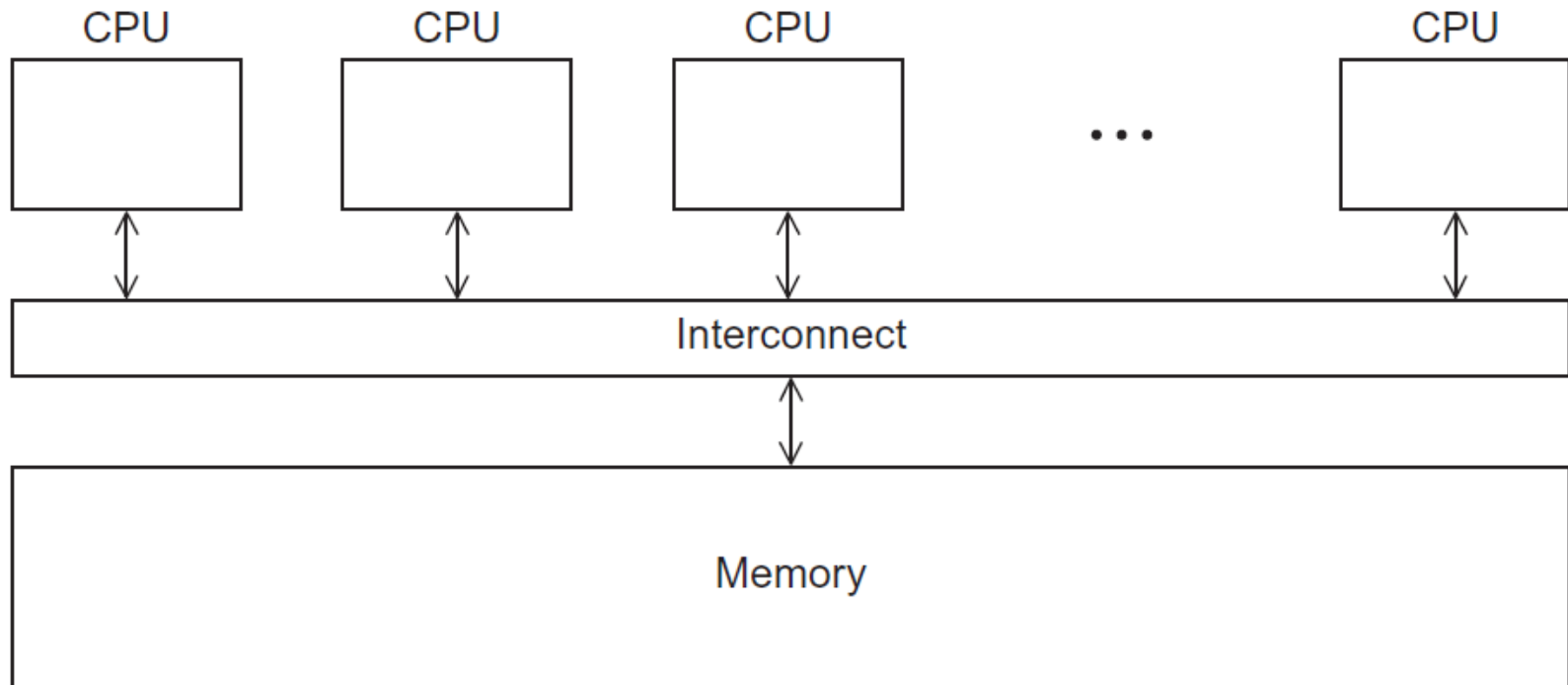
Roteiro

- Escrevendo programas usando OpenMP
- Usando OpenMP para paralelizar laços seriais com pequenas mudanças no código fonte
- Explorar paralelismo de threads
- 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
```

executando com 4 threads

compilando

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

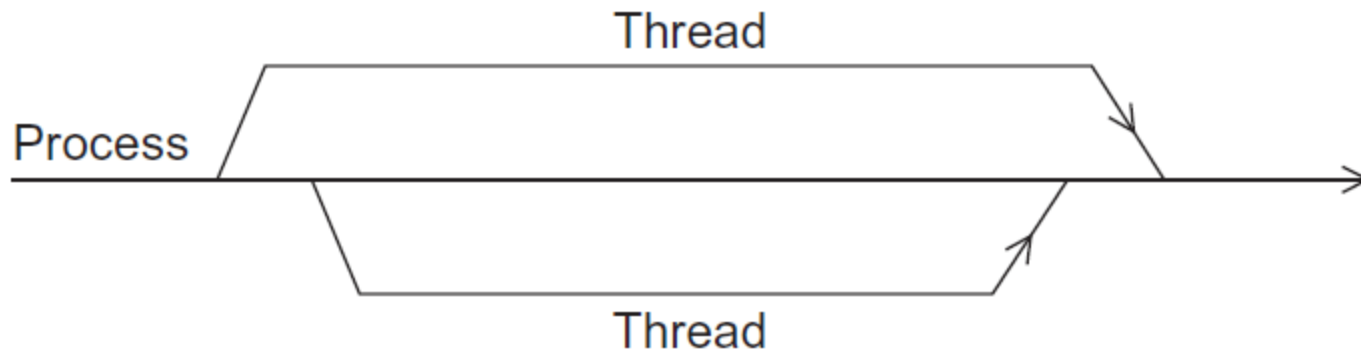
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

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();

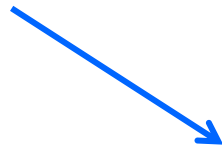
    return 0;
} /* main */

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>
```



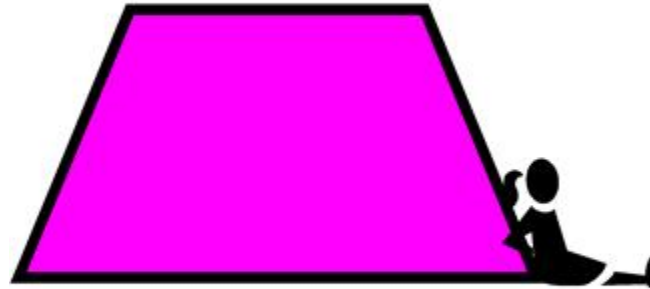
```
#ifdef _OPENMP
```

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

```
#endif
```

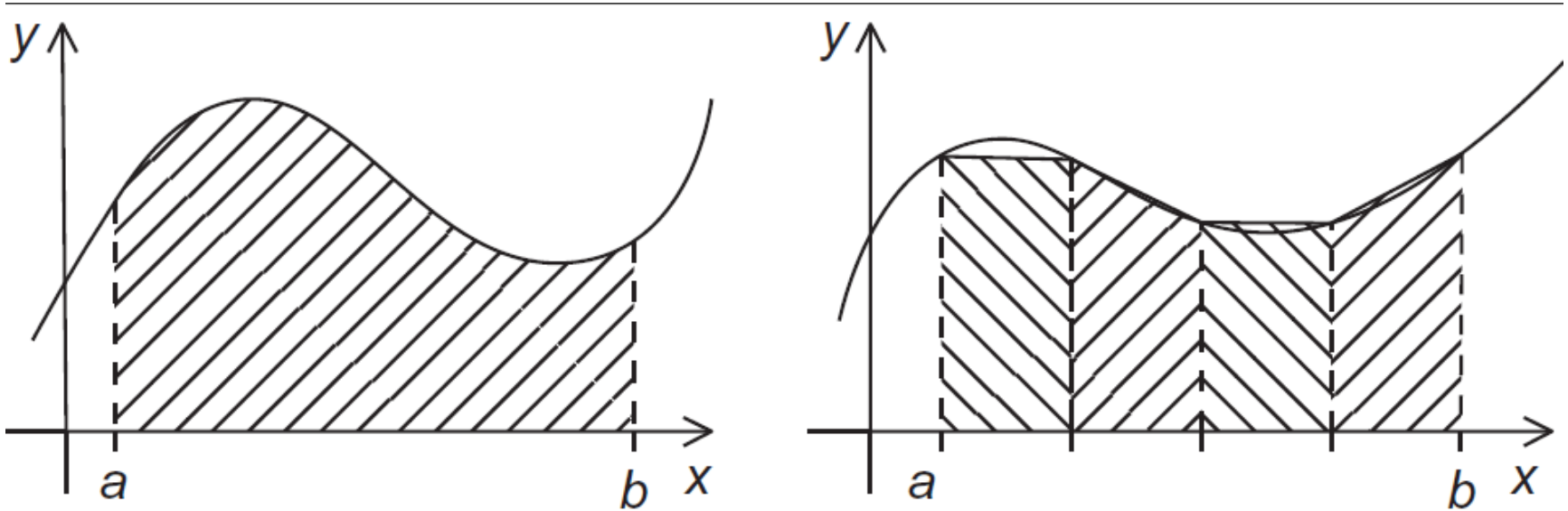
Caso o compilador não aceite OpenMP

```
# 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.

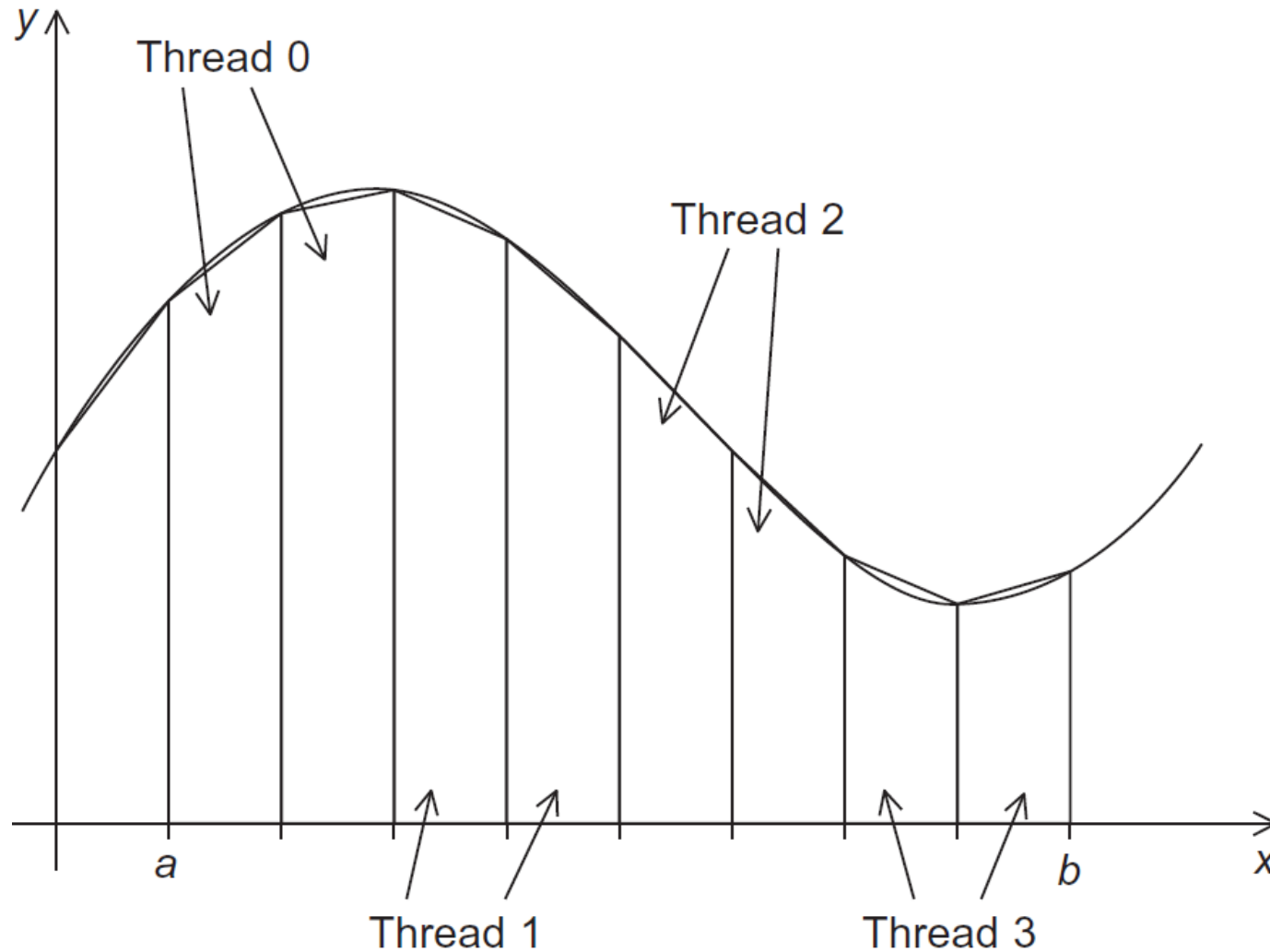


Qual o problema com isto?

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>

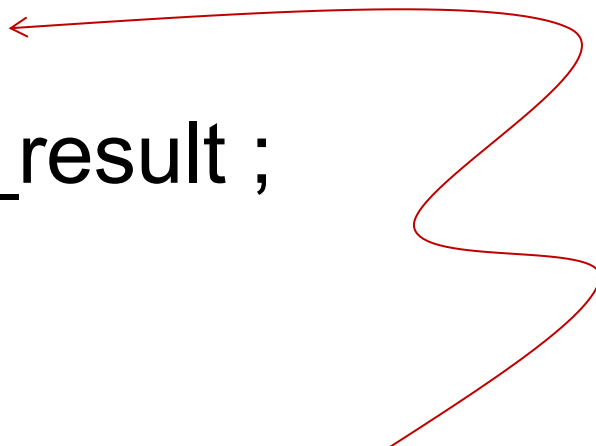
Resultados imprevisíveis podem ocorrer quando duas (ou mais) threads tentam simultaneamente executar:

`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**.





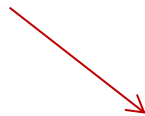
A CLÁUSULA REDUCTION

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;
}
```




Acho que
podemos
fazer melhor.

Nem
eu.

Não
gostei.

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.

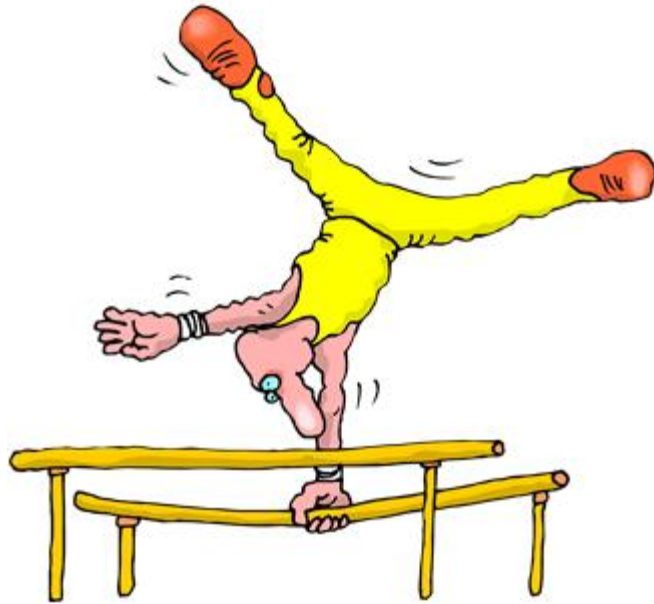
Uma cláusula de redução pode ser adicionada a uma diretiva paralela.

```
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);
```

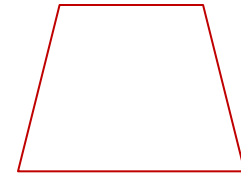


DIRETIVA “PARALLEL FOR”

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

```
fibonacci[0] = fibonacci[1] = 1;  
for (i = 2; i < n; i++)  
    fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
```



```
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];
```



1 1 2 3 5 8
this is correct



but sometimes
we get this

1 1 2 3 2 2

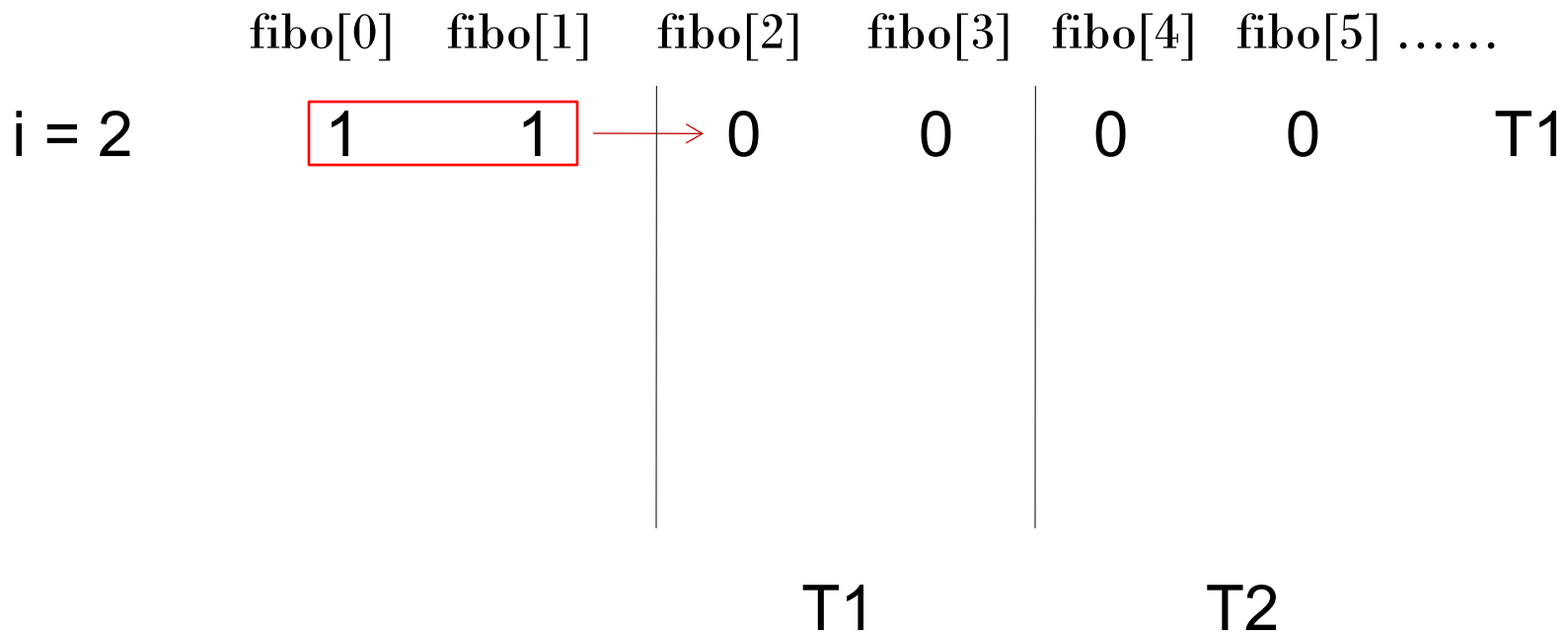
Dependencia de dados

- 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?

```
fibo[ 0 ] = fibo[ 1 ] = 1;
```

```
# pragma omp parallel for num_threads(2)
```

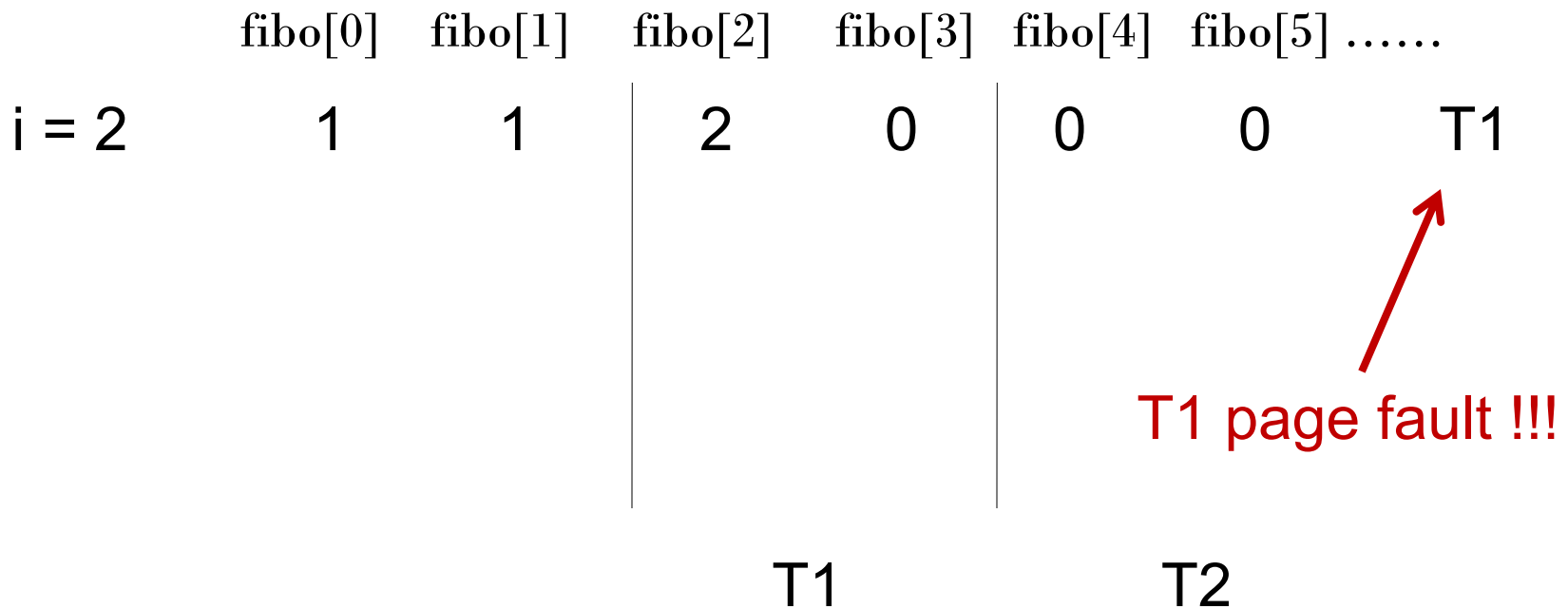
```
for (i = 2; i < n; i++)
```

```
fibo[ i ] = fibo[ i - 1 ] + fibo[ i - 2 ];
```

[illegible]

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?

```
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

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 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		

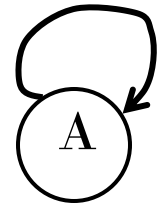
O que ocorreu?



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?

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



- Construa um grafo de dependências 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);
}
```


Qual a solução?

T2 (i=0)
:
factor = 1
:
:
:
:
sum += **factor** / (2*k + 1);

T3 (i=1)
:
factor = -1

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)

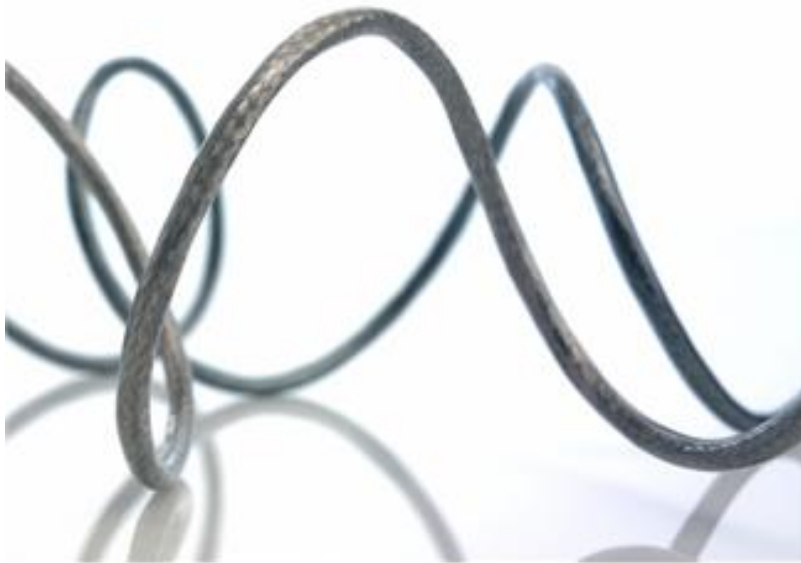
- Deixa o programador definir o escopo de cada variável em um bloco.

`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);
}
```



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

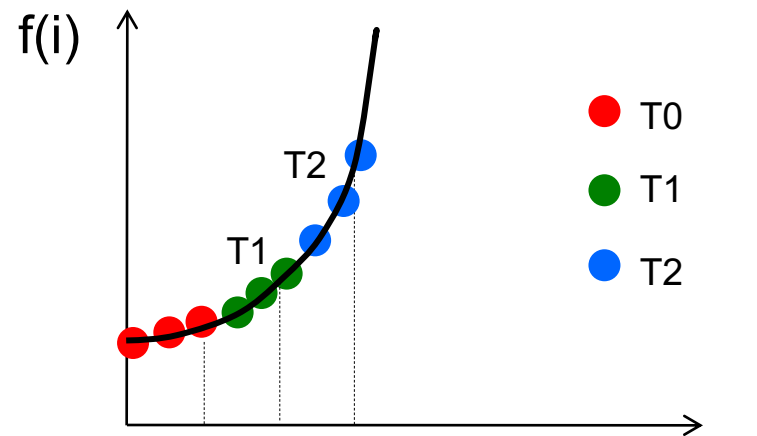
$f(2*i) \approx 2*f(i)$

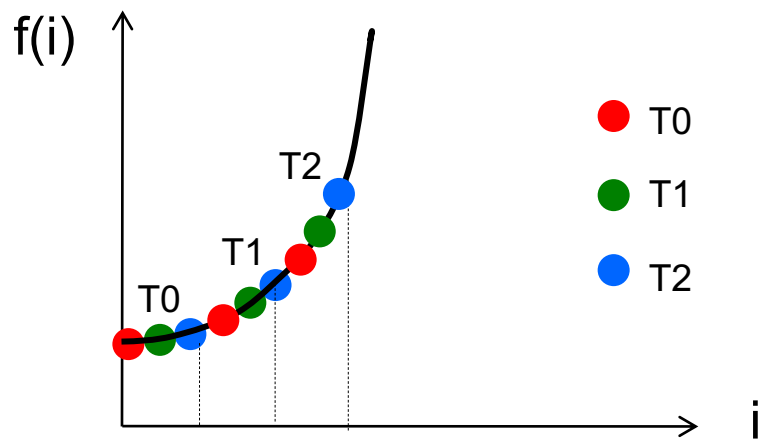
```
double f(int i) {  
    int j, start = i*(i+1)/2, finish = start + i;  
    double return_val = 0.0;  
  
    for (j = start; j <= finish; j++) {  
        return_val += sin(j);  
    }  
    return return_val;  
} /* f */
```

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 $f(i)$.
- $n = 10,000$
 - sequential
 - 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*: os 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*.

A Diretiva *Atomic* (1)

- Diferentemente da diretiva *critical* ela somente proteje 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.

A diretiva barrier

- Depois que todas as threads tenham alcançado a barreira, todas as threads no time podem continuar.

```
# pragma omp barrier
```

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?

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);  
    . . .  
}
```