

# IPPD

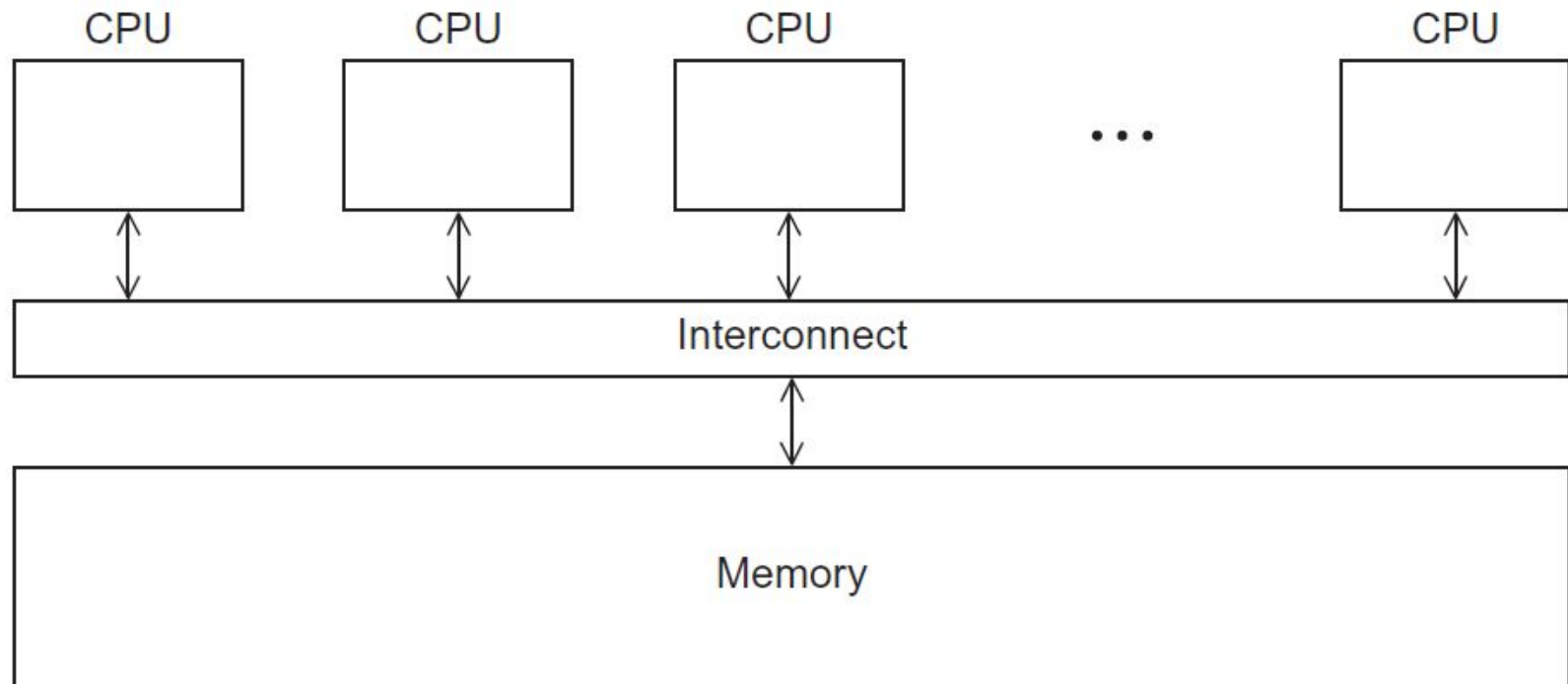
## Hoje: Memória Compartilhada

## OpenMP

**Prof. Dr. Rafael P. Torchelsen**  
[rafael.torchelsen@inf.ufpel.edu.br](mailto:rafael.torchelsen@inf.ufpel.edu.br)

- Uma API para programação paralela de memória compartilhada.
- MP = multiprocessamento
- Projetado para sistemas nos quais cada thread ou processo pode ter acesso a toda a memória disponível.
- O sistema é visto como uma coleção de núcleos ou CPUs, todos com acesso à memória principal.

# Memória Compartilhada



Nosso objetivo principal é estudar esse modelo e não OpenMP

# Pragmas

- Instruções especiais do pré-processador.
- Normalmente, adicionado a um sistema para permitir comportamentos que não fazem parte da especificação C básica.
- Compiladores que não suportam os pragmas os ignoram.

#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();

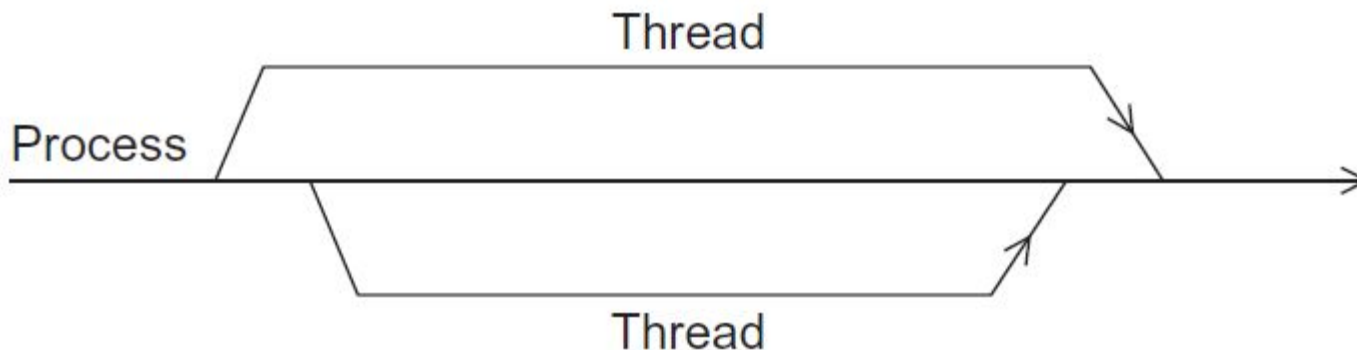
    return 0;
} /* main */

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 */
```

# OpenMP pragmas

- **# pragma omp parallel**
- Diretiva paralela mais básica.
- O número de threads que executam o seguinte bloco de código é determinado pelo sistema em tempo de execução.



- Texto que modifica uma diretiva.
- A cláusula *num\_threads* pode ser adicionada a uma diretiva paralela.
- Permite ao programador especificar o número de threads que devem executar o seguinte bloco.

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

# Terminologia

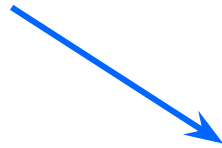
- No jargão do OpenMP, a coleção de threads executando o bloco paralelo - a thread original e as novas threads - é chamada de **team**, a thread original é chamado de **master** e as threads adicionais são chamados de **slaves**.





# Caso o compilador não suporte OpenMP

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

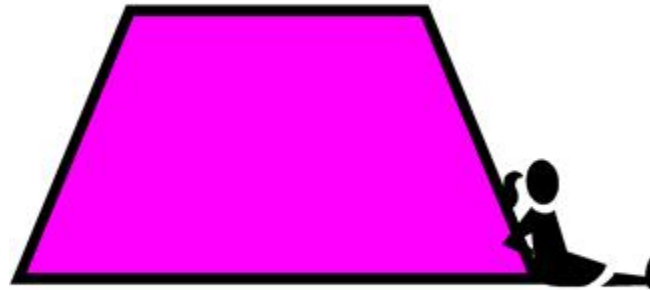


```
#ifdef _OPENMP  
# include <omp.h>  
#endif
```

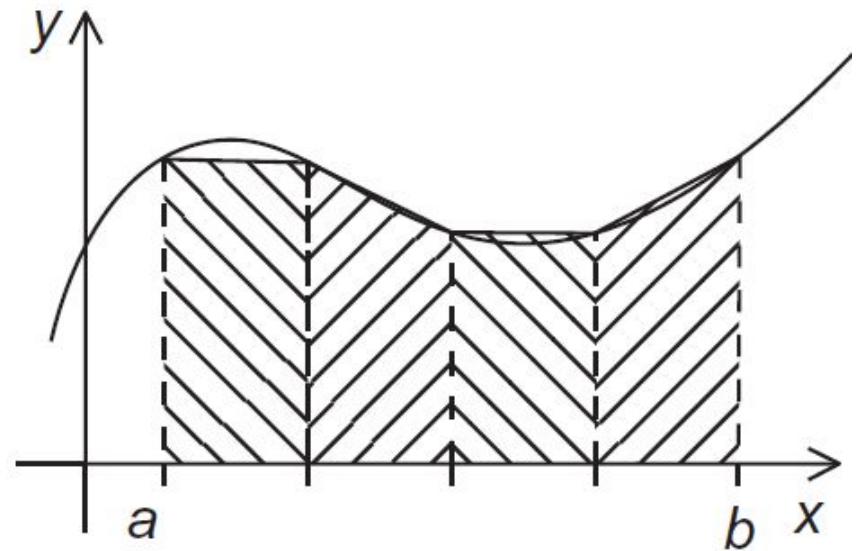
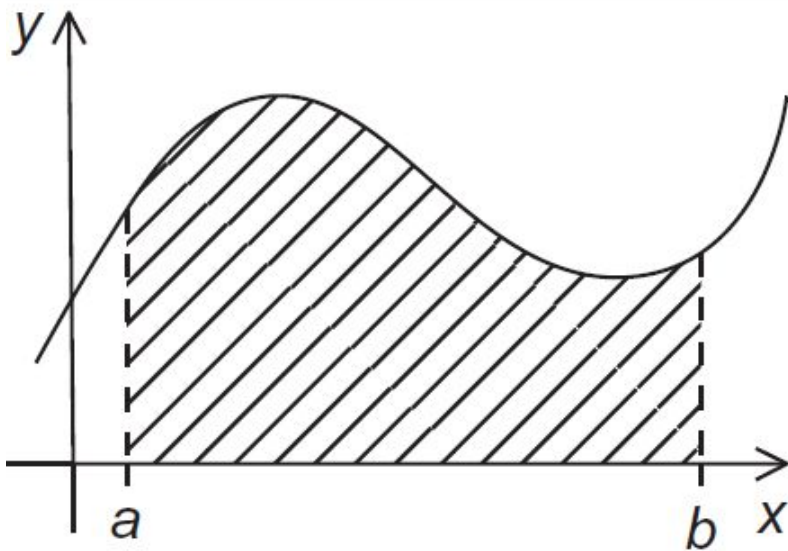
# Caso o compilador não suporte OpenMP

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

# A regra trapezoidal

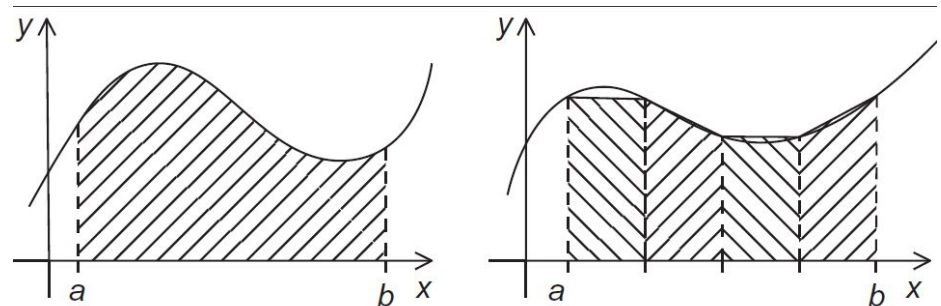


# A regra trapezoidal



# 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;
```



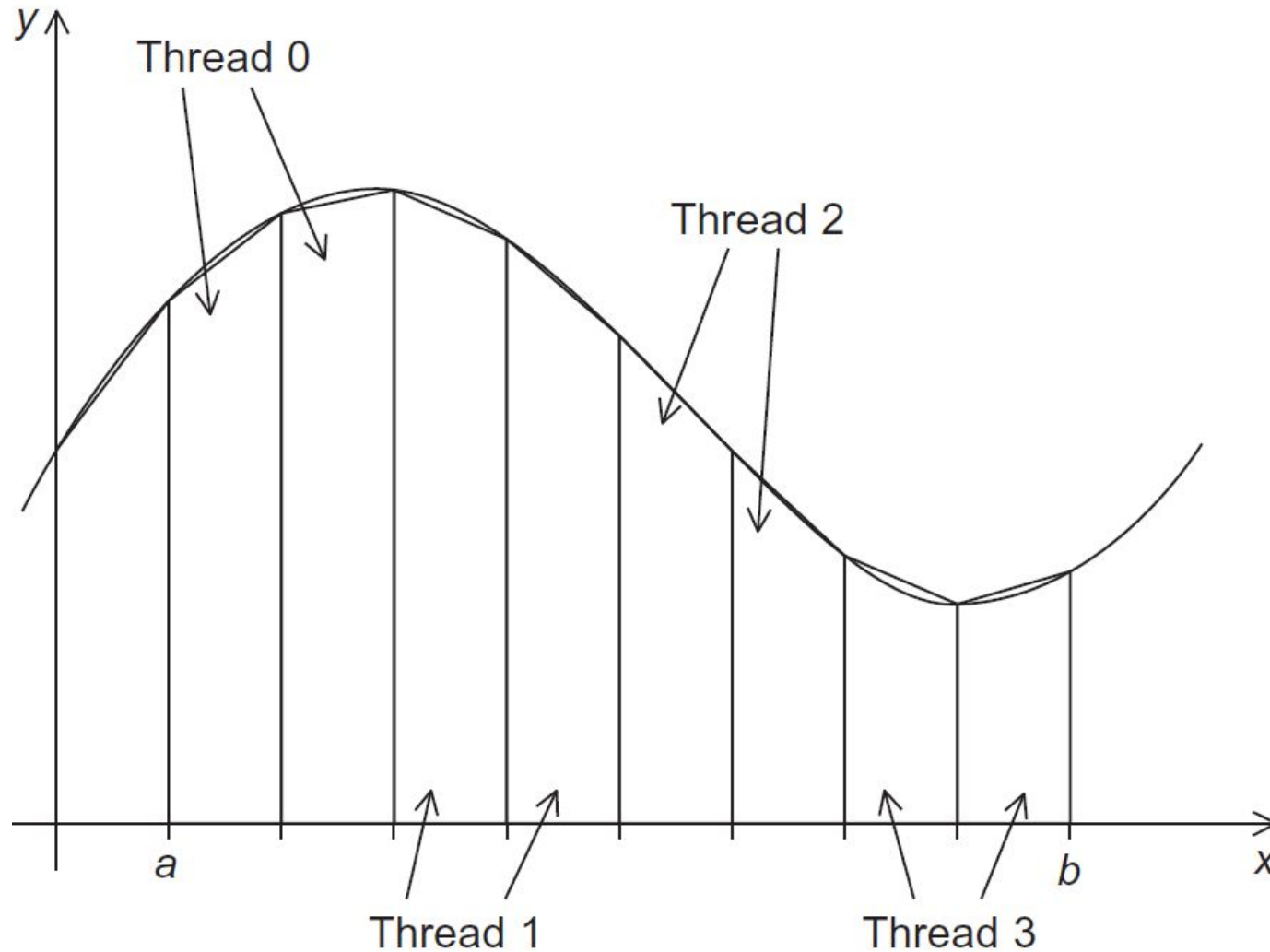
# Primeira implementação em OpenMP



- 1) Identificamos dois tipos de tarefas:
  - a) cálculo das áreas de trapézios individuais, e
  - b) adicionando as áreas de trapézios.
- 2) Não há comunicação entre as tarefas na primeira coleta, mas cada tarefa na primeira coleta se comunica com a tarefa 1b.

- 3) Nós assumimos que haveria muito mais trapézios do que cores.
- Então nós agregamos tarefas atribuindo um bloco contíguo de trapezóides a cada thread (e uma única thread a cada core).

# Atribuição de trapézios às 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>

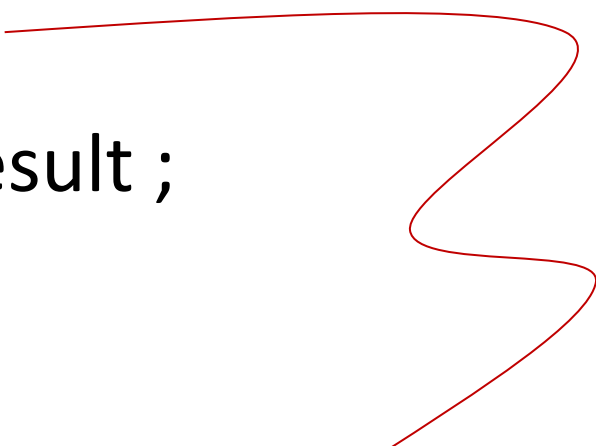
Resultados imprevisíveis quando dois (ou mais) threads tentam executar simultaneamente:

`global_result += my_result ;`



# Exclusão mutua

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



apenas uma thread pode executar  
o seguinte bloco de cada 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 */

```

# Escopo de variáveis



- Na programação serial, o escopo de uma variável consiste nas partes de um programa em que a variável pode ser usada.
- No 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 na **team** tem **escopo compartilhado**.
- Uma variável que só pode ser acessada por uma única thread tem **escopo privado**.
- O escopo padrão para variáveis declaradas antes de um bloco paralelo é o compartilhado.



# Clausula de redução



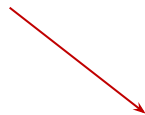


Precisamos desta versão mais complexa para adicionar o cálculo local de cada thread para obter **global\_result**.

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

Embora nós prefiramos isso.

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



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

Se usarmos isso, não há seção crítica!

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

Se consertarmos assim ...

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

... forçamos as threads a executarem sequencialmente.

Podemos evitar esse problema declarando uma variável privada dentro do bloco paralelo e movendo a seção crítica após a chamada de função.

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

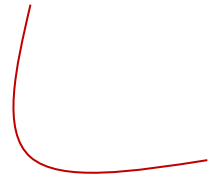
    my_result += Local_trap(double a, double b, int n);
# pragma omp critical
    global_result += my_result;
}
```

# Operador de redução

- Um **operador de redução** é uma operação binária (como adição ou multiplicação).
- Uma **redução** é uma computação que aplica repetidamente o mesmo operador de redução a uma seqüência de operandos para obter um único resultado.
- Todos os resultados intermediários da operação devem ser armazenados na mesma variável: a variável de redução.

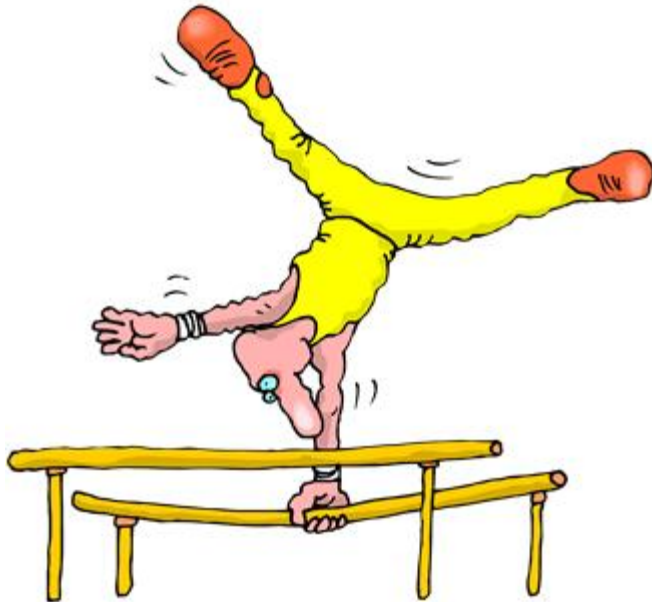
Uma cláusula de redução pode ser adicionada a uma diretiva paralela

```
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);
```

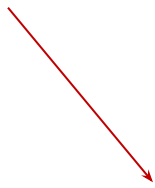
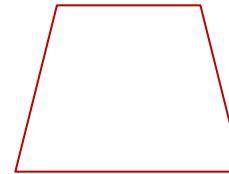
# Diretiva de “for” paralelo



# For paralelo

- Forks uma **team** de threads para executar o seguinte bloco.
- No entanto, o bloco seguinte tem que ser um loop for.
- O sistema paraleliza o **loop for** dividindo as iterações do loop entre as threads.

```
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;
```



```
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;
```



# Só permite as seguintes formas

**for**  $\left( \begin{array}{l} \text{index} = \text{start} ; \text{index} < \text{end} ; \text{index}++ \\ \text{index} = \text{start} ; \text{index} <= \text{end} ; ++\text{index} \\ \text{index} = \text{start} ; \text{index} >= \text{end} ; \text{index}-- \\ \text{index} = \text{start} ; \text{index} > \text{end} ; --\text{index} \\ \text{index} = \text{start} ; \text{index} += \text{incr} \\ \text{index} = \text{start} ; \text{index} -= \text{incr} \\ \text{index} = \text{start} ; \text{index} = \text{index} + \text{incr} \\ \text{index} = \text{start} ; \text{index} = \text{incr} + \text{index} \\ \text{index} = \text{start} ; \text{index} = \text{index} - \text{incr} \end{array} \right)$

- A variável **index** precisar ser um inteiro ou um ponteiro
- A expressão **start**, **end**, e **incr** devem ser compatíveis, por exemplo, se **index** for um ponteiro então **incr** precisa ser do tipo inteiro

- As expressões **start**, **end** e **incr** não devem mudar durante a execução do loop.
- Durante a execução do loop, a variável **index** só pode ser modificada pela expressão de incremento na instrução **for**.

# Dependência de dados

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

note 2 threads

```
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 ];
```

Mas as vezes resulta nisso

1 1 2 3 5 8 13 21 34 55

correto

1 1 2 3 5 8 0 0 0 0

# O que aconteceu?



1. Os compiladores OpenMP não verificam dependências entre as iterações em um loop que está sendo paralelizado.
2. Um loop no qual os resultados de uma ou mais iterações dependem de outras iterações não pode, em geral, ser corretamente paralelizado pelo OpenMP.

# Estimando Pi

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

```
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;
```

# Solução 1

dependência

```
# 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;
```

# Solução 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);
}
```



Escopo privado!



# A cláusula default

- Permite ao programador especificar o escopo de cada variável em um bloco.

`default (none)`

- Com esta cláusula, o compilador exigirá que especifiquemos o escopo de cada variável que usamos no bloco e que tenha sido declarada fora do bloco.

# A cláusula default

```
# 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 loops em OpenMP



# Divisão de tarefas no OpenMP

```
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>
#define THREADS 8
#define N 100
```

Qual o resultado?

```
int main ( ) {
    int i;
```

0	1	2	3	4	5	6	7
0-9	10-19	20-29	30-39	40-49	50-59	60-69	70-79

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

```
for (i = 0; i < N; i++) {
```

```
    printf("Thread %d is doing iteration %d.\n", omp_get_thread_num( ), i);
} /* all threads done */
```

```
printf("All done!\n");
```

```
return 0;
```

```
}
```

Qual o problema com  
isso?

# Static Schedule (Padrão)

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>
#define THREADS 4
#define N 16

int main ( ) {
    int i;
    #pragma omp parallel for schedule(static) num_threads(THREADS)
    for (i = 0; i < N; i++) {
        /* wait for i seconds */
        sleep(i);
        printf("Thread %d has completed iteration %d.\n", omp_get_thread_num( ), i);
    } /* all threads done */
    printf("All done!\n");
    return 0;
}
```

# Dynamic

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>
#define THREADS 4
#define N 16
```

Quanto mais rápido é esse programa?

0	1	2	3
?	?	?	?

```
int main () {
    int i;
    #pragma omp parallel for schedule(dynamic) num_threads(THREADS)
    for (i = 0; i < N; i++) {
        /* wait for i seconds */
        sleep(i);
        printf("Thread %d has completed iteration %d.\n", omp_get_thread_num(), i);
    } /* all threads done */
    printf("All done!\n");
    return 0;
}
```

# Qual o motivo do static ser o padrão?

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>
#define THREADS 16
#define N 100000000
int main () {
    int i;
    printf("Running %d iterations on %d threads dynamically.\n", N, THREADS);
    #pragma omp parallel for schedule(dyn static 1_threads(THREADS))
    for (i = 0; i < N; i++) {
        /* a loop that doesn't take very long */
    }
    /* all threads done */
    printf("All done!\n");
    return 0;
}
```

Quanto tempo  
esse loop  
levou?

Mais que esse

Overhead: Ao final de cada  
tarefa a thread precisa parar e  
pedir a próxima tarefa

# Chunk Sizes

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>
#define THREADS 16
#define N 100000000
#define CHUNK 100
int main () {
    int i;
    printf("Running %d iterations on %d threads dynamically.\n", N, THREADS);
    #pragma omp parallel for schedule(dynamic, CHUNK) num_threads(THREADS)
    for (i = 0; i < N; i++) {
        /* a loop that doesn't take very long */
    } /* all threads done */
    printf("All done!\n");
    return 0;
}
```

Agrupa em **CHUNK**  
tarefas ao distribuir as  
tarefas

Ficou mais rápido  
que a versão  
anterior?



# Guided Schedules

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>
#define THREADS 16
#define N 100000000
```

```
int main () {
    int i;
    printf("Running %d iterations on %d threads guided.\n", N, THREADS);
    #pragma omp parallel for schedule(guided) num_threads(THREADS)
    for (i = 0; i < N; i++) {
        /* a loop that doesn't take very long */
    } /* all threads done */
    printf("All done!\n");
    return 0;
}
```

Em vez de static ou dynamic, podemos especificar guided.

Essa política de agendamento é semelhante a um agendamento dinâmico, exceto que o tamanho do trecho é alterado conforme o programa é executado. Ele começa com grandes blocos, mas se ajusta a tamanhos menores se a carga de trabalho estiver **desbalanceada**.

- O OpenMP divide automaticamente as iterações de loop para nós. Dependendo do nosso programa, o comportamento padrão pode não ser o ideal.
- Para loops em que cada iteração leva aproximadamente o mesmo tempo, os planejamentos estáticos funcionam melhor, pois têm pouca sobrecarga.
- Para loops em que cada iteração pode levar quantidades muito diferentes de tempo, os planejamentos dinâmicos funcionam melhor, pois o trabalho será dividido de maneira mais uniforme entre os segmentos.
- Especificar blocos ou usar uma programação guiada fornece uma compensação entre os dois.
- Escolher o melhor horário **depende** da **compreensão do seu loop**.

# Produtor e consumidor



- Pode ser visto como uma abstração de uma linha de clientes esperando para pagar suas compras em um supermercado.
- Uma estrutura de dados natural para usar em muitos aplicativos multithread.
- Por exemplo, suponha que tenhamos várias threads “produtores” e várias threads de “consumidor”.
- As threads do produtor podem "produzir" solicitações de dados.
- As threads do consumidor podem "consumir" a solicitação localizando ou gerando os dados solicitados.

# Passando Mensagens

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

# Passando Mensagens

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

```
while (!Done())  
    Try_receive();
```

# Enviado mensagens

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

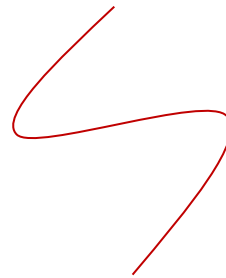
# Recebendo mensagens

```
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);
```



# Detectando o final

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



Cada thread incrementa ao final de cada tarefa

- Quando o programa inicia a execução uma única thread (master) envia tarefas por mensagem para todas as threads.
- Esse array precisa ser compartilhado entre as threads, já que qualquer thread pode enviar para qualquer outra thread, e, portanto, qualquer thread pode enfileirar uma mensagem em qualquer uma das filas.

- Uma ou mais threads podem terminar de alocar suas filas antes de alguma outra thread.
- Precisamos de uma barreira explícita para que, quando uma thread encontrar a barreira, bloqueie até que todos as threads da equipe atinjam a barreira.
- Depois de todos as threads terem atingido a barreira, todas as threads da team podem prosseguir.

```
# pragma omp barrier
```

- Ao contrário da diretiva crítica, ela só pode proteger seções críticas que consistem em uma única instrução de atribuição C.

```
# pragma omp atomic
```

- Além disso, a declaração deve ter uma das seguintes formas:

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

- Aqui <op> pode ser um dos operadores binários  
 $+$ ,  $*$ ,  $-$ ,  $/$ ,  $\&$ ,  $^$ ,  $|$ ,  $<<$ , or  $>>$
- Muitos processadores fornecem uma instrução carregar-modificar-armazenar especial.
- Uma seção crítica que apenas faz um carregar-modificar-armazenar pode ser protegida de forma muito mais eficiente usando essa instrução especial em vez das construções usadas para proteger seções críticas mais gerais.

# Seção crítica

- O OpenMP oferece a opção de adicionar um nome a uma diretiva crítica:  

```
# pragma omp critical(name)
```
- Quando fazemos isso, dois blocos protegidos por diretivas críticas com nomes diferentes podem ser executados simultaneamente.
- No entanto, os nomes são definidos durante a compilação e queremos uma seção crítica diferente para a fila de cada thread.

# Locks

- Um bloqueio consiste em uma estrutura de dados e funções que permitem ao programador aplicar explicitamente a 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;
```



# Usando Locks no programa de envio 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 no programa de envio 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);
```

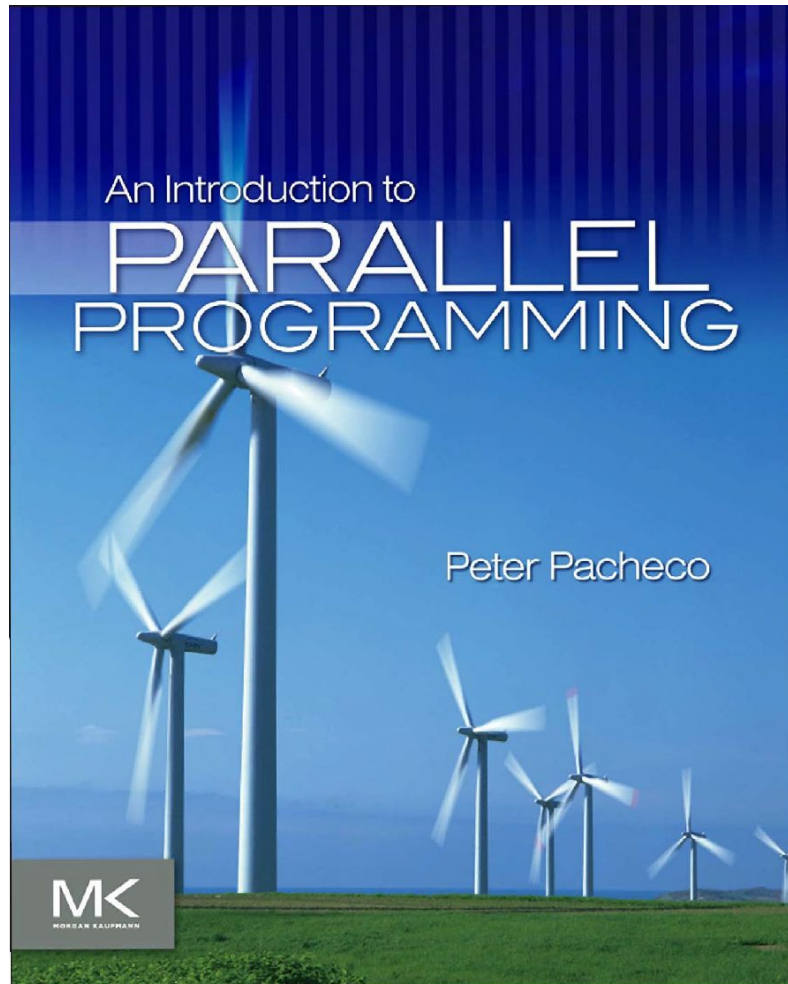
- <http://www.openmp.org/>
  - Tutoriais:  
<http://www.openmp.org/resources/tutorials-articles/>
- Tutorial
  - Vídeos:  
<https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG>
  - Slides:  
[http://www.openmp.org/wp-content/uploads/Intro\\_To\\_OpenMP\\_Mattson.pdf](http://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf)
  - Exercícios:  
[http://www.openmp.org/wp-content/uploads/Mattson\\_OpenMP\\_exercises.zip](http://www.openmp.org/wp-content/uploads/Mattson_OpenMP_exercises.zip)
  - <https://www.ibm.com/developerworks/br/aix/library/au-aix-openmp-framework/index.html>

- Google:
  - Compiling openmp gcc
  - Compiling openmp visual studio
  - Compiling openmp ....

# Merge-Sort

- Tarefa:
  - Implementar o Merge-Sort em OpenMP
  - Gere os mesmo gráficos de antes e compare com o MPI
  - Postar no forum

# Leitura



Ler capítulo 5

Faça os **exercícios** ao final do capítulo e postar as respostas no fórum