

# Introdução a OpenMP

Profa. Cristiana Bentes  
Departamento de Engenharia  
de Sistemas e Computação  
UERJ

# Roteiro

- Por que OpenMP?
- Modelo de Execução
- Diretivas OpenMP
- Sincronização
- OpenMP 4.0

# O que é OpenMP

- Open specifications for Multiprocessing
- API padrão para programação multithreading com memória compartilhada
- Especificação colaborativa: indústria, governo e academia
- <http://www.openmp.org>

# Por que OpenMP?

- Pthreads → requer controle explícito das threads

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#define SIZE 10      /* Size of matrices */
int N;               /* number of threads */
int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE]

void* mmult (void* slice)
{
    int s = (int)slice;
    int from = (s * SIZE)/N;
    int to = ((s+1) * SIZE)/N;
    int i,j,k;

    for (i=from; i<to; i++)
        for (j=0; j<SIZE; j++) {
            C[i][j]=0;
            for (k=0; k<SIZE; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    return 0;
}
```

```
int main(int argc, char *argv[])
{
    pthread_t *thread;
    int i;

    N=atoi(argv[1]);
    fill_matrix(A);
    fill_matrix(B);

    thread = malloc(N*sizeof(pthread_t));

    for (i=1; i<N; i++) {
        if (pthread_create (&thread[i], NULL, mmult, (void*)i) != 0 ) {
            perror("Can't create thread");
            exit(-1);
        }
    }
    mmult(0);

    for (i=1; i<N; i++) pthread_join (thread[i], NULL);

    return 0;
}
```

# Por que OpenMP?

- Pthreads → requer controle explícito das threads

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#define SIZE 10      /* Size of matrices */
int N;               /* number of threads */
int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE]

void* mmult (void* slice)
{
    int s = (int)slice;
    int from = (s * SIZE)/N;
    int to = ((s+1) * SIZE)/N;
    int i,j,k;

    for (i=from; i<to; i++)
        for (j=0; j<SIZE; j++) {
            C[i][j]=0;
            for (k=0; k<SIZE; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    return 0;
}
```

```
int main(int argc, char *argv[])
{
    pthread_t *thread;
    int i;

    N=atoi(argv[1]);
    fill_matrix(A);
    fill_matrix(B);

    thread = malloc(N*sizeof(pthread_t));

    for (i=1; i<N; i++) {
        if (pthread_create (&thread[i], NULL, mmult, (void*)i) != 0 ) {
            perror("Can't create thread");
            exit(-1);
        }
    }
    mmult(0);

    for (i=1; i<N; i++) pthread_join (thread[i], NULL);

    return 0;
}
```

Criar uma variável para cada thread

# Por que OpenMP?

- Pthreads → requer controle explícito das threads

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#define SIZE 10      /* Size of matrices */
int N;               /* number of threads */
int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE]

void* mmult (void* slice)
{
    int s = (int)slice;
    int from = (s * SIZE)/N;
    int to = ((s+1) * SIZE)/N;
    int i,j,k;

    for (i=from; i<to; i++)
        for (j=0; j<SIZE; j++) {
            C[i][j]=0;
            for (k=0; k<SIZE; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    return 0;
}
```

```
int main(int argc, char *argv[])
{
    pthread_t *thread;
    int i;

    N=atoi(argv[1]);
    fill_matrix(A);
    fill_matrix(B);

    thread = malloc(N*sizeof(pthread_t));

    for (i=1; i<N; i++) {
        if (pthread_create (&thread[i], NULL, mmult, (void*)i) != 0 ) {
            perror("Can't create thread");
            exit(-1);
        }
    }
    mmult(0);

    for (i=1; i<N; i++) pthread_join (thread[i], NULL);

    return 0;
}
```

Criar e esperar o término de cada thread

# Por que OpenMP?

- Pthreads → requer controle explícito das threads

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#define SIZE 10      /* Size of matrices */
int N;               /* number of threads */
int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE]

void* mmult (void* slice)
{
    int s = (int)slice;
    int from = (s * SIZE)/N;
    int to = ((s+1) * SIZE)/N;
    int i,j,k,
    for (i=from; i<to; i++)
        for (j=0; j<SIZE; j++) {
            C[i][j]=0;
            for (k=0; k<SIZE; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    return 0;
}
```

Definir o que cada thread faz

```
int main(int argc, char *argv[])
{
    pthread_t *thread;
    int i;

    N=atoi(argv[1]);
    fill_matrix(A);
    fill_matrix(B);

    thread = malloc(N*sizeof(pthread_t));

    for (i=1; i<N; i++) {
        if (pthread_create (&thread[i], NULL, mmult, (void*)i) != 0 ) {
            perror("Can't create thread");
            exit(-1);
        }
    }
    mmult(0);

    for (i=1; i<N; i++) pthread_join (thread[i], NULL);

    return 0;
}
```

# Por que OpenMP?

- Pthreads → requer controle explícito das threads

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#define SIZE 10      /* Size of matrices */
int N;               /* number of threads */
int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE]
```

```
void* mmult (void* slice)
{
    int s = (int)slice;
    int from = (s * SIZE)/N;
    int to = ((s+1) * SIZE)/N;
    int i,j,k;
```

```
    for (i=from; i<to; i++)
        for (j=0; j<SIZE; j++) {
            C[i][j]=0;
            for (k=0; k<SIZE; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    return 0;
```

```
int main(int argc, char *argv[])
{
    pthread_t *thread;
    int i;
```

```
    N=atoi(argv[1]);
    fill_matrix(A);
    fill_matrix(B);
```

```
    thread = malloc(N*sizeof(pthread_t));
```

```
    for (i=1; i<N; i++) {
        if (pthread_create (&thread[i], NULL, mmult, (void*)i) != 0 ) {
            perror("Can't create thread");
            exit( -1);
        }
    }
```

De fato o que se quer é paralelizar o loop para que ele seja executado em paralelo por N threads

```
    join (thread[i], NULL);
```

```
    return 0;
```



# Por que OpenMP?

- Paralelização automática pelo compilador?

# Por que OpenMP?

- Paralelização automática pelo compilador?
  - Só analisa loops com estrutura simples
  - Não paraleliza quando há chamadas de funções externas
  - Só paraleliza loops seguros → sem nenhuma dependência de dados

```
void add (int k, float *a, float *b)
{
    for (int i = 1; i < 10000; i++)
        a[i] = a[i+k] + b[i];
}
```

Dependendo do valor de  $k$  → as iterações podem gerar dependência (ex:  $k = -1$ )

# Sobre OpenMP

- Linguagens suportadas: Fortran e C/C++
- Mesmo código para implementação serial e paralela
- Modelo SPMD
- Implementação portátil
- Paralelização progressiva → começa pela computação mais pesada

# Sobre OpenMP

- Requer compilador → não é apenas uma biblioteca
- Compilador não detecta conflitos ou dependência de dados → é responsabilidade colocar diretivas e sincronização
- Compilador tem que saber o número de iterações
- Loop não pode conter exit ou break

# Sobre OpenMP

Diretivas  
Cláusulas

Regiões Paralelas  
Divisão de Trabalho  
Sincronização  
Escopo de Dados

Funções  
Biblioteca

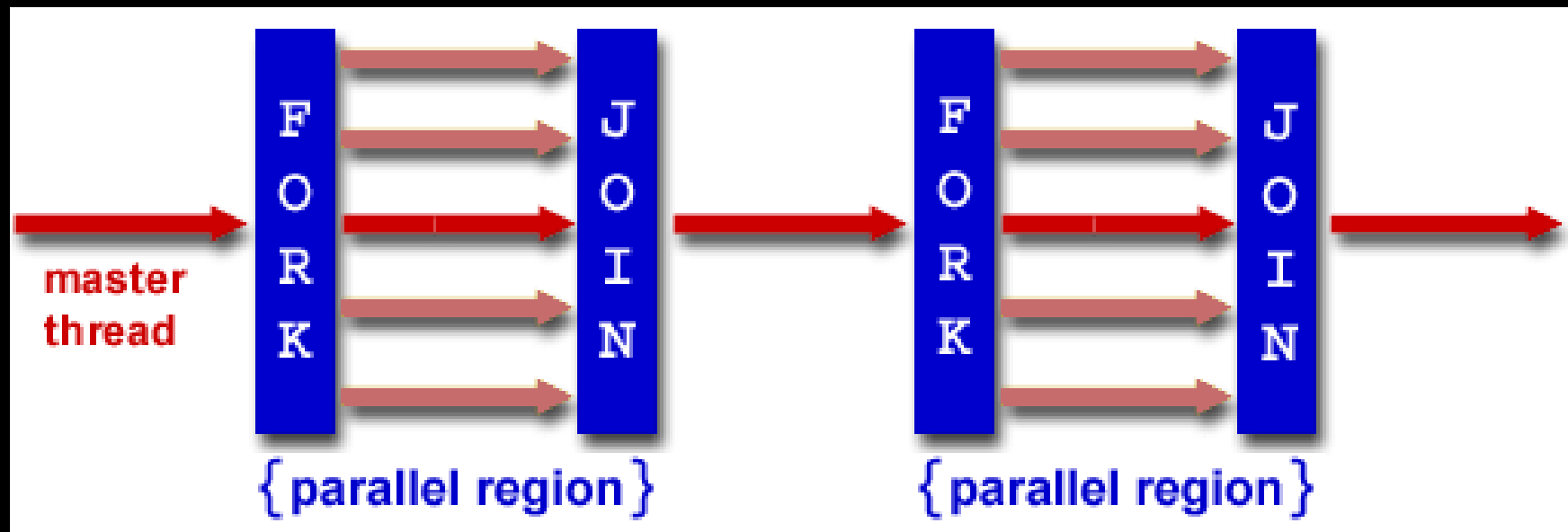
Set/Get:  
número de threads  
modo escalonar  
Timing

Variáveis  
Ambiente

Número de Threads  
Escalonamento  
Sincronização  
Escopo de Dados

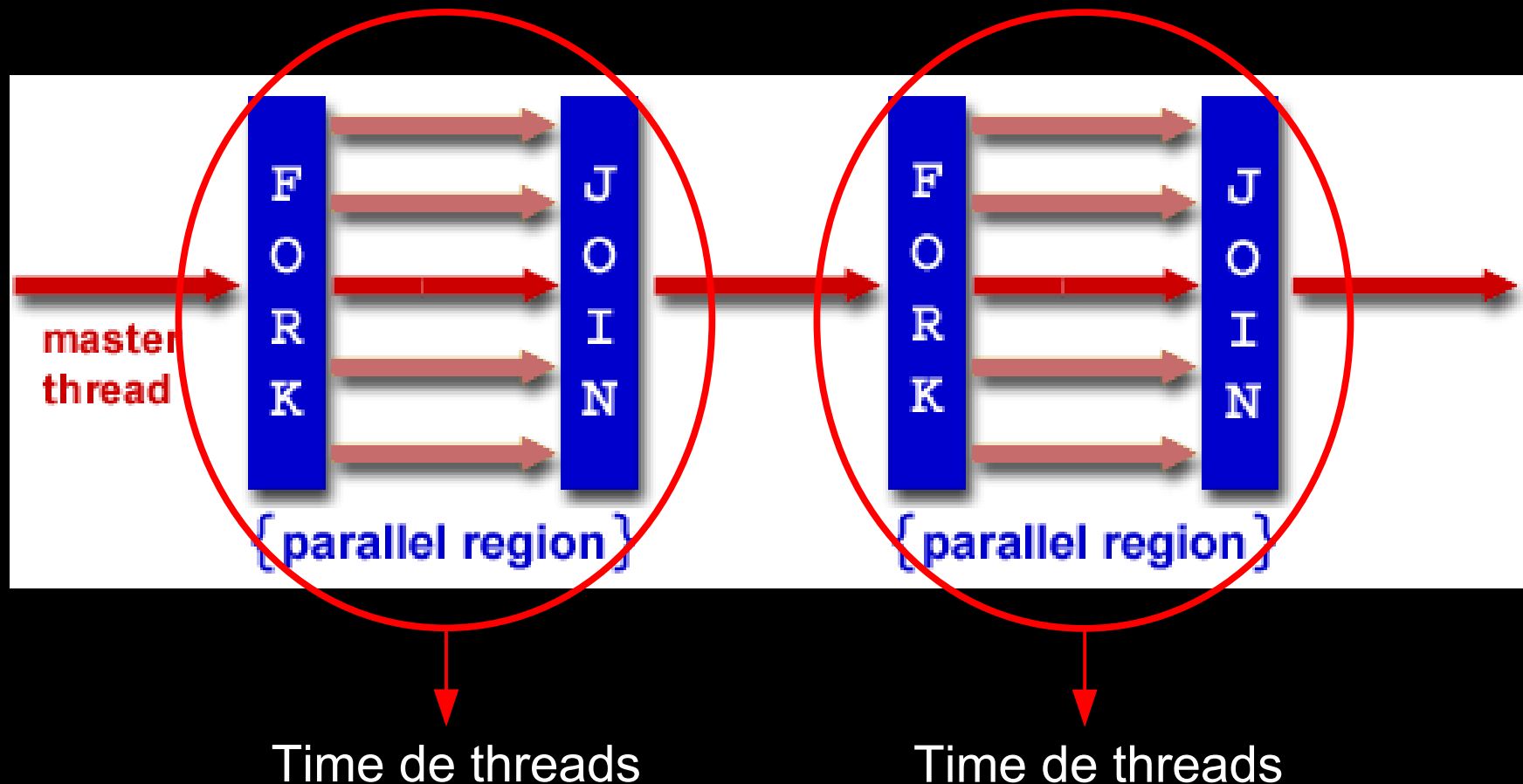
# Modelo de Execução

- Baseado em Fork-Join

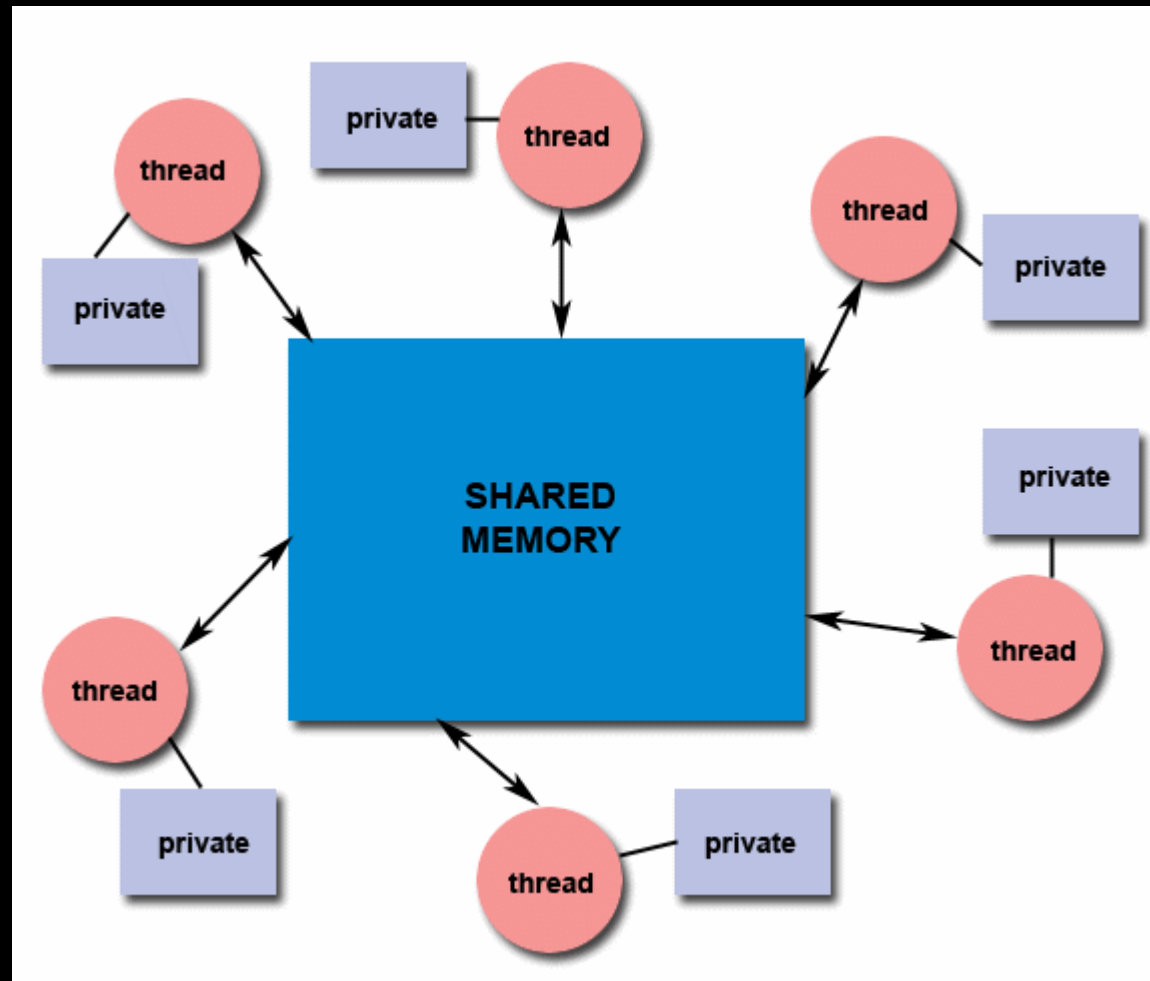


# Modelo de Execução

- Baseado em Fork-Join



# Modelo de Memória





# Como Compilar e Executar

- Compilar: A partir da versão 4.2 - GCC suporta OpenMP 3.0
  - `gcc -o omp_hello omp_hello.c -fopenmp`
- Executar: Setar variável de ambiente com número de threads
  - `export OMP_NUM_THREADS=2 (bash)`
  - `setenv OMP_NUM_THREADS 2 (tcsh)`
  - `./omp_helloc`

# Diretivas OpenMP

- Formato:  
    sentinela nome\_diretiva [clausulas]
- Sentinelas
  - C/C++: #pragma omp
  - Fortran: !\$OMP C\$OMP \*\$OMP
- Cláusulas podem aparecer em qualquer ordem
- Compilação condicional:
  - C/C++: #ifdef \_OPENMP

# Diretivas OpenMP

- Região Paralela

- `#pragma omp parallel [clausula] ...`
- Estrutura básica de paralelismo de OpenMP
- Bloco de código dentro de uma região paralela é executado por diferentes threads

```
f1();
```

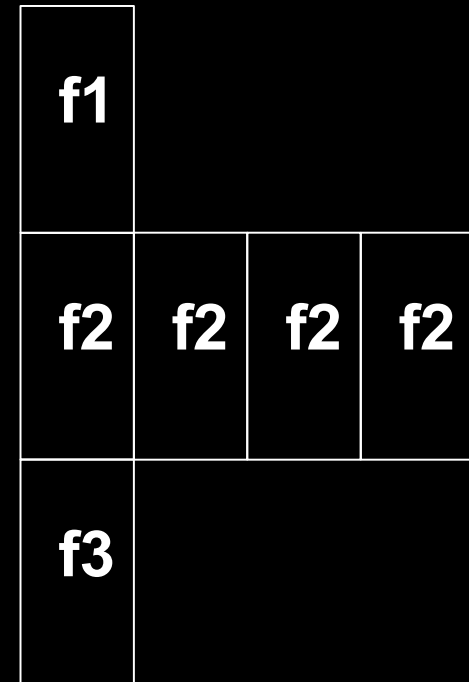
```
#pragma omp parallel
```

```
{
```

```
    f2();
```

```
}
```

```
f3();
```



# Diretivas OpenMP

- Região Paralela

- `#pragma omp parallel [clausula] ...`
- Estrutura básica de paralelismo de OpenMP
- Bloco de código dentro de uma região paralela é executado por diferentes threads

```
f1();
```

```
#pragma omp parallel
```

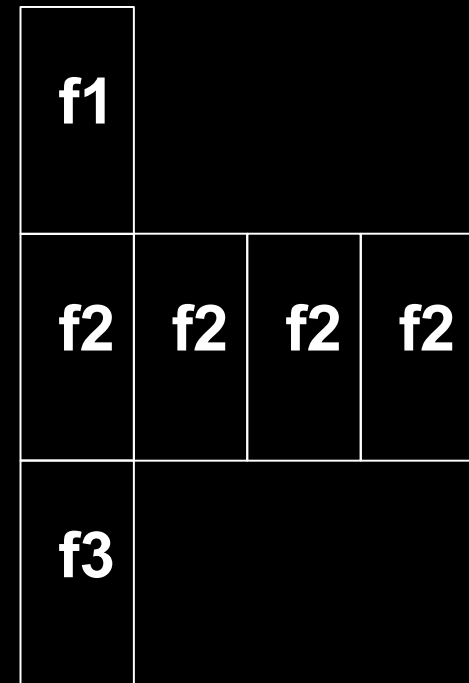
```
{
```

```
    f2();
```

```
}
```

```
f3();
```

→ Master espera



# Região Paralela

```
void main()  
{  
  
    int ID = 0;  
    printf(" hello world(%d) ", ID);  
  
}
```

# Região Paralela

```
void main()  
{  
  
    int ID = 0;  
    printf(" hello world(%d) ", ID);  
  
}
```

```
$ gcc -o hello hello_code.c  
$ ./hello
```



```
hello world(0)
```

# Região Paralela

```
void main()  
{  
#pragma omp parallel {  
    int ID = 0;  
    printf(" hello world(%d) ", ID);  
}  
}
```

# Região Paralela

```
void main()
{
    #pragma omp parallel {
        int ID = 0;
        printf(" hello world(%d) ", ID);
    }
}
```

```
$ export OMP_NUM_THREADS=3
$ gcc -o hello hello_code.c -fopenmp
$ ./hello
```



```
hello world(0)
hello world(0)
hello world(0)
```



# Identificação das Threads

- Chamadas a biblioteca
- Inserir

`#include <omp.h>`

- Chamadas mais comuns:

`void omp_set_num_threads(int number);`

`int omp_get_num_threads();`

`int omp_get_thread_num();`

# Região Paralela

```
#include <omp.h>
void main()
{
    #pragma omp parallel {
        int ID = omp_get_thread_num();
        printf(" hello world(%d) ", ID)
    }
}
```

```
$ export OMP_NUM_THREADS=3
$ gcc -o hello hello_code.c -fopenmp
$ ./hello
```




```
hello world(0)
hello world(2)
hello world(1)
```

# Região Paralela

```
#include <omp.h>
int main() {
#pragma omp parallel
{
    int np = omp_get_num_threads();
    int iam = omp_get_thread_num();
    printf("Hello from thread %d (total %d)\n",
        iam, np);
}
}
```

```
$ export OMP_NUM_THREADS=3
$ gcc -o hello hello_code.c -fopenmp
$ ./hello
```




```
Hello from thread 0 (total 3)
Hello from thread 1 (total 3)
Hello from thread 2 (total 3)
```

# Região Paralela

```
#include <omp.h>
int main() {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        int np = omp_get_num_threads();
        int iam = omp_get_thread_num();
        printf("Hello from thread %d (total %d)\n", iam, np);
    }
}
```

```
$ export OMP_NUM_THREADS=3
$ gcc -o hello hello_code.c -fopenmp
$ ./hello
```



Hello from thread 0 (total 4)  
Hello from thread 1 (total 4)  
Hello from thread 3 (total 4)  
Hello from thread 2 (total 4)

# Região Paralela

```
#include <omp.h>
int main() {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        int np = omp_get_num_threads();
        int iam = omp_get_thread_num();
        printf("Hello from thread %d (total %d)\n", iam, np);
    }
}
```

→ Só funciona fora da região paralela

```
$ export OMP_NUM_THREADS=3
$ gcc -o hello hello_code.c -fopenmp
$ ./hello
```

→ Hello from thread 0 (total 4)  
Hello from thread 1 (total 4)  
Hello from thread 3 (total 4)  
Hello from thread 2 (total 4)

# Compilação Condicional

```
#ifdef _OPENMP
#include <omp.h>
#endif
int main() {
#pragma omp parallel
{
#ifdef _OPENMP
    int np = omp_get_num_threads();
    int iam = omp_get_thread_num();
#endif
    printf("Hello from thread %d (total %d)\n",
        iam, np);
}
}
```


# Região Paralela - Cláusulas

- Especificam informação adicional nas diretivas
- Cláusulas:
  - num\_threads(integer-expression)
  - if([parallel :] scalar-expression)
  - private(list)
  - firstprivate(list)
  - shared(list)
  - default(shared | none)
  - reduction(reduction-identifier : list)
  - proc\_bind(master | close | spread)

# Cláusula Num\_threads

```
#include <omp.h>
int main() {
omp_set_num_threads(4);
#pragma omp parallel num_threads(4)
{
    int np = omp_get_num_threads();
    int iam = omp_get_thread_num();
    printf("Hello from thread %d (total %d)\n",
          iam, np);
}
}
```

```
$ export OMP_NUM_THREADS=3
$ gcc -o hello hello_code.c -fopenmp
$ ./hello
```



Hello from thread 0 (total 4)  
Hello from thread 1 (total 4)  
Hello from thread 3 (total 4)  
Hello from thread 2 (total 4)



# Cláusula If

- Cria uma condição para o paralelismo

```
#pragma omp parallel if (n>100000){
```

Só executa a região em paralelo  
se a condição for verdadeira

```
}
```

# Cláusulas Escopo de Dados

- Todas as variáveis declaradas antes da região paralela são consideradas shared por default
- Todas as variáveis declaradas dentro da região paralela são private

# Cláusulas Escopo de Dados

**shared (list):** lista de variáveis que todas as threads acessam a mesma cópia do dado criado na master thread

**private (list):** lista de variáveis que possuem uma cópia privada em cada thread, são descartada no final da região paralela

**firstprivate (list):** lista de variáveis privadas cuja inicialização é feita na thread master

**lastprivate(list):** lista de variáveis privadas cujo valor atribuído pela última thread é mantido depois da última iteração

# Cláusulas Escopo de Dados

```
#include <string>
#include <iostream>
#define N 10000
int main()
{
    float a[N], b[N];
    int i, n;
    float temp, csum;
    n = N;
    csum = 0.0;
    #pragma omp parallel
    {
        for (i=0; i<n; i++){
            temp = a[i] / b[i];
            csum += cos(temp);
        }
    }
}
```

# Cláusulas Escopo de Dados

```
#include <string>
#include <iostream>
#define N 10000
int main()
{
    float a[N], b[N];
    int i, n;
    float temp, csum;
    n = N;
    csum = 0.0;
    #pragma omp parallel
    {
        for (i=0; i<n; i++){
            temp = a[i] / b[i];
            csum += cos(temp);
        }
    }
}
```

Qual o escopo das variáveis  
nesta região paralela?

# Cláusulas Escopo de Dados

```
#include <string>
#include <iostream>
#define N 10000
int main()
{
    float a[N], b[N];
    int i, n;
    float temp, csum;
    n = N;
    csum = 0.0;
    #pragma omp parallel shared(a,b,n) private(temp,i) firstprivate(csum)
    {
        for (i=0; i<n; i++){
            temp = a[i] / b[i];
            csum += cos(temp);
        }
    }
}
```

# Cláusulas Escopo de Dados

```
#include <string>
#include <iostream>
int main()
{
    std::string a = "x", b = "y";
    int c = 3;
    #pragma omp parallel private(a,c) shared(b) num_threads(2)
    {
        a = "k";
        c = 7;
        std::cout << "A becomes (" << a << "), b is (" << b << ")\n";
    }
```

```
}
OpenMP_thread_fork(2);
{ // novo escopo
    std::string a; // Nova variável
    int c;         // Nova variável
    a = "k";
    c = 7;
    std::cout << "A becomes (" << a << "), b is (" << b << ")\n";
} // fim do escopo de variáveis locais
OpenMP_join();
```

# Cláusulas Escopo de Dados

```
#include <string>
#include <iostream>
int main()
{
    std::string a = "x", b = "y";
    int c = 3;
```

```
#pragma omp parallel
```

```
{
    a += "k";
    c += 7;
    std::cout << "A becomes (" << a << "), b is (" << b << ")\n";
}
}
```

Defina o escopo das variáveis  
Para que o resultado seja

A becomes (xk), b is (y)



# Cláusulas Escopo de Dados

```
#include <string>
#include <iostream>
int main()
{
    std::string a = "x", b = "y";
    int c = 3;

    #pragma omp parallel firstprivate(a,c) shared(b) num_threads(2)
    {
        a += "k";
        c += 7;
        std::cout << "A becomes (" << a << "), b is (" << b << ")\n";
    }
}
```

# Cláusulas Escopo de Dados

```
void loop()
{
    int i;
    #pragma omp lastprivate(i)
    for(i=0; i<Iteracoes; ++i)
        { ... }

    printf("%d\n", i); // mostra o número de iterações realizadas
}
```

# Cláusula Default

- Especifica o escopo default de variáveis que não foram listadas com nenhum escopo específico

```
#pragma omp parallel default(shared){
```

Variáveis não especificadas são  
shared

```
}
```

```
#pragma omp parallel default(none){
```

Variáveis não especificadas  
geram erro de compilação

```
}
```

# Cláusula Reduction

- Quando é necessário acumular (ou outra operação) em uma única variável o resultado de várias threads

```
#pragma omp parallel reduction(op:list){
```

Variáveis privadas são criadas para cada variável da lista

No final a variável recebe a combinação (operação) de todas as variáveis privadas em um único valor

```
}
```

op: +, \*, -, /, &, ^, |, &&, ||

list: variáveis shared

# Cláusula Reduction

```
int csum = 0;  
#pragma omp parallel private(i) shared(a,b) reduction(+:csum)  
{  
    for (i = 0; i < N; i++)  
        csum += a[i] + b[i];  
}  
printf("csum %d \n", csum);
```

# Work Sharing

- Quando um time encontra uma construção worksharing, o trabalho dentro da construção é dividido pelos membros do time
- Trabalho é executado cooperativamente ao invés de executado por todas as threads

```
#pragma omp for  
{  
....  
}
```

```
#pragma omp sections  
{  
....  
}
```

```
#pragma omp single  
{  
....  
}
```

# Work Sharing

- As construções devem estar dentro de uma região paralela
- Barreira no final da execução da construção (exceto se houver a cláusula `nowait`)
- Uma construção `worksharing` não cria novas threads

# Diretiva For

- Paralelismo de dados
- Divide iterações do loop

`#pragma omp for [clausula] ...`

`<original for-loop>`

- Clausulas:
  - private
  - firstprivate
  - lastprivate
  - reduction
  - nowait
  - schedule
  - ordered



# Diretiva For

```
#define N 1000
main () {
    int i;
    float a[N], b[N], c[N];
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    #pragma omp parallel shared(a,b,c) private(i)
    {
        #pragma omp for
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
    } /* end of parallel section */
}
```

# Exemplo Multiplicação de Matrizes

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE 1000      /* Max size of matrices */
int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
int main(int argc, char *argv[])
{
    int i,j,k,N;
    N=atoi(argv[1]);
    fill_matrix(A);
    fill_matrix(B);
    for ( i = 0; i < N; i++ )
    {
        for ( j = 0; j < N; j++ )
        {
            c[i][j] = 0.0;
            for ( k = 0; k < N; k++ )
            {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

# Exemplo Multiplicação de Matrizes

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#define SIZE 10      /* Size of matrices */
int N;               /* number of threads */
int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE]

void* mmult (void* slice)
{
    int s = (int)slice;
    int from = (s * SIZE)/N;
    int to = ((s+1) * SIZE)/N;
    int i,j,k;

    for (i=from; i<to; i++)
        for (j=0; j<SIZE; j++) {
            C[i][j]=0;
            for (k=0; k<SIZE; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    return 0;
}
```

```
int main(int argc, char *argv[])
{
    pthread_t *thread;
    int i;

    N=atoi(argv[1]);
    fill_matrix(A);
    fill_matrix(B);

    thread = malloc(N*sizeof(pthread_t));

    for (i=1; i<N; i++) {
        if (pthread_create (&thread[i], NULL, mmult, (void*)i) != 0 ) {
            perror("Can't create thread");
            exit(-1);
        }
    }
    mmult(0);

    for (i=1; i<N; i++) pthread_join (thread[i], NULL);

    return 0;
}
```

# Exemplo Multiplicação de Matrizes

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE 1000          /* Max size of matrices */
int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
int main(int argc, char *argv[])
{
    int i,j,k,N;
    N=atoi(argv[1]);
    fill_matrix(A);
    fill_matrix(B);
    #pragma omp parallel shared(A,B,C,N) private(i,j,k)
    #pragma omp for
    for ( i = 0; i < N; i++ )
    {
        for ( j = 0; j < N; j++ )
        {
            c[i][j] = 0.0;
            for ( k = 0; k < N; k++ )
            {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

# Clausula Nowait

- Evita barreira implícita ao final de uma construção worksharing

```
int factorial(int number)
{
    int fac = 1;
    #pragma omp parallel
    {
        int omp_priv = 1;
        #pragma omp for nowait
        for(int n=2; n<=number; ++n)
            omp_priv *= n;
        #pragma omp atomic
        fac *= omp_priv;
    }
    return fac;
}
```

# Clausula Schedule

- Determina como as iterações são executadas em paralelo

`#pragma omp for schedule(tipo,chunksize)`

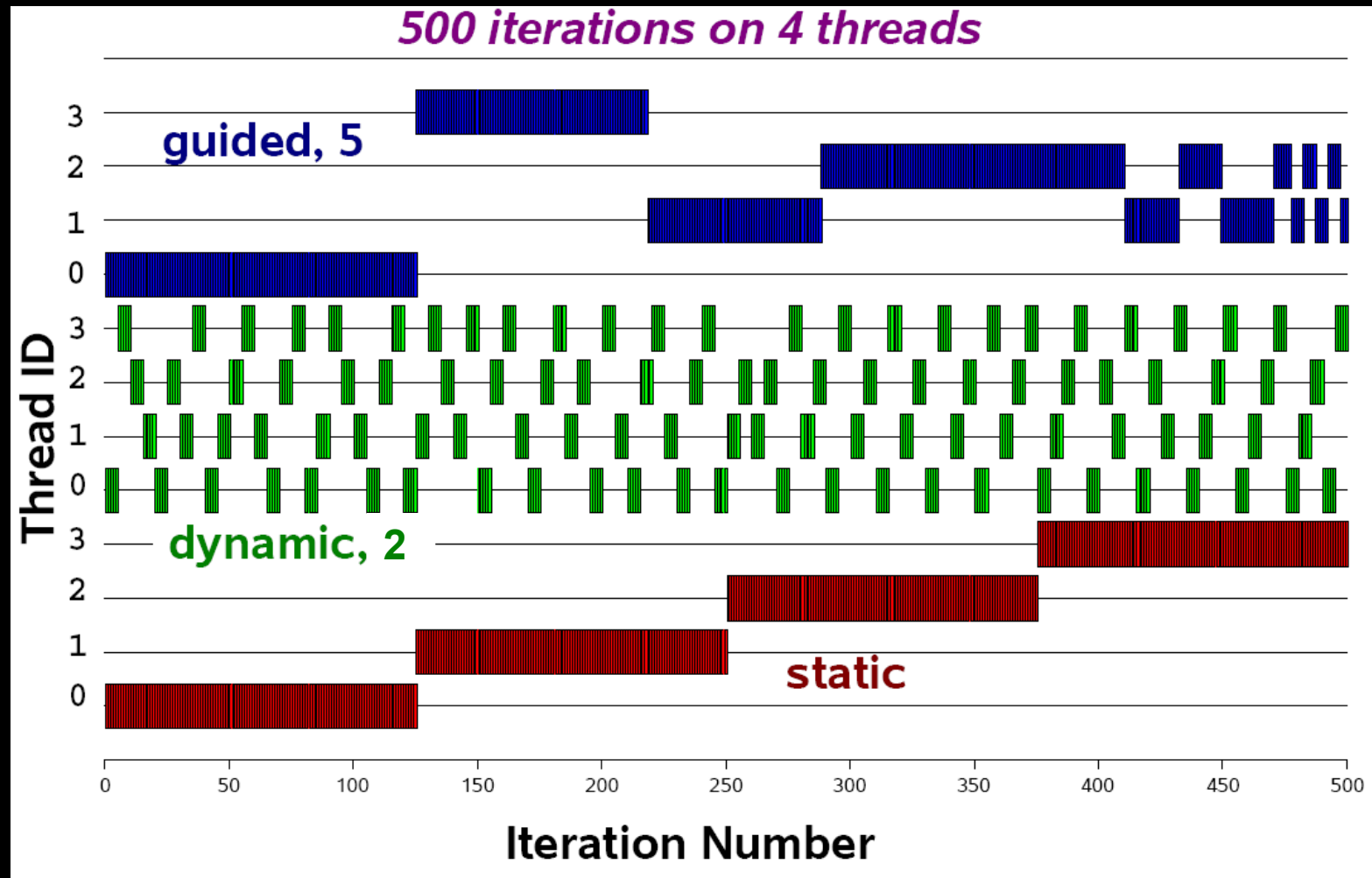
tipo = STATIC, DYNAMIC, GUIDED, RUNTIME

chunksize = tamanho das iterações

# Clausula Schedule

- Static (default) → as iterações são agrupadas em blocos (chunks) estaticamente atribuídos às threads
- Dynamic → divide as iterações em blocos do tamanho do chunksize. Os blocos são dinamicamente distribuídos pela threads, quando uma termina pega outro (chunksize default = 1)
- Guided → chunksize é proporcional a iterações restantes.

# Clausula Schedule





# Clausula Ordered

- Força que eventos aconteçam em determinada ordem

```
#pragma omp for ordered schedule(dynamic)
```

```
for(int n=0; n<100; ++n)
```

```
{
```

```
    files[n].compress();
```

```
#pragma omp ordered
```

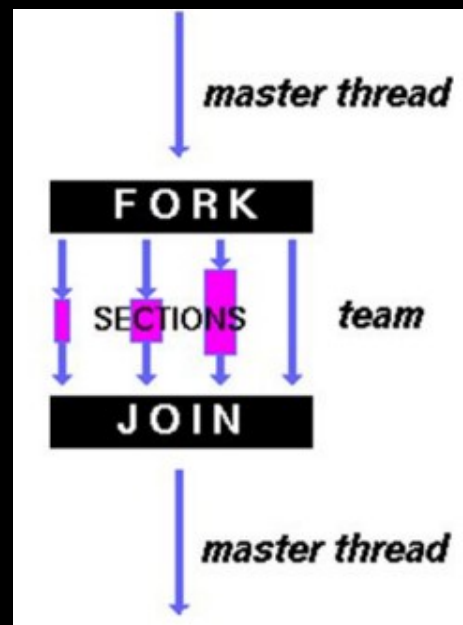
```
    send(files[n]);
```

```
}
```

100 arquivos são comprimidos. Alguns em paralelo. O envio dos arquivos é feito na ordem, mesmo que uma thread tenha que esperar que a anterior termine.

# Diretiva Sections

- Paralelismo Funcional
- Cada section é executada por uma thread
- $\# \text{ threads} > \# \text{ sections}$  – threads idle
- $\# \text{ threads} < \# \text{ sections}$  – sections serializadas

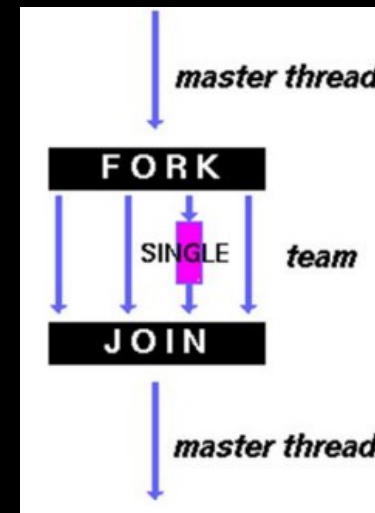


# Diretiva Sections

```
#pragma omp parallel default(none) \  
    shared(n,a,b,c,d) private(i)  
{  
    #pragma omp sections nowait  
    {  
        #pragma omp section  
        for (i=0; i<n-1; i++)  
            b[i] = (a[i] + a[i+1])/2;  
  
        #pragma omp section  
        for (i=0; i<n; i++)  
            d[i] = 1.0/c[i];  
  
    } /*-- End of sections --*/  
  
} /*-- End of parallel region --*/
```

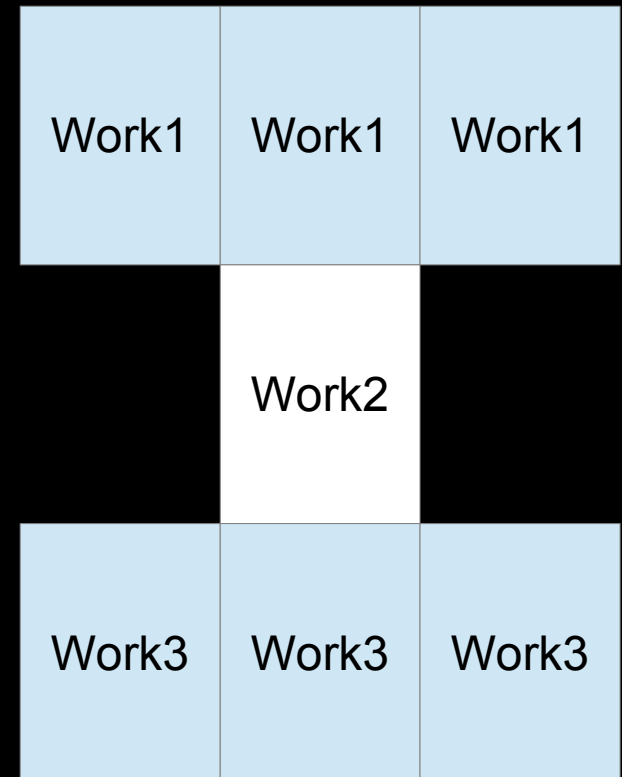
# Diretiva Single

- O bloco de código especificado pela diretiva é executado por apenas uma thread do time
- Não necessariamente a master thread
- Outras threads esperam numa barreira implícita ao final do single (exceto se clausal no wait for especificada)



# Diretiva Single

```
#pragma omp parallel
{
    Work1();
    #pragma omp single
    {
        Work2();
    }
    Work3();
}
```



# Combinando Diretivas

- Combinação de regiões paralelas com diretivas worksharing

```
#pragma omp parallel  
#pragma omp for  
for (...)
```



```
#pragma omp parallel for  
for (...)
```

Único loop

```
#pragma omp parallel  
#pragma omp sections  
{...}
```



```
#pragma omp parallel sections  
{...}
```

Somente sections paralelas

# Exercício Cálculo de Pi

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

int main(int argc, char *argv[])
{
    int n, i;
    double mypi, pi, h, sum, x, a;

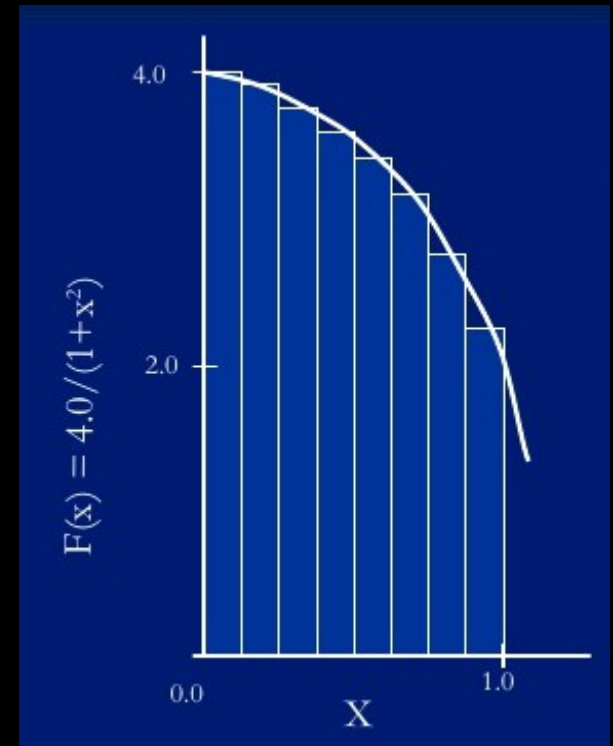
    n = atoi(argv[1]);

    h = 1.0 / (double) n;
    sum = 0.0;

    for (i = 1; i <= n; i++) {
        x = h * ((double) i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;

    printf("pi is approximately %.16f, mypi);
    return 0;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



# Exercício Cálculo de Pi

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

```
int main(int argc, char *argv[])
```

```
{
    int n, i;
    double pi, h, sum, x, a;
```

```
    n = atoi(argv[1]);
```

```
    h = 1.0 / (double) n;
    sum = 0.0;
```

```
#pragma omp parallel for private(i,x) shared(h) reduction(+: sum)
```

```
    for (i = 1; i <= n; i++) {
        x = h * ((double) i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
```

```
    pi = h * sum;
```

```
    printf("pi is approximately %.16f, pi);
```

```
    return 0;
```

```
}
```



# Sincronização

- Evitar condições de corrida
  - Barrier
  - Critical
  - Master
  - Atomic
  - Ordered

# Barrier

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

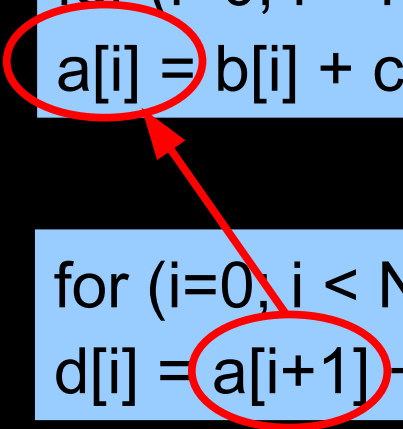
```
for (i=0; i < N; i++)  
d[i] = a[i+1] + b[i];
```

O que acontece se executarmos os dois trechos de código em paralelo?

# Barrier

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

```
for (i=0; i < N; i++)  
d[i] = a[i+1] + b[i];
```



O que acontece se executarmos os dois trechos de código em paralelo?

# Barrier

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

```
for (i=0; i < N; i++)  
d[i] = a[i+1] + b[i];
```

Barreira

O que acontece se executarmos os dois trechos de código em paralelo?

# Barrier

```
#pragma omp parallel {  
#pragma omp for  
for (i=0; i < N; i++)  
    a[i] = b[i] + c[i];  
#pragma omp barrier  
#pragma omp for  
for (i=0; i < N; i++)  
    d[i] = a[i+1] + b[i];  
}
```

Usada entre “fases” da computação

Sincronização com alto overhead

Desbalanceamento de carga → aumenta overhead da barreira

# Dados Compartilhados

```
#pragma omp parallel shared(a,sum) private(value,i)
#pragma omp for
for(i = 0; i < n; i++){
    value = f(a[i]);
    sum = sum + value;
}
```

O que acontece se executarmos este código em paralelo?

# Dados Compartilhados

```
#pragma omp parallel shared(a,sum) private(value,i)
#pragma omp for
for(i = 0; i < n; i++){
    value = f(a[i]);
    sum = sum + value;
}
```

value = 59

value = 78

sum = 0

# Dados Compartilhados

```
#pragma omp parallel shared(a,sum) private(value,i)
#pragma omp for
for(i = 0; i < n; i++){
    value = f(a[i]);
    sum = sum + value;
}
```

value = 59  
sum = 0

value = 78  
sum = 0

sum = 0



# Dados Compartilhados

```
#pragma omp parallel shared(a,sum) private(value,i)
#pragma omp for
for(i = 0; i < n; i++){
    value = f(a[i]);
    sum = sum + value;
}
```

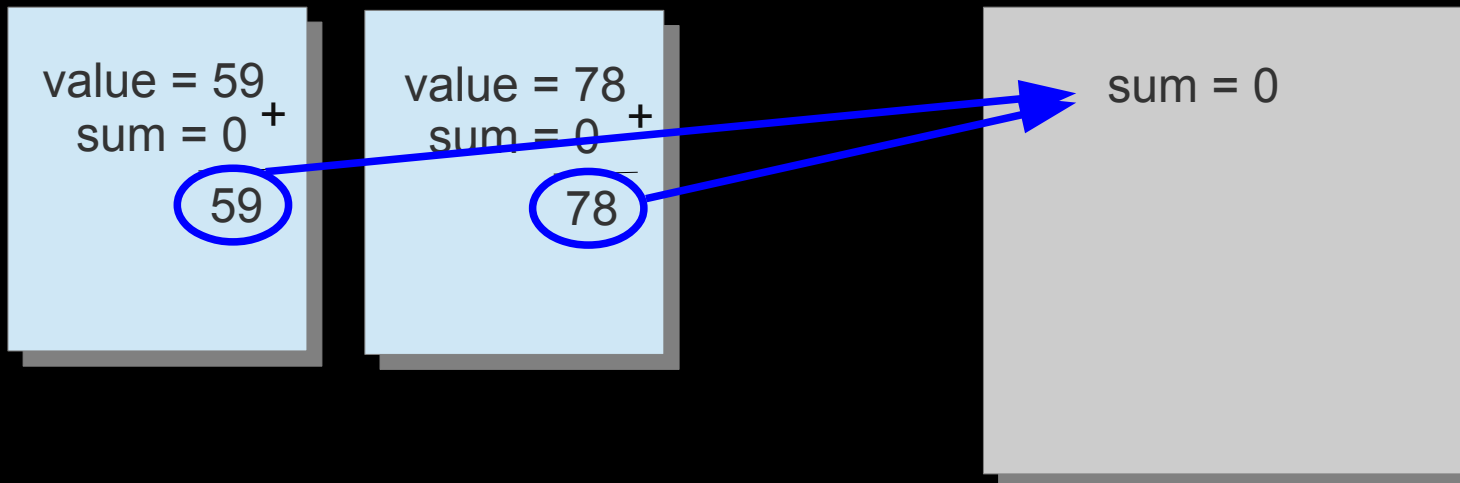
value = 59  
sum = 0<sup>+</sup>  
59

value = 78  
sum = 0<sup>+</sup>  
78

sum = 0

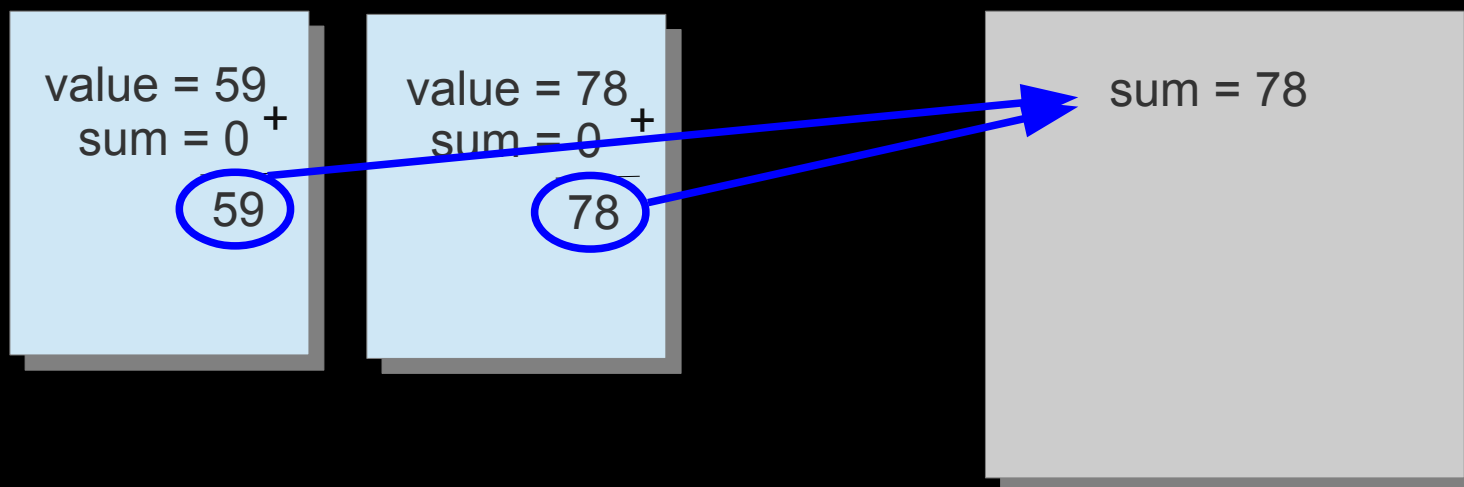
# Dados Compartilhados

```
#pragma omp parallel shared(a,sum) private(value,i)
#pragma omp for
for(i = 0; i < n; i++){
    value = f(a[i]);
    sum = sum + value;
}
```

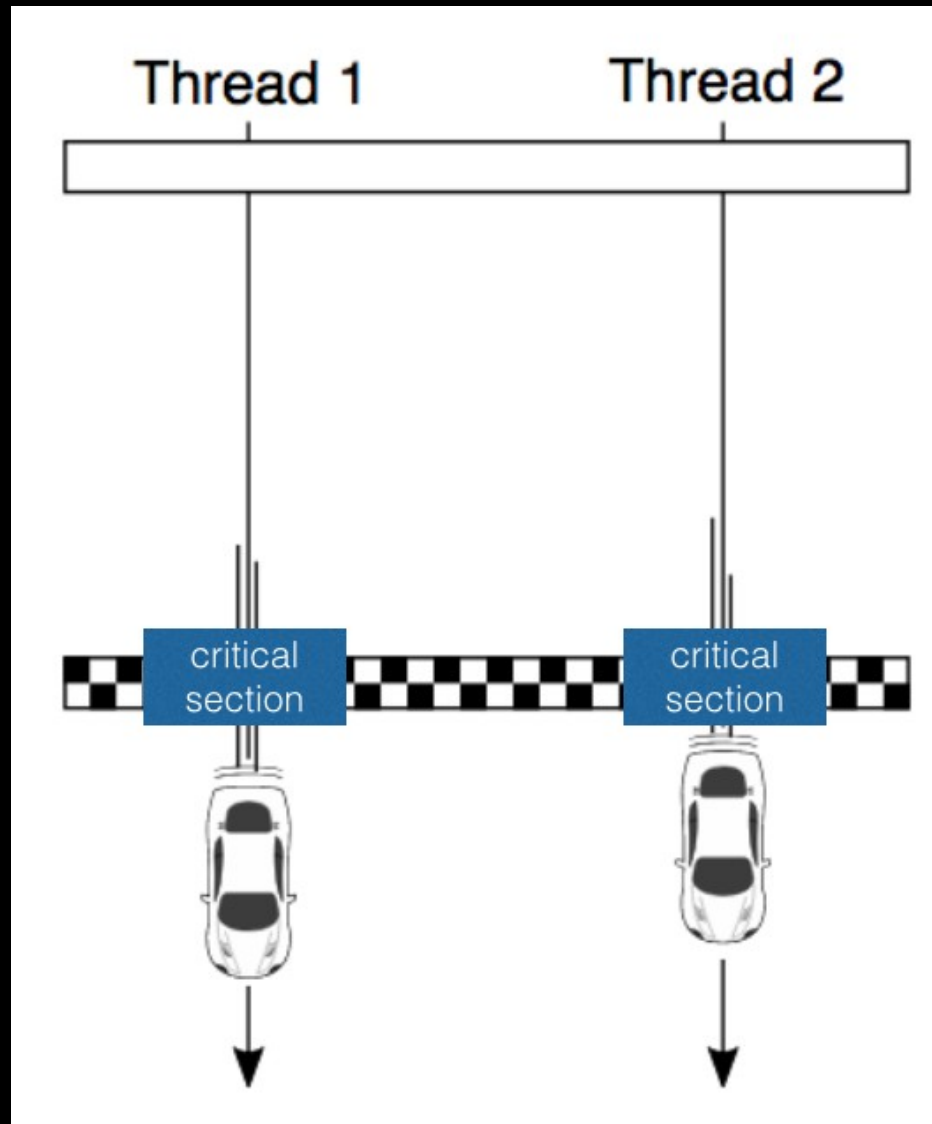


# Dados Compartilhados

```
#pragma omp parallel shared(a,sum) private(value,i)
#pragma omp for
for(i = 0; i < n; i++){
    value = f(a[i]);
    sum = sum + value;
}
```



# Condições de Corrida



# Critical

```
#pragma omp parallel shared(a,sum) private(value,i)
#pragma omp for
for(i = 0; i < n; i++){
    value = f(a[i]);
    sum = sum + value;
}
```

Proteger o acesso compartilhado  
Seção crítica

# Critical

```
#pragma omp parallel shared(a,sum) private(value,i)
#pragma omp for
for(i = 0; i < n; i++){
    value = f(a[i]);
    #pragma omp critical
    sum = sum + value;
}
```

# Critical

```
#pragma omp parallel shared(a,sum) private(value,i)
#pragma omp for
for(i = 0; i < n; i++){
    value = f(a[i]);
    #pragma omp critical
    sum = sum + value;
}
```

Desempenho?

# Critical

```
#pragma omp parallel shared(a,sum) private(value,i)
#pragma omp for {
    double temp = 0.0;
    for(i = 0; i < n; i++){
        value = f(a[i]);
        temp += value;
    }
    #pragma omp critical
        sum = sum + temp;
}
```

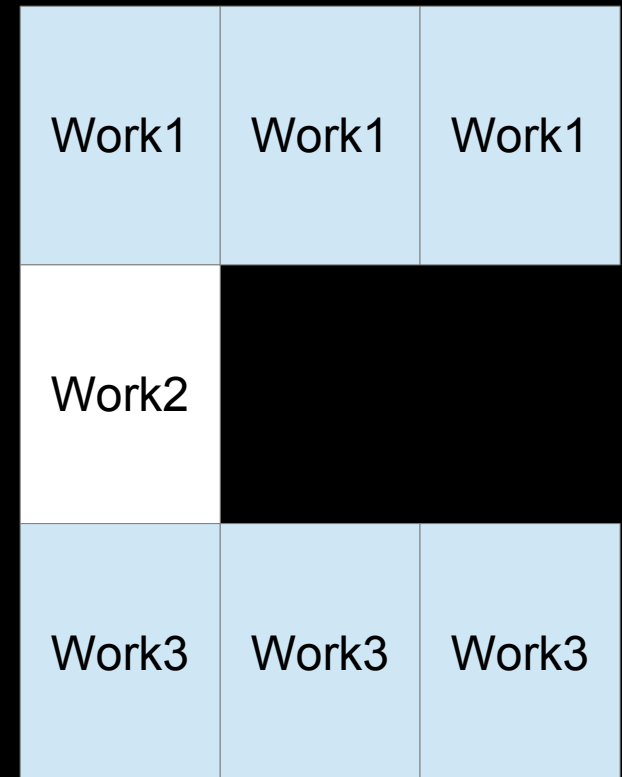


# Master

- Bloco de código executado somente pela master thread
- Diferença com single → não há barreira implícita com outras threads

# Diretiva Master

```
#pragma omp parallel
{
    Work1();
    #pragma omp master
    {
        Work2();
    }
    Work3();
}
```



# Atomic

#pragma omp atomic

i++;

- Somente para algumas expressões
  - x = expr
  - x++
  - ++x
  - x--
  - --x

# Exercício Produto Escalar

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

int main(int argc, char* argv[]){
    double sum;
    double a[256], b[256];
    int n = 256, i;
    for (i=0; i<n; i++){
        a[i] = i * 0.5;
        b[i] = i * 2.0;
    }
    sum = 0;

    for (i=1; i<n; i++){
        sum = sum + a[i]*b[i];
    }

    printf ("sum = %f\n", sum);
} //main
```

# Exercício Produto Escalar

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

int main(int argc, char* argv[]){
    double sum;
    double a[256], b[256];
    int n = 256, i, aux;
    for (i=0; i<n; i++){
        a[i] = i * 0.5;
        b[i] = i * 2.0;
    }
    sum = 0;
    #pragma omp parallel shared(a,b,n,sum) private(i,aux) {
        aux = 0;
        #pragma omp for
        for (i=1; i<n; i++){
            aux = aux + a[i]*b[i];
        }
        #pragma omp critical
        sum = sum + aux
    }
    printf ("sum = %f\n", sum);
} //main
```

# Exercício Produto Escalar

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

int main(int argc, char* argv[]){
    double sum;
    double a[256], b[256];
    int n = 256, i;
    for (i=0; i<n; i++){
        a[i] = i * 0.5;
        b[i] = i * 2.0;
    }
    sum = 0;
    #pragma omp parallel for shared(a,b,n,sum) private(i) reduction(+;sum)
    for (i=1; i<n; i++){
        sum = sum + a[i]*b[i];
    }

    printf ("sum = %f\n", sum);
} //main
```

# Algumas Dicas

- Ao inserir pragmas em programa serial, faça de forma incremental
- Não paralelize loops pequenos
- Escolha os loops para paralelizar através de profiling do código
- Minimize o uso de barreiras

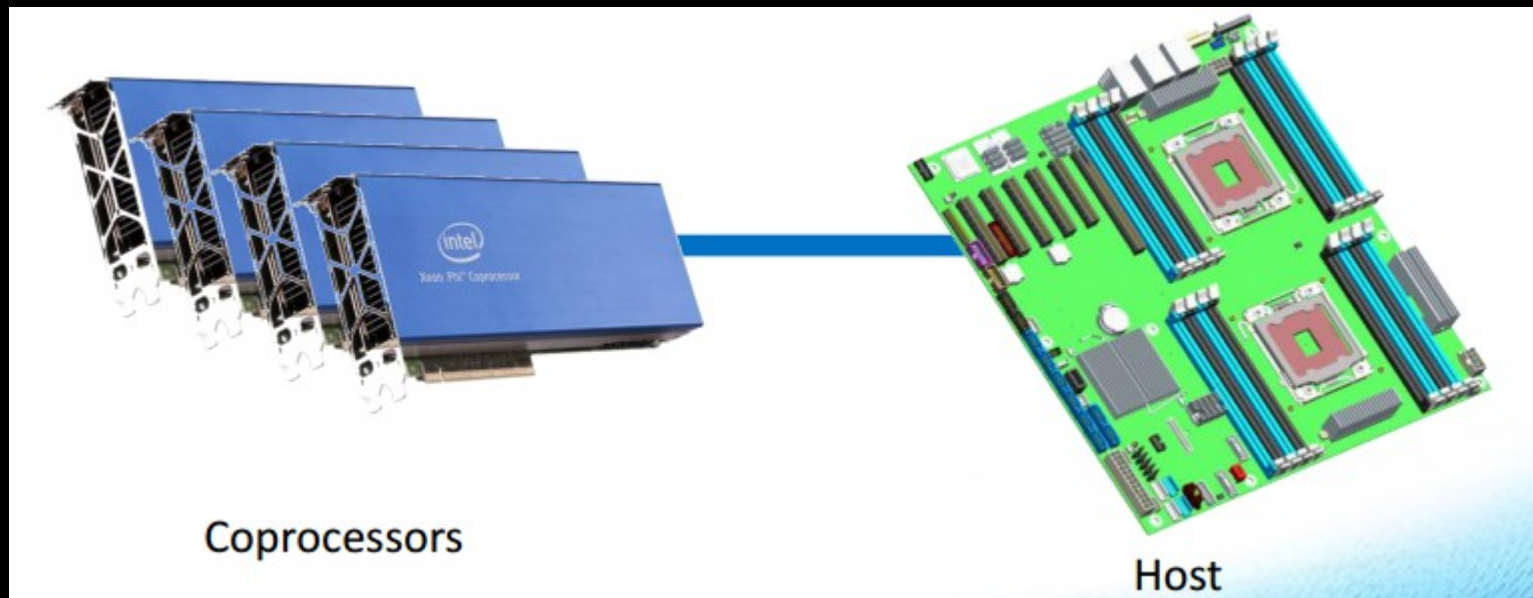
# OpenMP 4.0

- Novas Funcionalidades:
  - Suporte para aceleradores
  - Suporte para vetorização
  - Suporte para afinidade de threads



# Suporte para Aceleradores

- Modelo de hardware

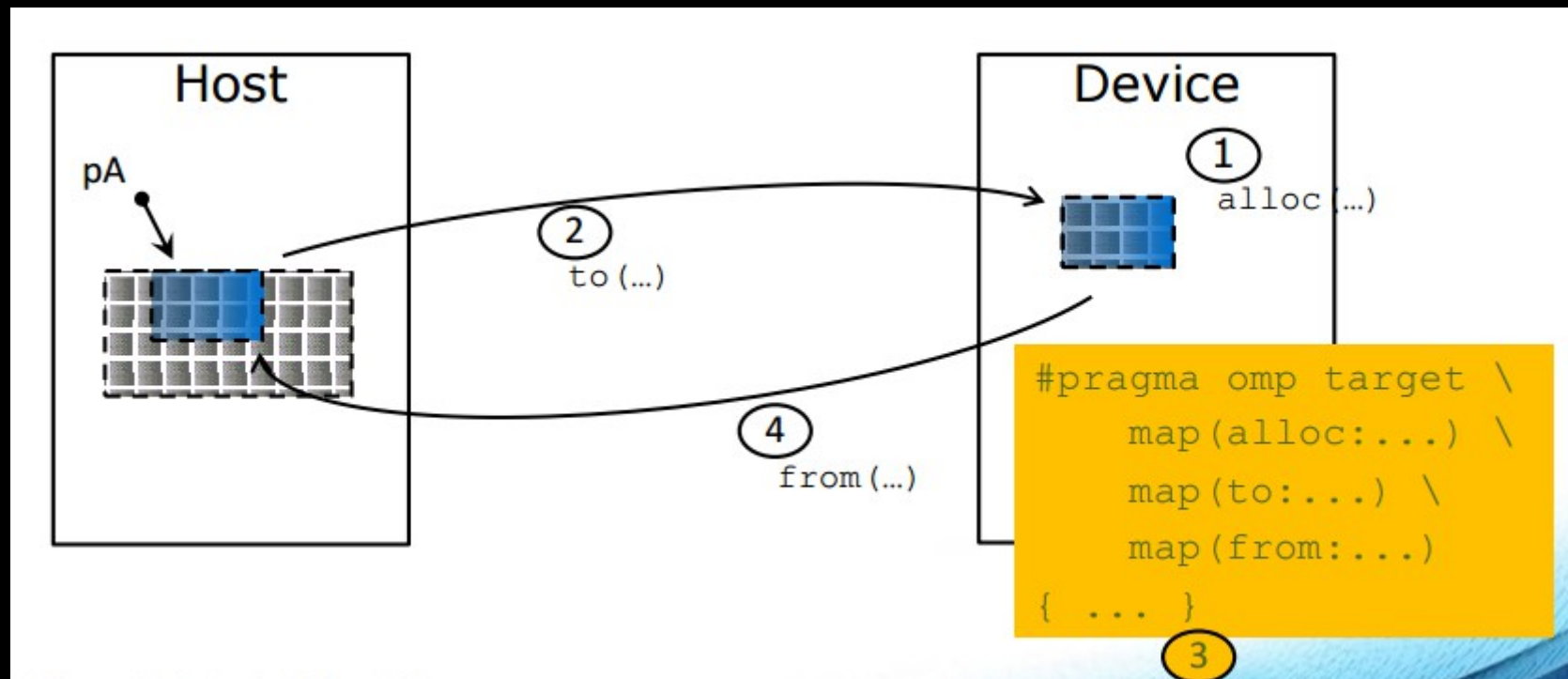


# Suporte para Aceleradores

- Construções target → transferem controle para o device
- Construções target data → criam o ambiente de dados no device
- **#pragma omp target [clausulas]...**
- Clausulas:
  - device (device\_num)
  - map ([alloc | to | from | tofrom:] list)
  - if

# Suporte para Aceleradores

- Ambiente de dados



# Suporte para Aceleradores

```
#pragma omp target map(to:b[0:count]) map(to:c,d) map(from:a[0:count])
{
#pragma omp parallel for
    for (i=0; i<count; i++) {
        a[i] = b[i] * c + d;
    }
}
```

host target host

```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
{
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
        tmp[i] = some_computation(input[i], i);

    do_some_other_stuff_on_host();

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
        res += final_computation(tmp[i], i)
}
```

host target host target host

# Suporte para Aceleradores

```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
{
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
        tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
        res += final_computation(input[i], tmp[i], i)
}
```

host

target

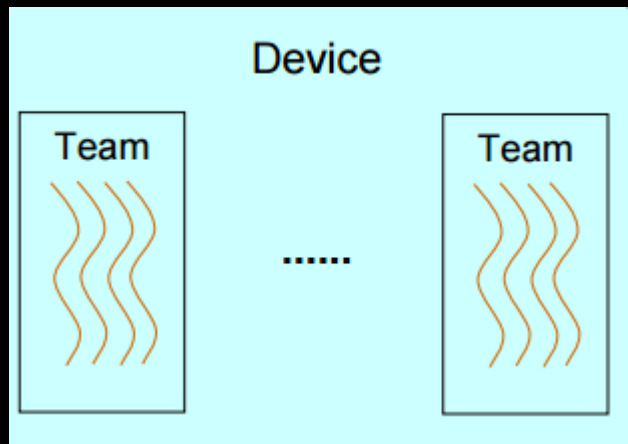
host

target

host

# Suporte para Aceleradores

- Granularidade fina:
  - Construção Teams: Criação de uma liga de times de threads
  - Construção Distribute: Distribui blocos de trabalho para os times



```
#pragma omp target teams
#pragma omp \
    distribute parallel for \
    reduction(+:sum)
for (i=0; i<N; i++)
    sum += B[i] * C[i];
```


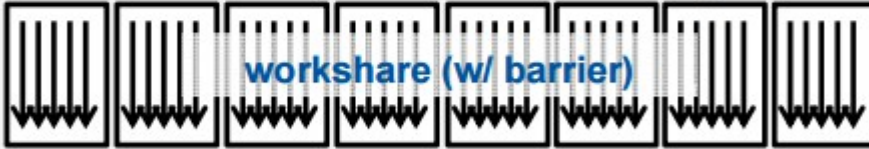
# Suporte para Aceleradores

```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{
    #pragma omp parallel for
    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
}
```

```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{
    #pragma omp target map(to:x[0:n], n, a) map(y[0:n])
    {
        #pragma omp parallel for
        for (int i = 0; i < n; ++i){
            y[i] = a*x[i] + y[i];
        }
    }
}
```



# Suporte para Aceleradores

```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{
    int num_blocks = Fb(n);
    int nthreads = Ft(n);
    #pragma omp target data map(to:x[0:n], n, a) map(y[0:n])
    #pragma omp teams num_teams(num_blocks) thread_limit(nthreads)
    {
        
        #pragma omp parallel for
        for (int i = 0; i < n; i += num_blocks){
            

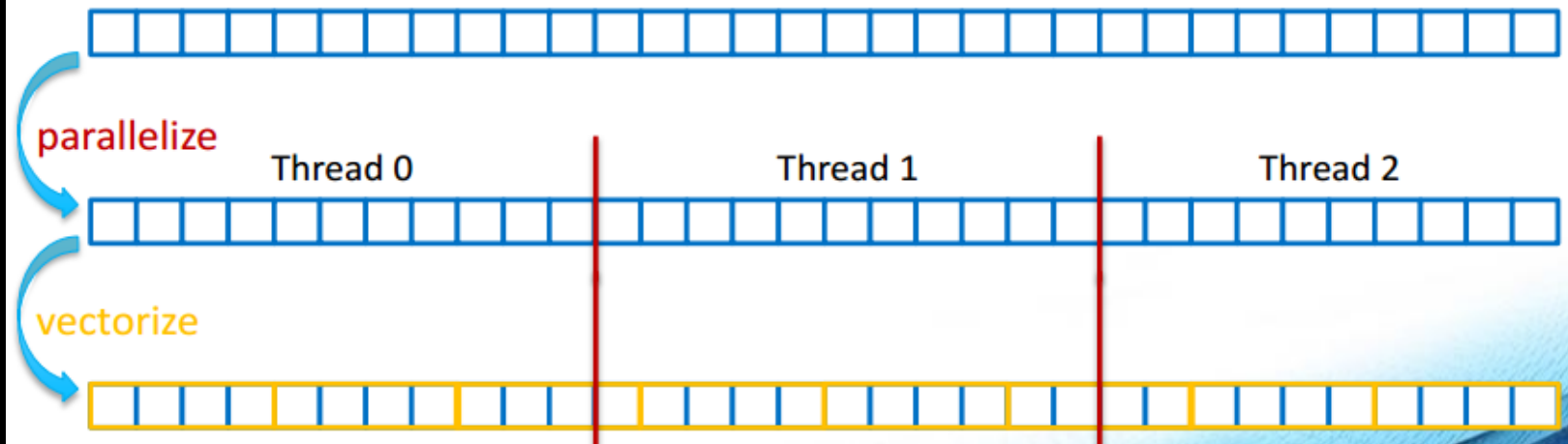
            for (int j = i; j < i + num_blocks; j++) {
                y[j] = a*x[j] + y[j];
            }
        }
    }
}
```



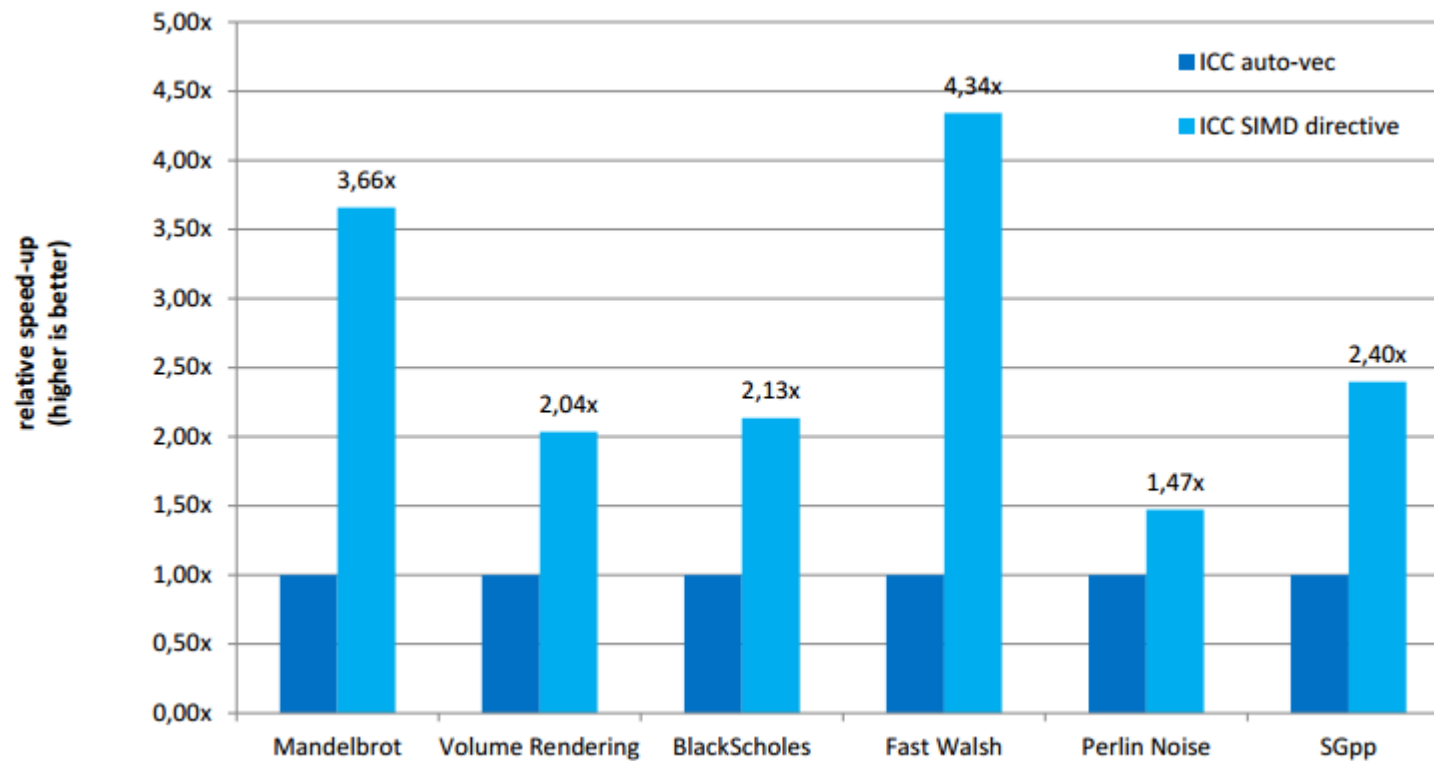
# Suporte para Vetorização

- Construção `simd` → indica que um loop deve ser vetorizado

```
void sprod(float *a, float *b, int n) {  
    float sum = 0.0f;  
    #pragma omp for simd reduction(+:sum)  
    for (int k=0; k<n; k++)  
        sum += a[k] * b[k];  
    return sum;  
}
```

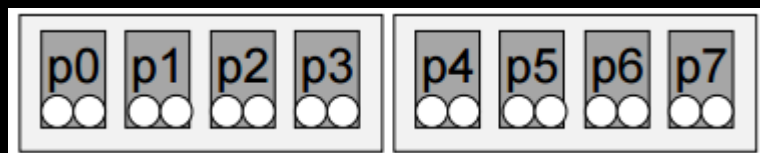


# Suporte para Vetorização



# Afinidade de Threads

- OpenMP cria o conceito de *places* para execução



8 places designados de p0 a p7

`OMP_PLACES="{0,1},{2,3},{4,5},{6,7},{8,9},{10,11},{12,13},{14,15}"`

# Afinidade de Threads

- Proc\_bind (master | close | spread)
  - Controla a afinidade das threads
  - Determina a distribuição das threads de um time pelos places disponíveis
  - Master → threads vão para o mesmo place da thread master
  - Close → threads vão para o mais próximo da thread pai
  - Spread → espalha as threads com distribuição round-robin

# Referências

- <http://openmp.org/wp/>
- Barbara Chapman, Gabriele Jost and Ruud van der Pas, Using OpenMP Portable Shared Memory Parallel Programming, MIT Press.
- <https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG>