

Unidade V:

Algoritmos Paralelos para Ordenação Interna



PUC Minas

Instituto de Ciências Exatas e Informática
Departamento de Ciência da Computação
Curso de Ciência da Computação

- Introdução
- Conceitos Básicos
- OpenMP
- Algoritmo Odd Even Paralelo
- Algoritmo Quicksort Paralelo

- **Introdução**
- Conceitos Básicos
- OpenMP
- Algoritmo Odd Even Paralelo
- Algoritmo Quicksort Paralelo

- Aumento do desempenho
 - Redução do tempo de execução
- Alta disponibilidade das arquiteturas paralelas
 - Realidade atual: processadores multicore

- Poucos desenvolvedores conhecem Computação Paralela
- Menos ainda são aqueles que aplicam a Computação Paralela

- Introdução
- **Conceitos Básicos**
- OpenMP
- Algoritmo Odd Even Paralelo
- Algoritmo Quicksort Paralelo

Conceito Básico: Processo

- Software que executa alguma ação
- Controlado pelo usuário, sistema operacional ou aplicativo
- Constituído por uma sequência de instruções, um conjunto de dados e um registro descritor

Exercício Resolvido (1)

- Descreva o comando top do Linux e, em seguida, mostre a execução desse comando

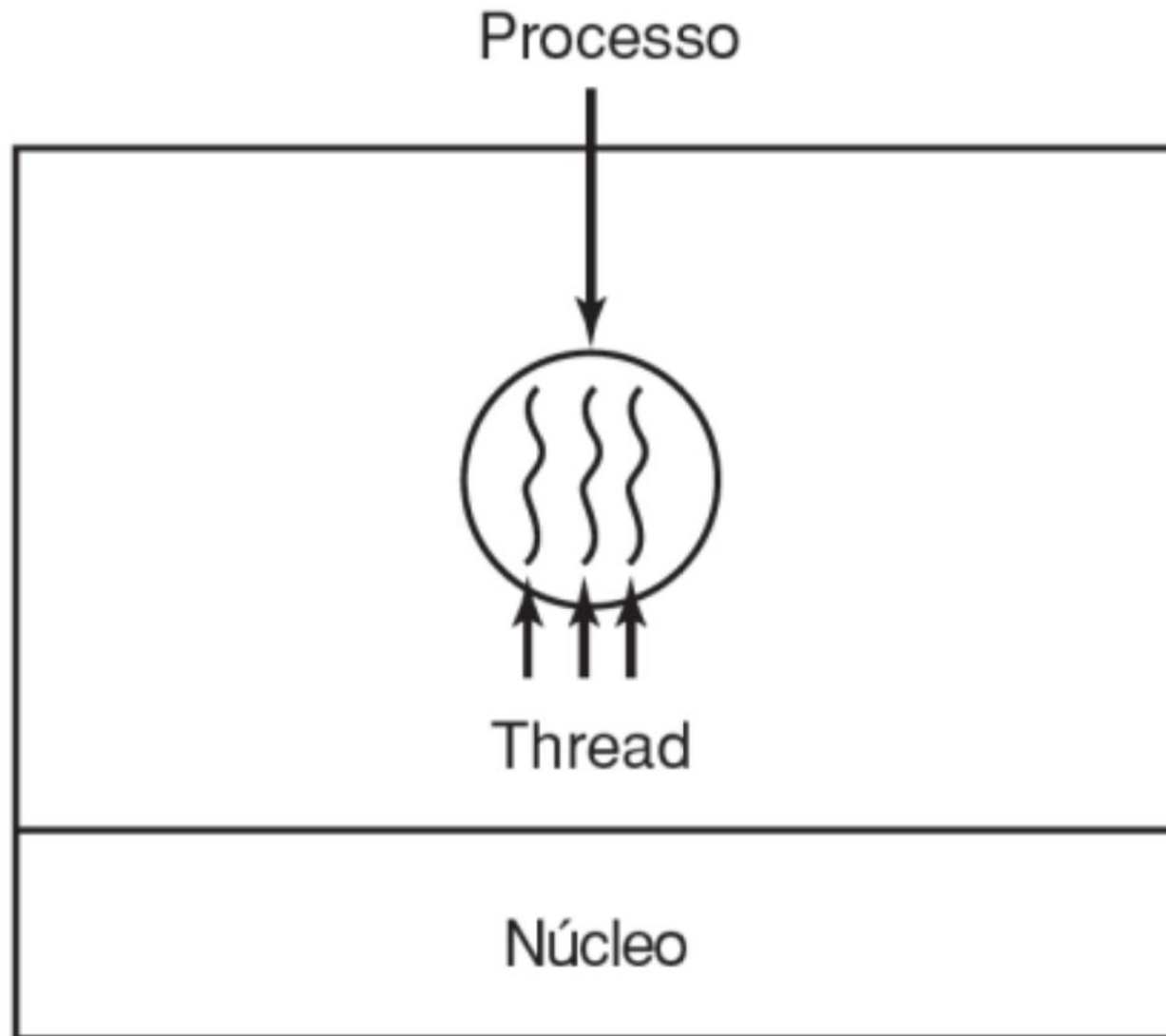
Exercício Resolvido (1)

- Descreva o comando top do Linux e, em seguida, mostre a execução desse comando

Arquivo Editar Ver Pesquisar Terminal Ajuda											
top - 14:43:20 up 8:54, 1 user, load average: 1,17, 1,52, 1,26											
Tarefas: 270 total, 2 em exec., 216 dormindo, 0 parado, 1 zumbi											
%CPU(s): 12,9 us, 2,8 sis, 0,0 ni, 84,1 oc, 0,1 ag, 0,0 ih, 0,0 is 0,0 tr											
KB mem : 8045512 total, 536172 livre, 3695980 usados, 3813360 buff/cache											
KB swap: 2097148 total, 2096880 livre, 268 usados, 3325736 mem dispon.											
PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TEMPO+	COMANDO
15739	maxm	20	0	793224	168748	112652	S	32,5	2,1	4:34.02	chrome
2539	maxm	20	0	730680	243796	188264	S	20,9	3,0	26:57.64	chrome
1308	maxm	20	0	1692496	124004	94260	R	8,9	1,5	14:15.61	Xorg
1860	maxm	20	0	3692792	281024	146992	S	0,3	3,5	13:38.45	gnome-shell
2117	maxm	20	0	1145412	23700	16212	S	0,3	0,3	1:37.60	core
2151	maxm	20	0	1139660	69316	52484	S	0,3	0,9	0:24.91	nautilus-deskto
2427	maxm	20	0	1423028	460144	312340	S	0,3	5,7	31:38.91	chrome
2542	maxm	20	0	568848	98148	64236	S	0,3	1,2	3:10.29	chrome
4452	maxm	20	0	818616	40952	30228	S	0,3	0,5	0:08.51	gnome-terminal-
16020	maxm	20	0	51460	4316	3516	R	0,3	0,1	0:00.10	top
1	root	20	0	225632	9540	6852	S	0,0	0,1	0:17.04	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.01	kthreadd
4	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	kworker/0:0H
6	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	mm_percpu_wq
7	root	20	0	0	0	0	S	0,0	0,0	0:00.23	ksoftirqd/0
8	root	20	0	0	0	0	I	0,0	0,0	0:49.41	rcu_sched
9	root	20	0	0	0	0	I	0,0	0,0	0:00.00	rcu_bh
10	root	rt	0	0	0	0	S	0,0	0,0	0:00.00	migration/0
11	root	rt	0	0	0	0	S	0,0	0,0	0:00.12	watchdog/0
12	root	20	0	0	0	0	S	0,0	0,0	0:00.00	cpuhp/0
13	root	20	0	0	0	0	S	0,0	0,0	0:00.00	cpuhp/1
14	root	rt	0	0	0	0	S	0,0	0,0	0:00.11	watchdog/1
15	root	rt	0	0	0	0	S	0,0	0,0	0:00.00	migration/1
16	root	20	0	0	0	0	S	0,0	0,0	0:00.24	ksoftirqd/1
18	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	kworker/1:0H
19	root	20	0	0	0	0	S	0,0	0,0	0:00.00	cpuhp/2
20	root	rt	0	0	0	0	S	0,0	0,0	0:00.13	watchdog/2

Conceito Básico: *Thread*

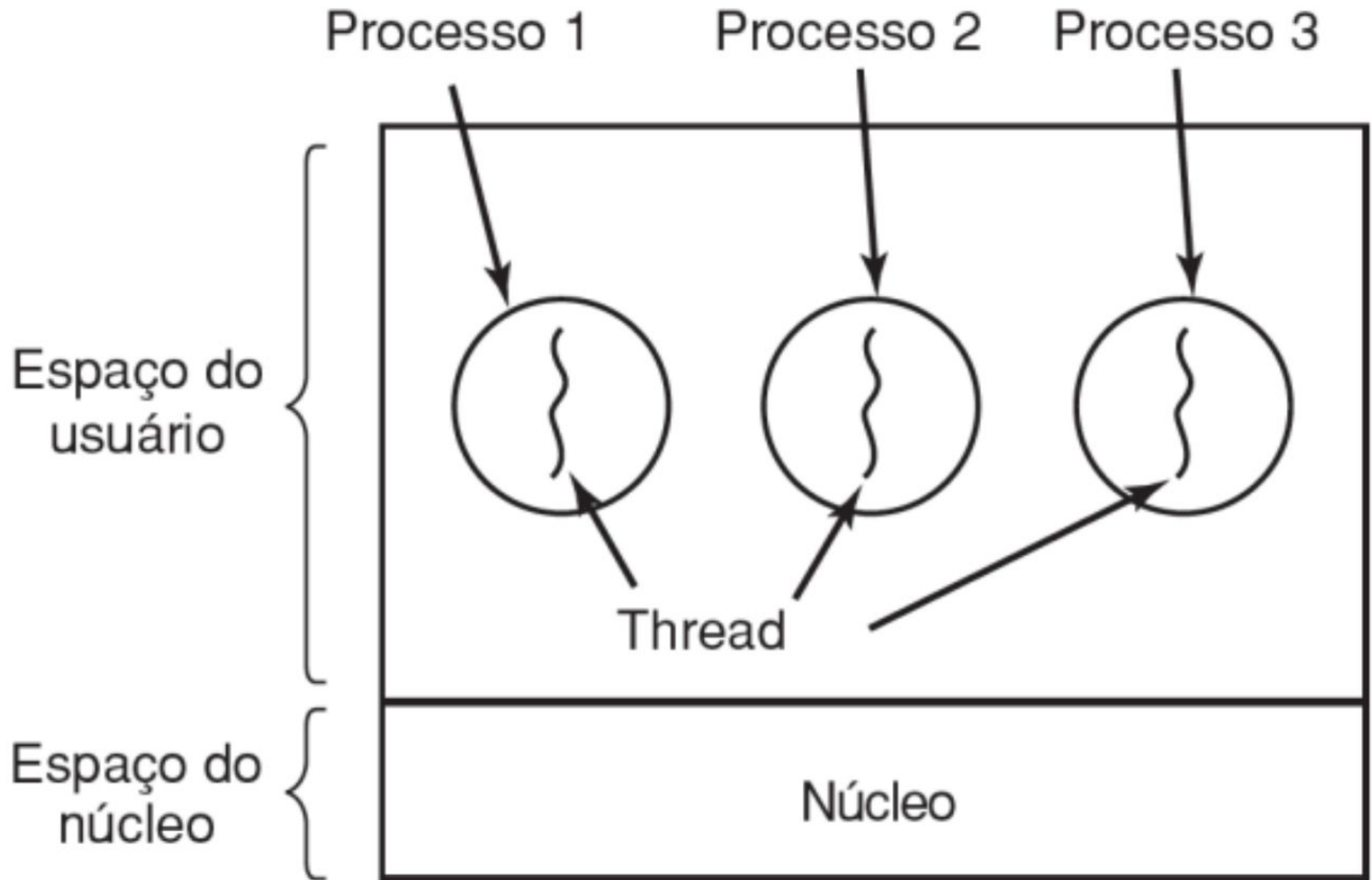
- Um fluxo de execução dentro de um processo



Conceito Básico: *Multithread*

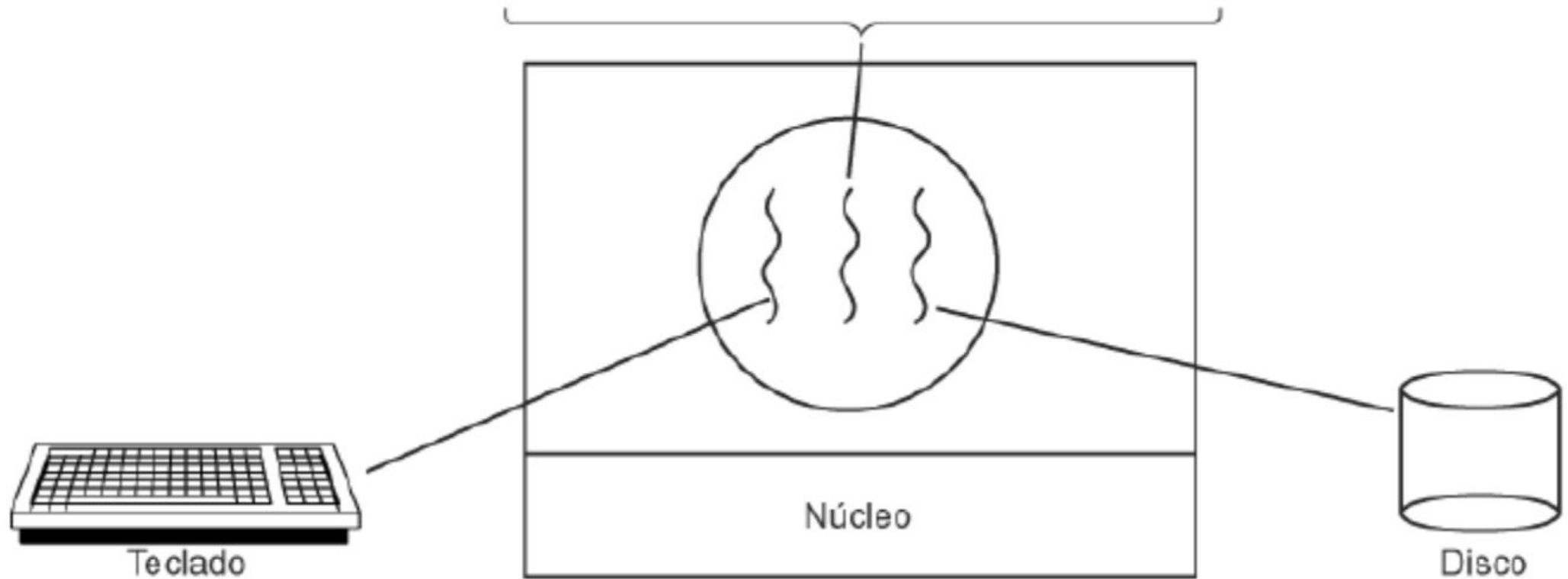
- Forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente
 - Uma *thread* compartilha recursos do processo com outras *threads*

Exemplo: Três Processos com uma *Thread* cada

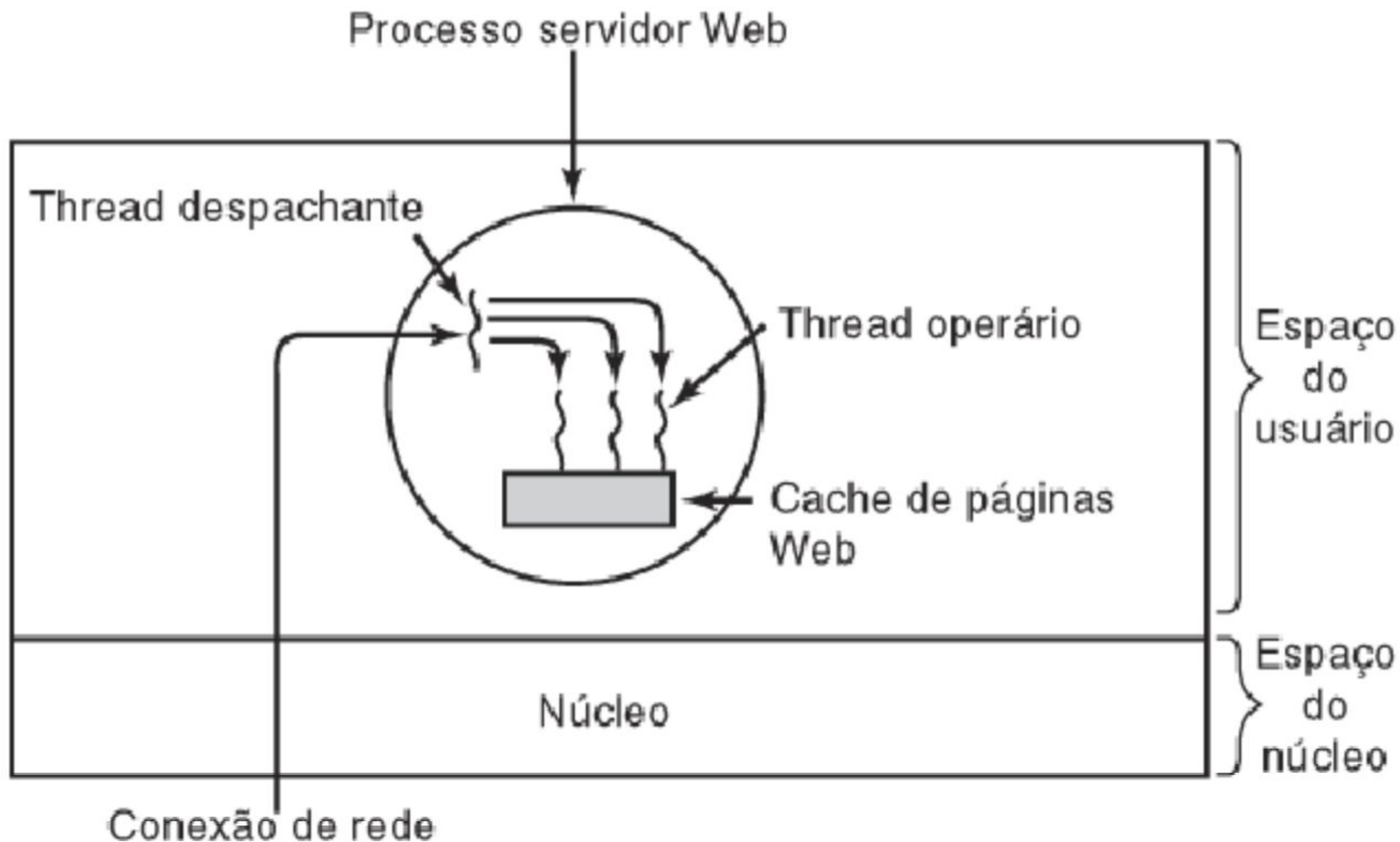


Exemplo: Processador com Três *Threads*

Four years and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal. Now we are	engaged in a great civil war testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war. We have come to dedicate a portion of	this field as a final resting place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this. But, in a larger sense, we cannot dedicate, we cannot	this field as a final resting place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this. But, in a larger sense, we cannot dedicate, we cannot	consecrate we must hallow this ground. The brave men, living and dead, who struggled here, have consecrated it, far above our poor power to add or deduct. The world will little note, nor long remember, what	we say here, but I can never forget what they did here. It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is	rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion. That we here highly
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



Exemplo: Servidor WEB com Três *Threads*



Vantagens das *Threads*

- Compartilham espaço de endereçamento e dados



Vantagens das *Threads*

- São criadas e destruídas rapidamente



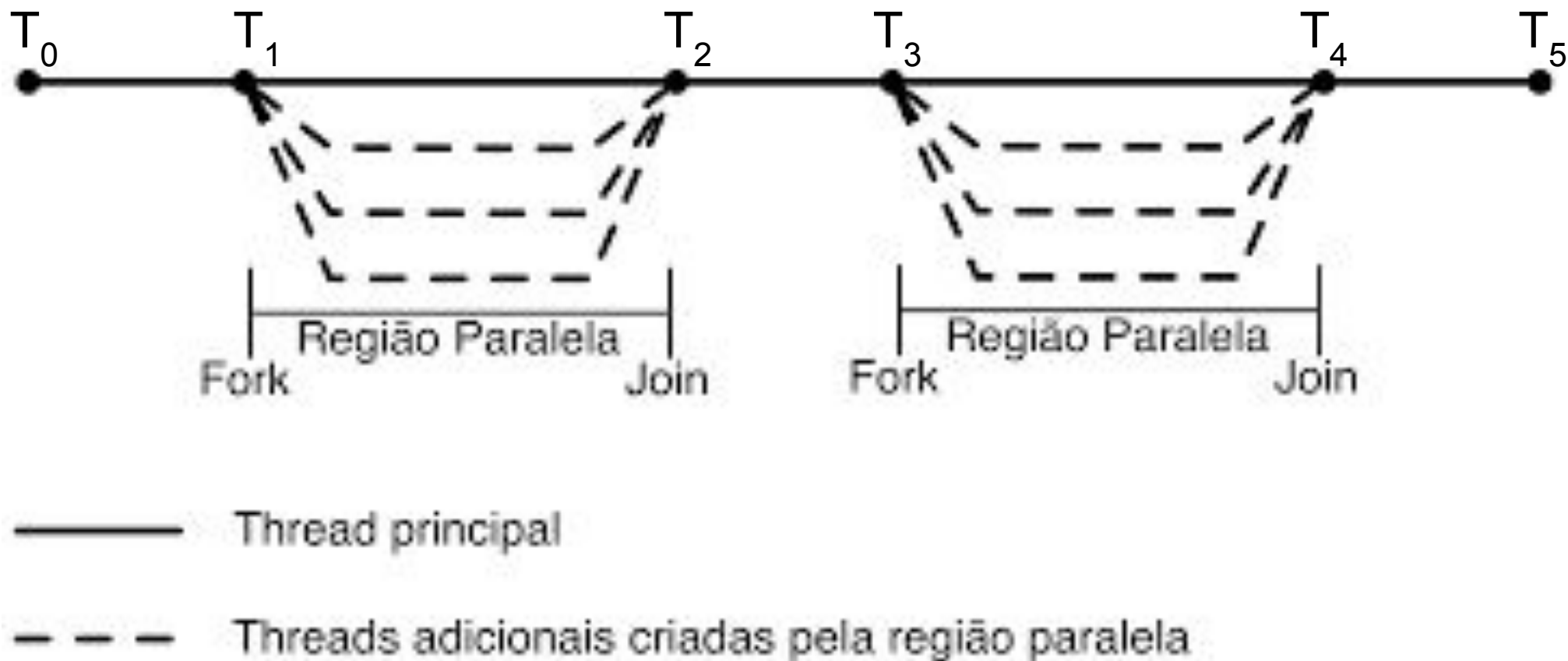
Vantagens das *Threads*

- Recurso poderoso para sistemas com múltiplas CPUs



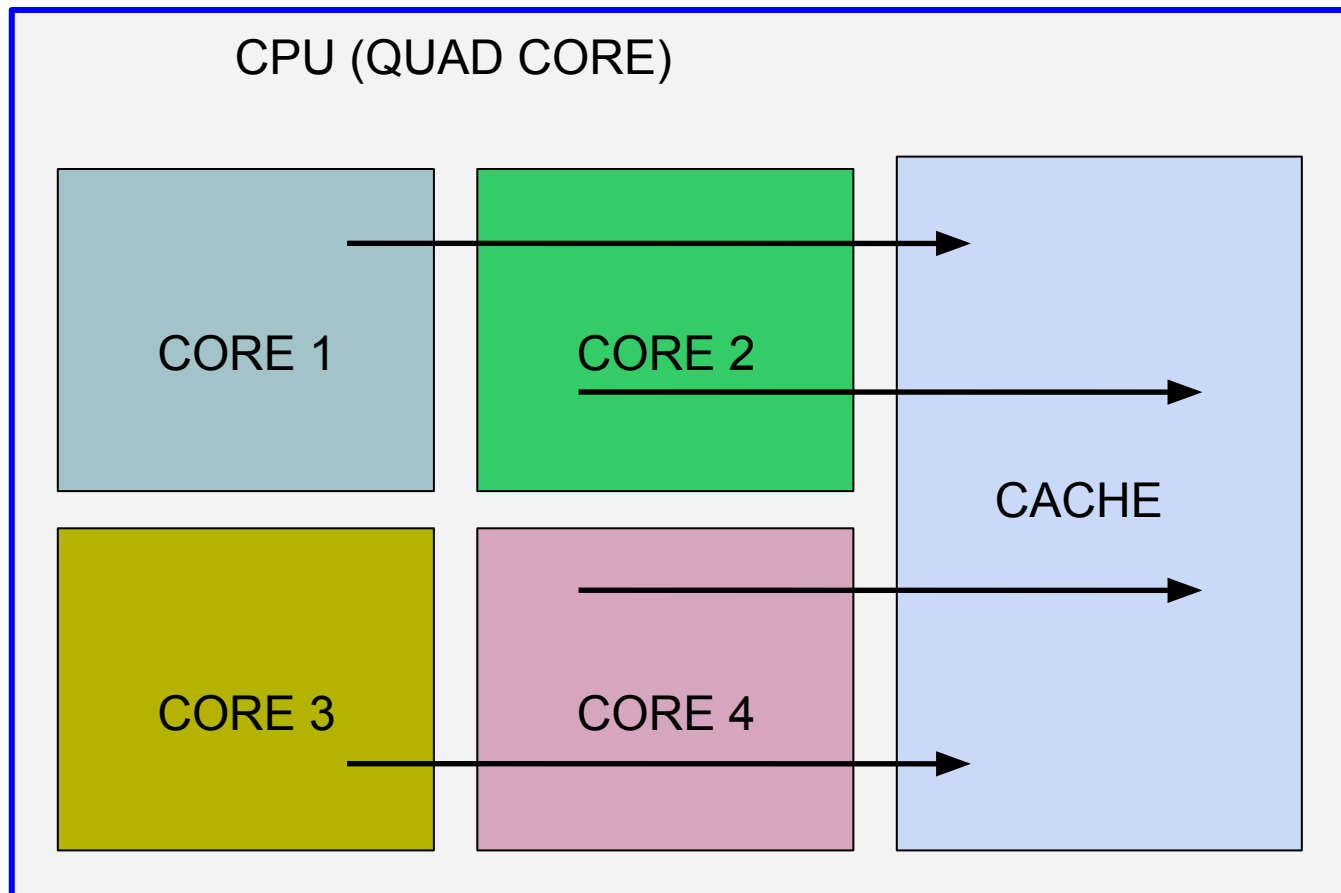
Vantagens das *Threads*

- Tornam possível manter a idéia de processos sequenciais e mesmo assim conseguem obter paralelismo



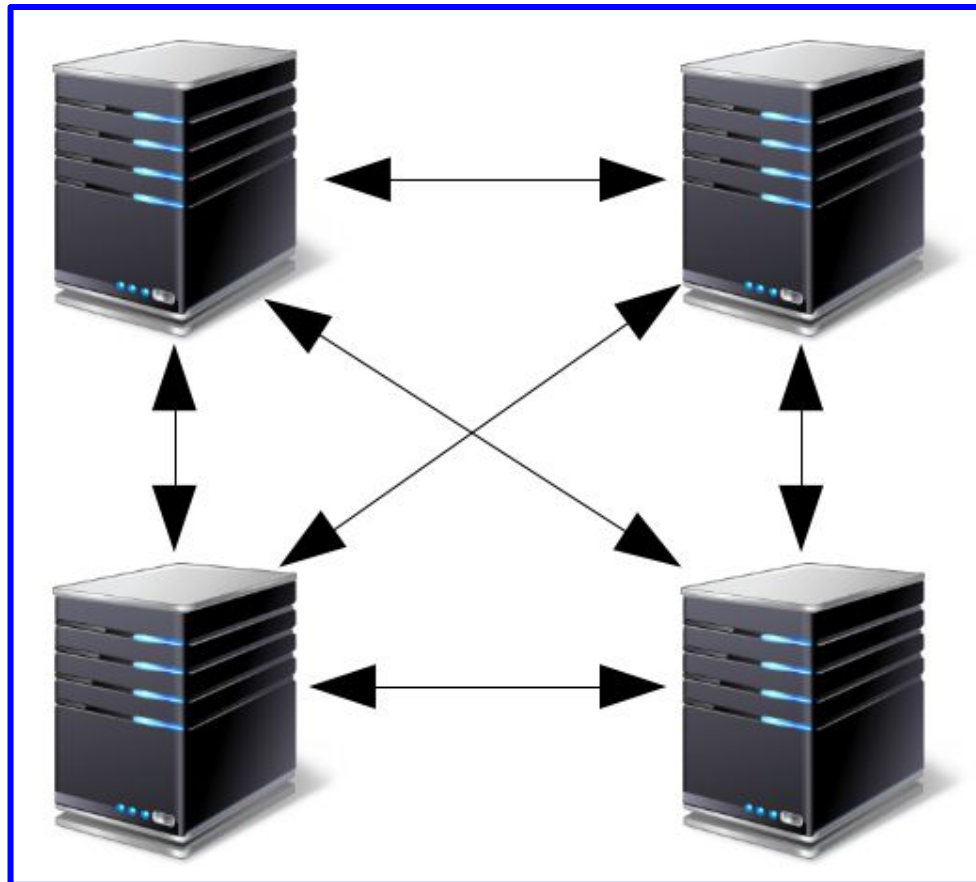
Conceito Básico: Memória Compartilhada

- Cada núcleo executa uma ou mais *threads* de um mesmo processo e elas se comunicam por variáveis compartilhadas

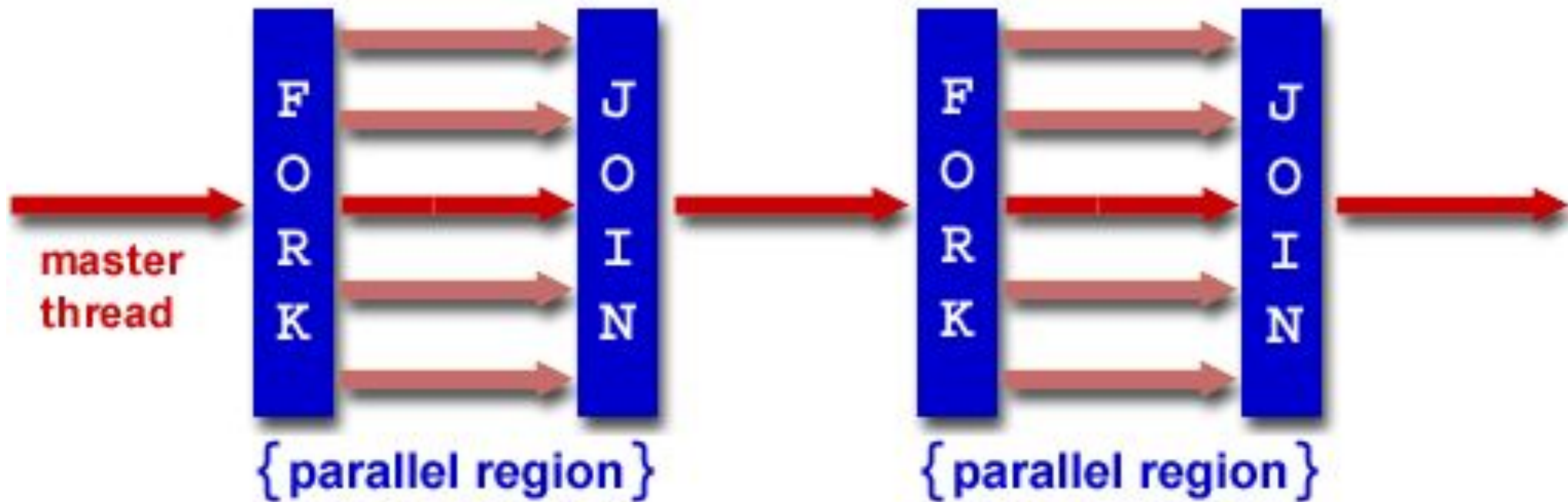


Conceito Básico: Memória Distribuída

- Cada máquina executa um processo da aplicação paralela e os processos se comunicam por troca de mensagens via rede



Conceito Básico: Modelo *Fork and Join*



Outros Conceitos Básicos Explicados à Frente

- Granularidade
- *Speedup*
- Eficiência

- Introdução
- Conceitos
- **OpenMP**
 - Sobre o OpenMP
 - Compilador gcc for Linux
 - Diretivas parallel e parallel for
 - Mais conceitos básicos
 - Diretivas sections, barrier, master e critical
 - Escopo das Variáveis
 - Cláusula Reduction
- Algoritmo
- Algoritmo

- Introdução
- Conceitos
- **OpenMP**
 - **Sobre o OpenMP**
 - Compilador gcc for Linux
 - Diretivas parallel e parallel for
 - Mais conceitos básicos
 - Diretivas sections, barrier, master e critical
 - Escopo das Variáveis
 - Cláusula Reduction
- Algoritmo
- Algoritmo

- API para a programação paralela de memória compartilhada
- Suportada pelas linguagens C, C++ e Fortran
- Possui um modelo *fork and join* para a execução paralela
- Constituída de:
 - diretivas de compilação
 - bibliotecas de função
 - variáveis de ambiente



- Introdução
- Conceitos
- **OpenMP**
 - Sobre o OpenMP
 - **Compilador gcc for Linux**
 - Diretivas parallel e parallel for
 - Mais conceitos básicos
 - Diretivas sections, barrier, master e critical
 - Escopo das Variáveis
 - Cláusula Reduction
- Algoritmo
- Algoritmo

Compilador GCC *for* Linux

```
#include<stdio.h>

int main(){
    int recebeNumero = 10;
    printf("recebeNumero = %d\n", recebeNumero);
    return 0;
}
```

01

Compilação no Linux com GCC
gcc 01-recebenum.c -o exec

Execução no Linux
./recebenum

- Introdução
- Conceitos
- **OpenMP**
 - Sobre o OpenMP
 - Compilador gcc for Linux
 - **Diretivas parallel e parallel for**
- Algoritmo
- Algoritmo
 - Mais conceitos básicos
 - Diretivas sections, barrier, master e critical
 - Escopo das Variáveis
 - Cláusula Reduction

Diretiva *omp parallel*

- Abre uma região paralela na qual todas as *threads* executam o código

Exemplo da Diretiva *omp parallel*

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

int main(){
    printf("Sequencial %d\n", omp_get_num_threads());
    #pragma omp parallel
    {
        printf("Paralela %d\n", omp_get_num_threads());
    }
    printf("Sequencial %d\n", omp_get_num_threads());
}
```

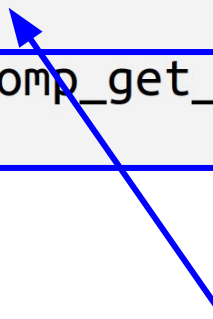
02

Compilação do GCC com OpenMP:
gcc -fopenmp 02-diretiva-op.c -o exec

Exemplo da Diretiva *omp parallel*

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

int main(){
    printf("Sequencial %d\n", omp_get_num_threads());
    #pragma omp parallel
    {
        printf("Paralela %d\n", omp_get_num_threads());
    }
    printf("Sequencial %d\n", omp_get_num_threads());
}
```



02

Região Paralela

Compilação do GCC com OpenMP:
gcc -fopenmp 02-diretiva-op.c -o exec

Exemplo da Diretiva *omp parallel*

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

int main(){
    printf("Sequencial %d\n", omp_get_num_threads());
    #pragma omp parallel
    {
        printf("Paralela %d\n", omp_get_num_threads());
    }
    printf("Sequencial %d\n", omp_get_num_threads());
}
```

02

TELA

Sequencial 1



Thread Master

Exemplo da Diretiva *omp parallel*

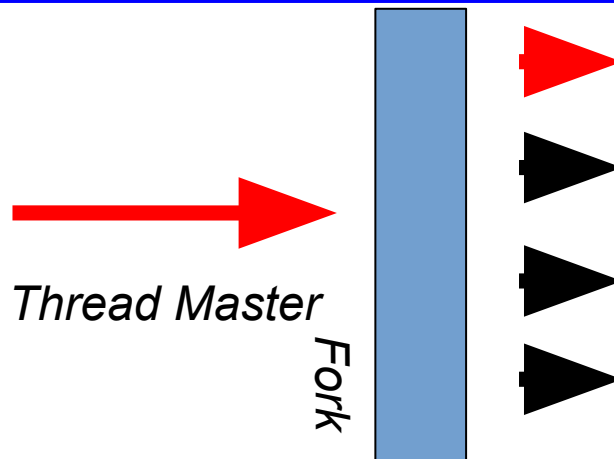
```
#include<stdio.h>
#include<omp.h>

int main(){
    printf("Sequencial %d\n", omp_get_num_threads());
    #pragma omp parallel
    {
        printf("Paralela %d\n", omp_get_num_threads());
    }
    printf("Sequencial %d\n", omp_get_num_threads());
}
```

02

TELA

Sequencial 1



Exemplo da Diretiva *omp parallel*

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

int main(){
    printf("Sequencial %d\n", omp_get_num_threads());
    #pragma omp parallel
    {
        printf("Paralela %d\n", omp_get_num_threads());
    }
    printf("Sequencial %d\n", omp_get_num_threads());
}
```

02

TELA

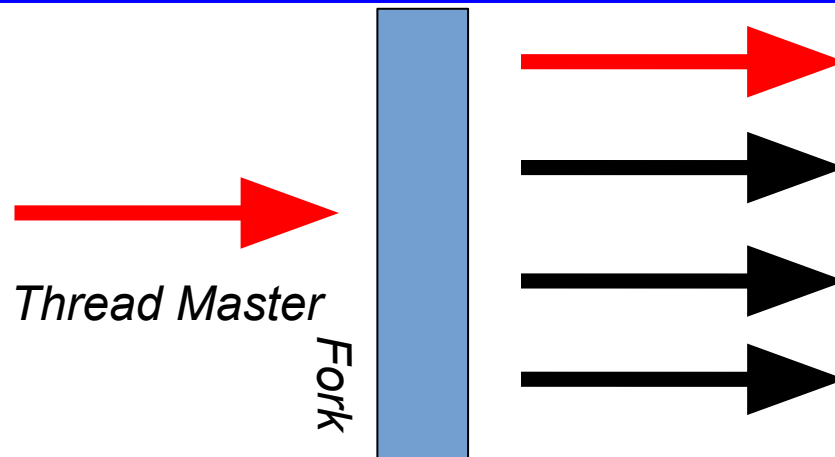
Sequencial 1

Paralela 4

Paralela 4

Paralela 4

Paralela 4



Exemplo da Diretiva *omp parallel*

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

int main(){
    printf("Sequencial %d\n", omp_get_num_threads());
    #pragma omp parallel
    {
        printf("Paralela %d\n", omp_get_num_threads());
    }
    printf("Sequencial %d\n", omp_get_num_threads());
}
```

02

TELA

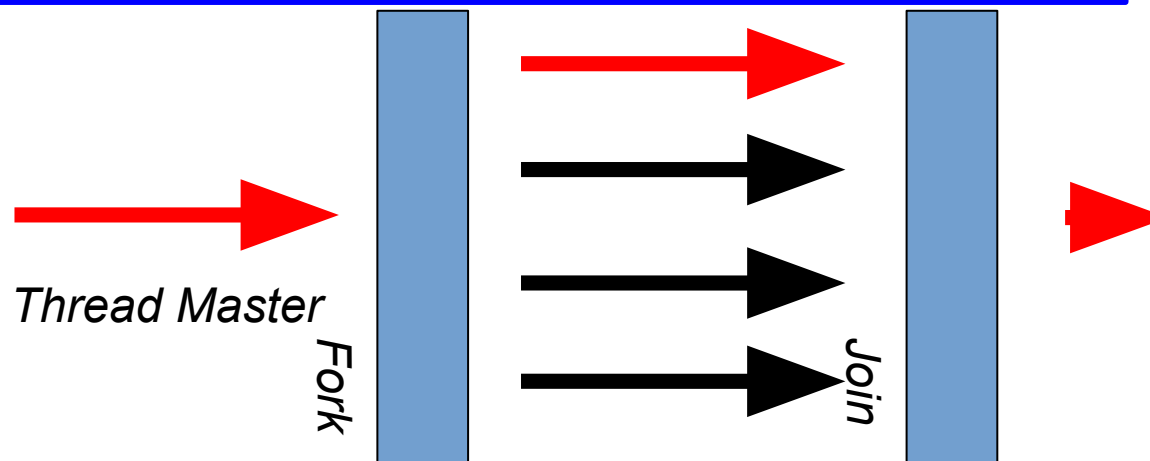
Sequencial 1

Paralela 4

Paralela 4

Paralela 4

Paralela 4



Exemplo da Diretiva *omp parallel*

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

int main(){
    printf("Sequencial %d\n", omp_get_num_threads());
    #pragma omp parallel
    {
        printf("Paralela %d\n", omp_get_num_threads());
    }
    printf("Sequencial %d\n", omp_get_num_threads());
}
```

02

TELA

Sequencial 1

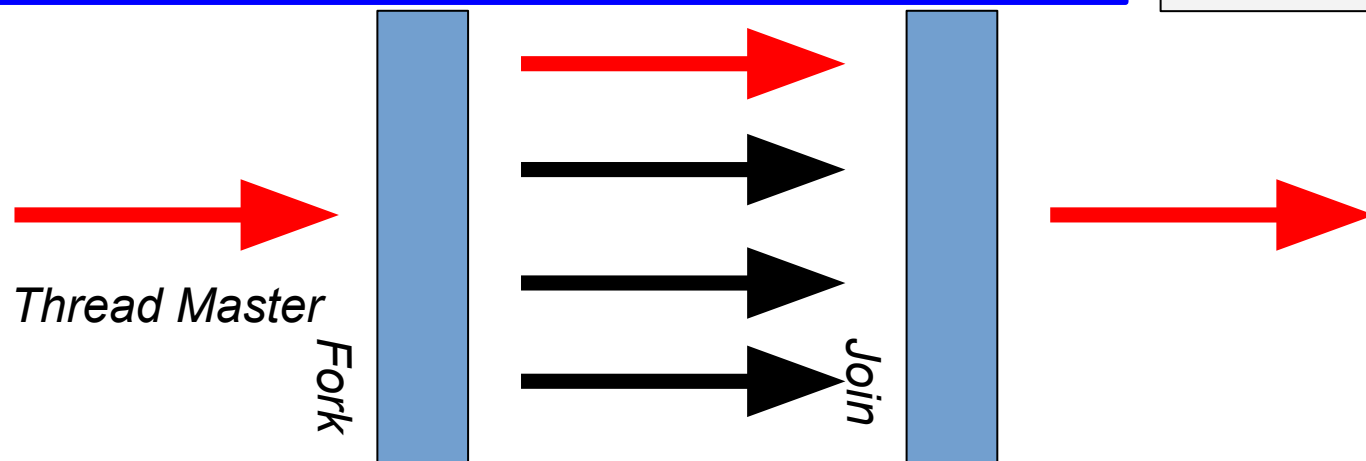
Paralela 4

Paralela 4

Paralela 4

Paralela 4

Sequencial 1



Exercício Resolvido (2)

- No código anterior, insira o comando `omp_set_num_threads(13)` antes do primeiro *printf*

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

int main(){
    omp_set_num_threads(13);

    printf("Sequencial %d\n", omp_get_num_threads());
    #pragma omp parallel
    {
        printf("Paralela %d\n", omp_get_num_threads());
    }
    printf("Sequencial %d\n", omp_get_num_threads());
}
```

03

Exercício Resolvido (3)

- O que acontece se executarmos o código abaixo várias vezes?

```
int main(){  
    #pragma omp parallel  
    for(int i = 1; i <= 4; i++)  
        printf("I = %d -- %d\n", i, omp_get_thread_num());  
}
```

04

Exercício Resolvido (3)

- O que acontece se executarmos o código abaixo várias vezes?

```
int main(){  
    #pragma omp parallel  
    for(int i = 1; i <= 4; i++)  
        printf("I = %d -- %d\n", i, omp_get_thread_num());  
}
```

04

Resposta: O *i* é uma variável local e não conseguimos garantir a ordem de execução de *threads* em paralelo

Exercício Resolvido (4)

- Mostre a diferença deste código para o anterior e explique o resultado

```
int main(){  
    int i;  
    #pragma omp parallel  
    for(i = 1; i <= 4; i++)  
        printf("I = %d -- %d\n", i, omp_get_thread_num());  
}
```

05

Exercício Resolvido (4)

- Mostre a diferença deste código para o anterior e explique o resultado

```
int main(){  
    int i;  
    #pragma omp parallel  
    for(i = 1; i <= 4; i++)  
        printf("I = %d -- %d\n", i, omp_get_thread_num());  
}
```

05

Resposta: A variável *i* é compartilhada entre as *threads*. Inicialmente, todas lêem o valor inicial de *i* que é um, contudo, como a *thread* 0 foi mais rápida, na segunda leitura das demais *threads*, o valor de *i* será 5

Diretiva *omp parallel for*

- Abre uma região paralela na qual as iterações do for são igualmente distribuídas entre as *threads*
- Este padrão de programação é conhecido como MAP

Exercício Resolvido (5)

- Explique o que acontece quando executamos o código abaixo

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

int main(){
    int i;
    #pragma omp parallel for num_threads(3)
    for(i = 0; i < 9; i++)
        printf("ID:%d, I: %d \n", omp_get_thread_num(), i);
}
```

06

Exercício Resolvido (5)

- Explique o que acontece quando executamos o código abaixo

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

int main(){
    int i;
    #pragma omp parallel for num_threads(3)
    for(i = 0; i < 9; i++)
        printf("ID:%d, I: %d \n", omp_get_thread_num(), i);
}
```

06

Thread 0	0	1	2
Thread 1	3	4	5
Thread 2	6	7	8

ID: 0 I: 0	ID: 1 I: 5
ID: 2 I: 6	ID: 2 I: 7
ID: 0 I: 1	ID: 2 I: 8
ID: 1 I: 3	ID: 0 I: 2
ID: 1 I: 4	

Exercício Resolvido (6)

- Explique o que acontece em cada código abaixo

```
for(int i = 0; i < 3; i++) 07  
    printf("I = %d\n", i);
```

```
#pragma omp parallel 07  
for(int i = 0; i < 3; i++)  
    printf("I = %d\n", i);
```

```
#pragma omp parallel for  
for(int i = 0; i < 3; i++)  
07    printf("I = %d\n", i);
```

Exercício Resolvido (6)

- Explique o que acontece em cada código abaixo

Sequencial

```
for(int i = 0; i < 3; i++) 07
    printf("I = %d\n", i);
```

Replica o código

```
#pragma omp parallel 07
for(int i = 0; i < 3; i++)
    printf("I = %d\n", i);
```

Distribui as iterações

```
#pragma omp parallel for
for(int i = 0; i < 3; i++) 07
    printf("I = %d\n", i);
```

I = 0
I = 1
I = 2

I = 0
I = 0
I = 0
I = 1
I = 1
I = 2
I = 1
I = 2
I = 2
I = 0
I = 1
I = 2

*4 threads e
a ordem da
impressão
pode variar*

I = 2
I = 0
I = 1

*A ordem da
impressão
pode variar*

Exercício Resolvido (7)

- Paralelize o código abaixo usando a diretiva “*omp parallel for*”

```
void matriz(int** mat, int size){  
    for(int i = 0; i < size; i++)  
        for(int j = 0; j < size; j++)  
            mat[i][j] = rand();  
}
```

08

Exercício Resolvido (7)

- Paralelize o código abaixo usando a diretiva “*omp parallel for*”

```
void matriz(int** mat, int size){  
    for(int i = 0; i < size; i++)  
        for(int j = 0; j < size; j++)  
            mat[i][j] = rand();  
}
```

08

1ª Resposta

```
void matriz2(int **mat, int size){  
    for(int i = 0; i < size; i++)  
        #pragma omp parallel for  
        for(int j = 0; j < size; j++)  
            mat[i][j] = rand();  
}
```

08

- Teremos *size fork* e *join*, um para cada *i*
- Em cada *fork* e *join*, dividimos as instruções entre as *t threads*

Exercício Resolvido (7)

- Paralelize o código abaixo usando a diretiva “*omp parallel for*”

```
void matriz(int** mat, int size){  
    for(int i = 0; i < size; i++)  
        for(int j = 0; j < size; j++)  
            mat[i][j] = rand();  
}
```

08

2ª Resposta

```
void matriz3(int **mat, int size){  
    int i, j;  
    #pragma omp parallel for private(j)  
    for(i = 0; i < size; i++)  
        for(j = 0; j < size; j++)  
            mat[i][j] = rand();  
}
```

08

Qual é a diferença nesta segunda resposta?

Exercício Resolvido (7)

- Paralelize o código abaixo usando a diretiva “*omp parallel for*”

```
void matriz(int** mat, int size){  
    for(int i = 0; i < size; i++)  
        for(int j = 0; j < size; j++)  
            mat[i][j] = rand();  
}
```

08

2ª Resposta

```
void matriz3(int **mat, int size){  
    int i, j;  
    #pragma omp parallel for private(j)  
    for(i = 0; i < size; i++)  
        for(j = 0; j < size; j++)  
            mat[i][j] = rand();  
}
```

08

- Teremos 1 *fork* e *join*, distribuindo as size^2 instruções pelas t threads

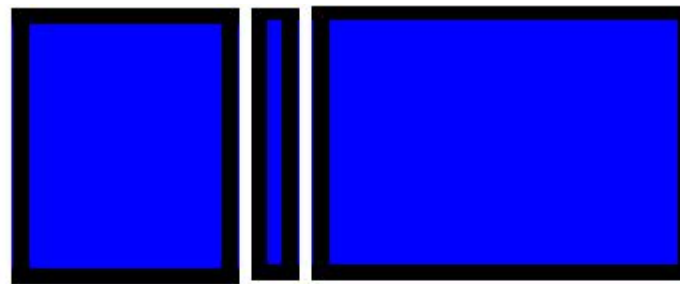
- Introdução
- Conceitos
- **OpenMP**
 - Sobre o OpenMP
 - Compilador gcc for Linux
 - Diretivas parallel e parallel for
 - **Mais conceitos básicos**
 - Diretivas sections, barrier, master e critical
 - Escopo das Variáveis
 - Cláusula Reduction
- Algoritmo
- Algoritmo

- Introdução
- Conceitos
 - Sobre o OpenMP
 - Compilador gcc for Linux
 - Diretivas parallel e parallel for
 - **Mais conceitos básicos**
 - Diretivas sections, barrier, master e critical
 - Escopo das Variáveis
 - Cláusula Reduction
- **OpenMP**
- Algoritmo
 - Granularidade
 - *Speedup*
 - Eficiência
- Algoritmo

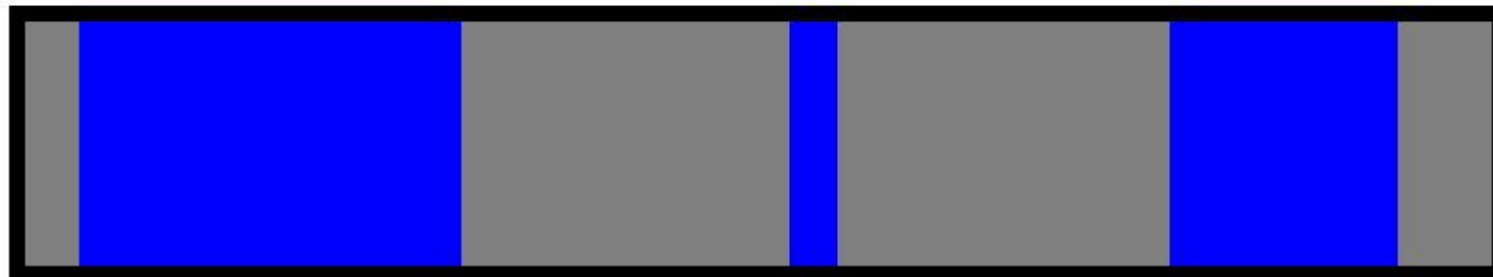
- Introdução
- Conceitos
- **OpenMP**
 - Sobre o OpenMP
 - Compilador gcc for Linux
 - Diretivas parallel e parallel for
 - **Mais conceitos básicos**
 - Diretivas sections, barrier, master e critical
 - Escopo das Variáveis
 - Cláusula Reduction
- Algoritmo
- Algoritmo
 - **Granularidade**
 - *Speedup*
 - Eficiência

Conceito Básico: Granularidade

- Porção das instruções do processo que uma mesma *thread* executa



Instruções
executadas
por uma
thread



Todas as
instruções

Tipos de Granularidade

- **Grossa**: Quando essa proporção é “grande”
- **Fina**: Quando a porção é pequena

```
//grão grosso com 2 processadores
switch (procesorID) {
  case 1: compute 1 - 50; break;
  case 2: compute 50 - 100; break;
}
```

```
//grão médio com 25 processadores
switch (procesorID) {
  case 1: compute 1 - 4; break;
  case 2: compute 5 - 8; break;
  case 3: compute 9 - 12; break;
  .
  .
  .
  case 100: compute 97 - 100; break;
}
```

```
//grão fino com 100 processadores
switch (procesorID) {
  case 1: compute 1; break;
  case 2: compute 2; break;
  case 3: compute 3; break;
  .
  .
  .
  case 100: compute 100; break;
}
```

Paralelismo de Grão Fino

- Programa é dividido em várias tarefas pequenas
- Cada tarefa é atribuída a um processador
- Aumentando o número de processadores, podemos comprometer a escalabilidade da solução

Paralelismo de Grão Fino

- Reduz a quantidade de trabalho associado a uma tarefa
- Distribui uniformemente o trabalho entre os processadores, facilitando o balanceamento de carga
- Demanda mais comunicação (*overhead*) entre as *threads*

Paralelismo de Grão Grosso

- Programa é dividido em grandes tarefas
- Grande quantidade de computação ocorre em processadores causando, eventualmente, desequilíbrio de carga entre os processadores
- Demanda menos comunicação e sobrecarga de sincronização

- Introdução
- Conceitos
 - Sobre o OpenMP
 - Compilador gcc for Linux
 - Diretivas parallel e parallel for
 - **Mais conceitos básicos**
 - Diretivas sections, barrier, master e critical
 - Escopo das Variáveis
 - Cláusula Reduction
- **OpenMP**
- Algoritmo
 - Granularidade
 - ***Speedup***
 - Eficiência
- Algoritmo

Conceito Básico: *Speedup*

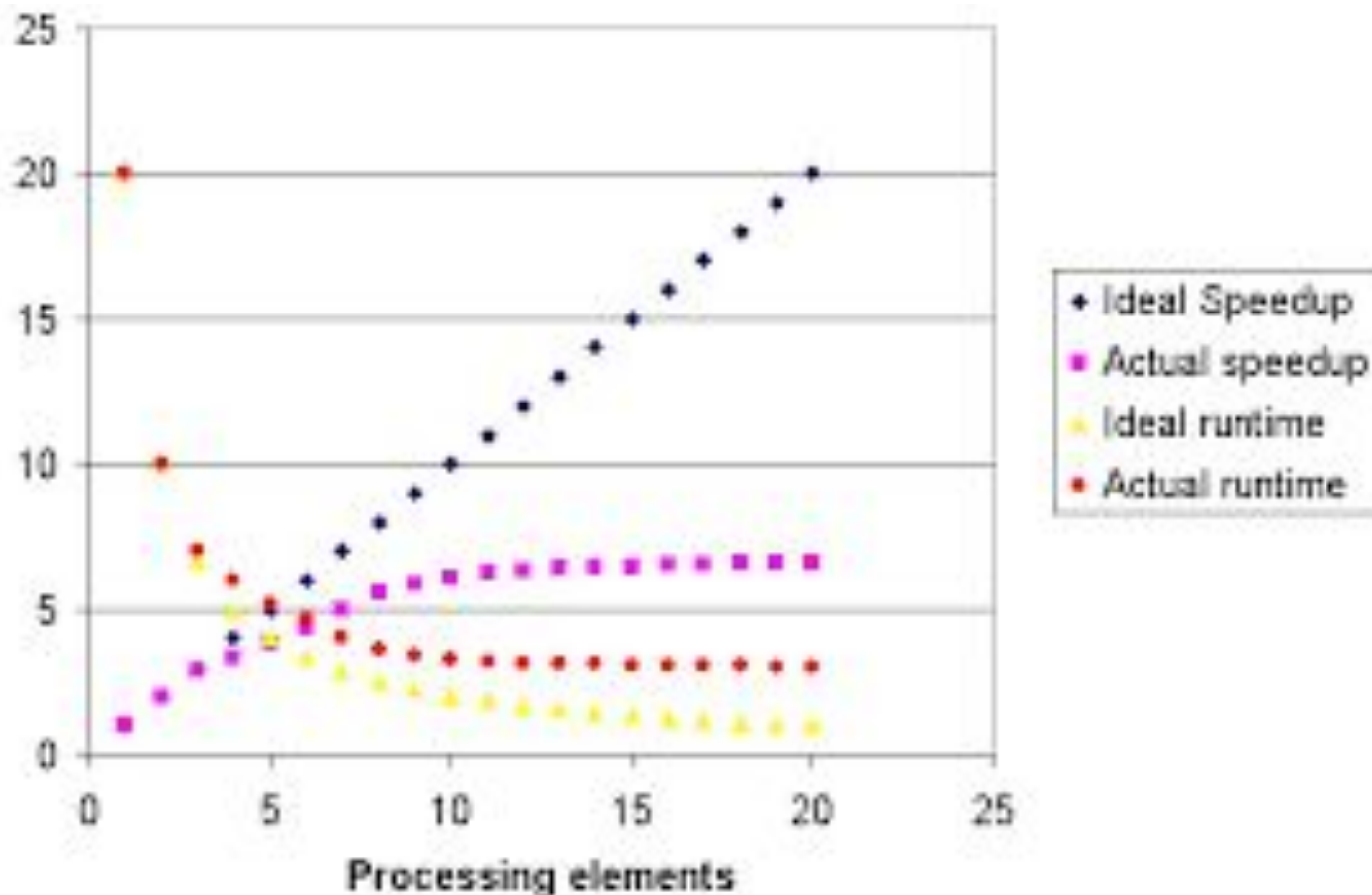
- Ganho de desempenho ao paralelizar uma aplicação

Tempo do Programa Sequencial

Tempo do Programa Paralelo

Conceito Básico: *Speedup*

- O ideal é que o *speedup* fosse igual ao número de núcleos utilizados, contudo, na prática, é menor



Exercício Resolvido (8)

- Temos um programa sequencial que executa em 10 segundos e sua versão paralela usando quatro núcleos executa em 5 segundos
- Qual é o nosso *speedup* ideal e o real?

Exercício Resolvido (8)

- Temos um programa sequencial que executa em 10 segundos e sua versão paralela usando quatro núcleos executa em 5 segundos
- Qual é o nosso *speedup* ideal e o real?

Resposta:

Speed up ideal = 4 (quatro núcleos)

$$\text{Speed up} = (T_{\text{SEQUENCIAL}} / T_{\text{PARALELO}}) = (10 / 5) = 2$$

- Introdução
- Conceitos
 - Sobre o OpenMP
 - Compilador gcc for Linux
 - Diretivas parallel e parallel for
 - **Mais conceitos básicos**
 - Diretivas sections, barrier, master e critical
 - Escopo das Variáveis
 - Cláusula Reduction
- **OpenMP**
- Algoritmo
- Algoritmo
 - Granularidade
 - *Speedup*
 - **Eficiência**

Conceito Básico: Eficiência

- Métrica derivada do *speedup* cujo valor varia entre 0 e 1 sendo que 1 (ou 100%) acontece com o *speedup* ideal

Speedup

Número de núcleos

Exercício Resolvido (9)

- Qual é a eficiência do exemplo apresentado no exercício anterior?

Exercício Resolvido (9)

- Qual é a eficiência do exemplo apresentado no exercício anterior?

Resposta:

Como temos 4 núcleos e 2 de *speed up*, a eficiência será:

$$2 / 4 = 0,5 \text{ (50\% de aproveitamento)}$$

Exercício Resolvido (10)

- Apresente o *speedup* e a eficiência obtidos nas paralelizações abaixo

(size = 10...)

```
void matriz(int** mat, int size){  
    for(int i = 0; i < size; i++)  
        for(int j = 0; j < size; j++)  
            mat[i][j] = rand();  
} 09
```

```
void matriz2(int **mat, int size){  
    for(int i = 0; i < size; i++)  
        #pragma omp parallel for  
        for(int j = 0; j < size; j++)  
            mat[i][j] = rand();  
} 09
```

```
void matriz3(int **mat, int size){  
    int i, j;  
    #pragma omp parallel for private(j)  
    for(i = 0; i < size; i++)  
        for(j = 0; j < size; j++)  
            mat[i][j] = rand();  
} 09
```

Exercício Resolvido (10)

- Apresente o *speedup* e a eficiência obtidos nas paralelizações abaixo

(size = 10...)

```
void matriz(int** mat, int size){
    for(int i = 0; i < size; i++)
        for(int j = 0; j < size; j++)
            mat[i][j] = rand();
} 09
```

```
void matriz2
    for(int i
        #pragm
        for(in
            mat
    } 09
```

Tempo (seq): 1.2 s.

Tempo (par1): 70.9 s.

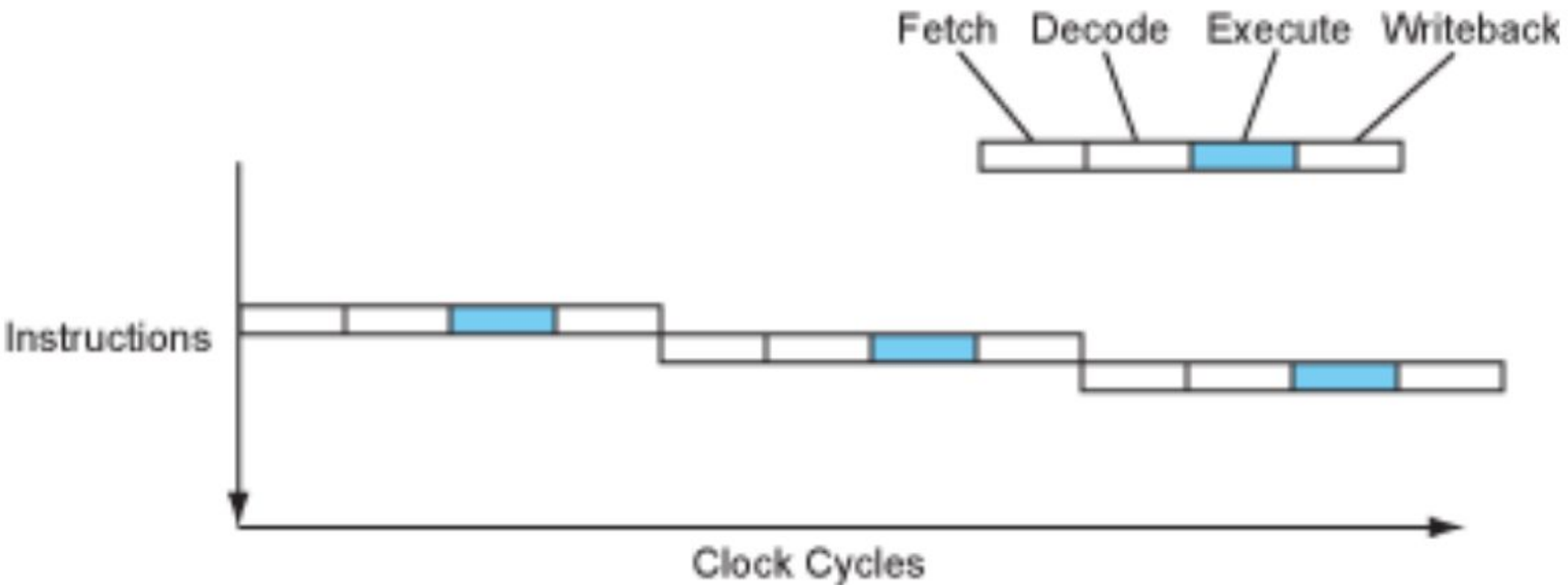
Tempo (par2): 57.1 s.

E agora José?

```
#pragma omp parallel for private(j)
for(i = 0; i < size; i++)
    for(j = 0; j < size; j++)
        mat[i][j] = rand();
} 09
```

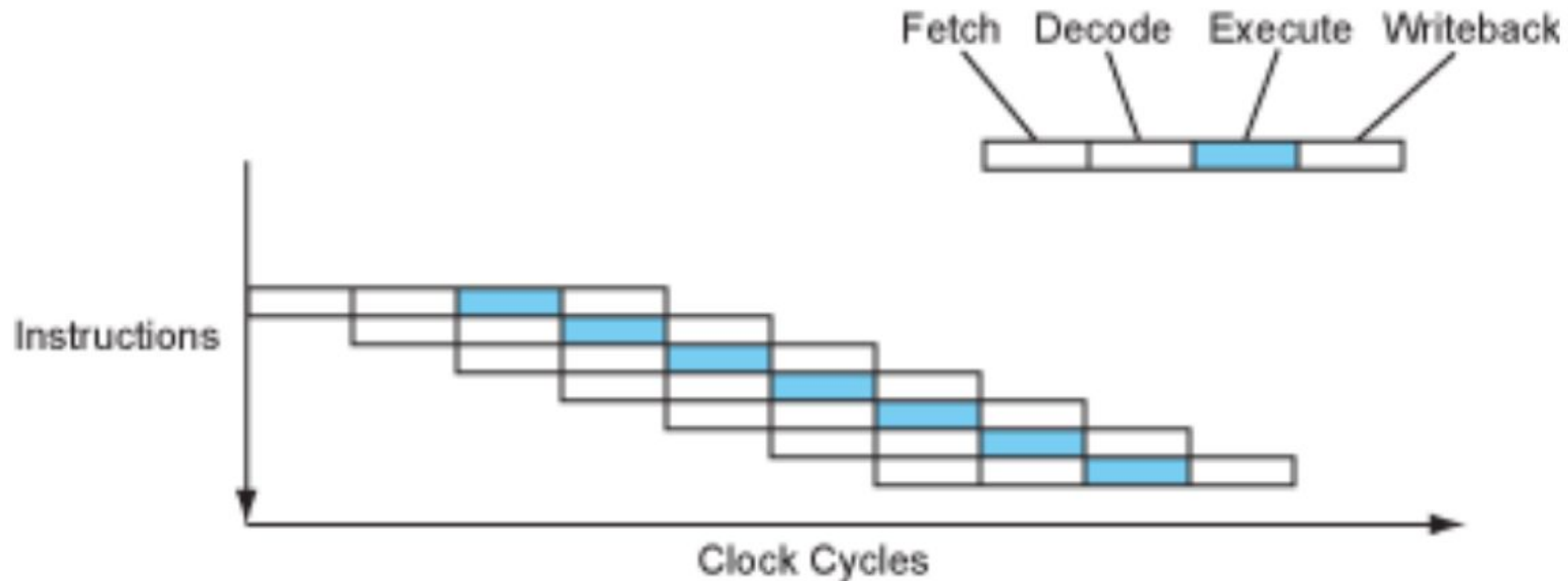
Arquiteturas Paralelas

- Por exemplo, paralelismo de instruções (e.g., pipeline)



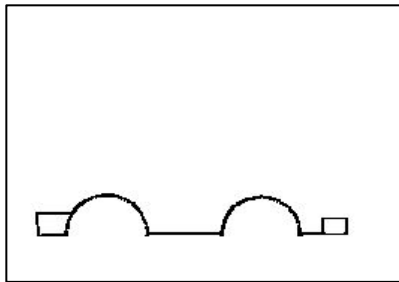
Arquiteturas Paralelas

- Por exemplo, paralelismo de instruções (e.g., pipeline)

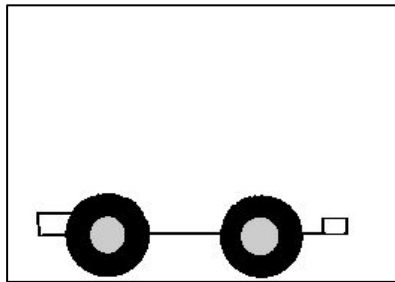


Exemplo de Pipeline: Indústria Automobilística

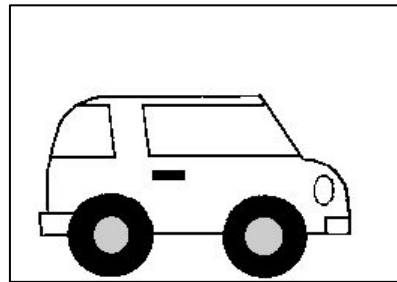
Etapa (1):
Assoalho



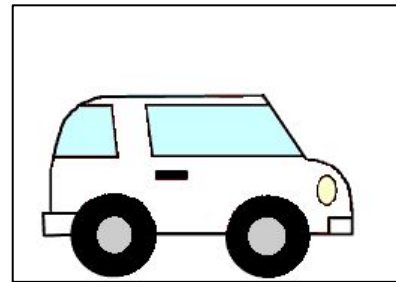
Etapa (2):
Roda



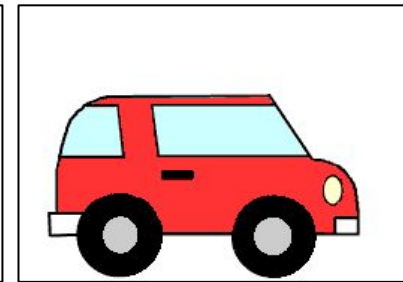
Etapa (3):
Carcaça



Etapa (4):
Vidros



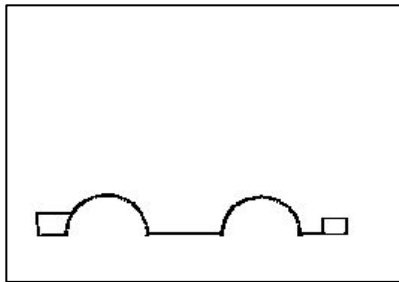
Etapa (5):
Pintura



Se cada etapa demora 1 minuto, nós produzimos um carro a cada 5 minutos. E se utilizarmos *pipeline*?

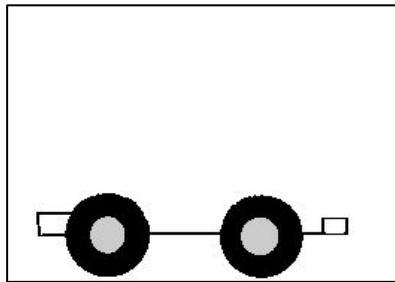
Exemplo de Pipeline: Indústria Automobilística

Etapa (1):
Assoalho

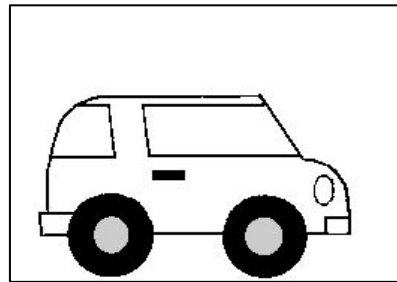


CARRO(1)

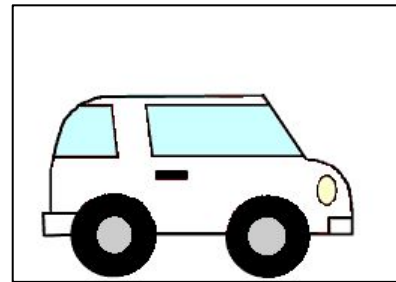
Etapa (2):
Roda



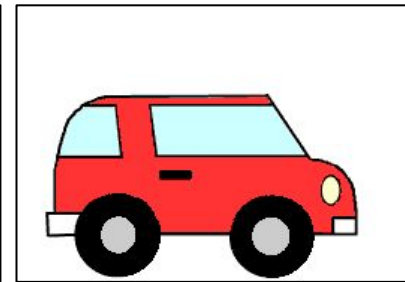
Etapa (3):
Carcaça



Etapa (4):
Vidros



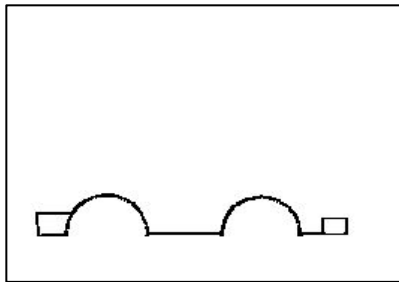
Etapa (5):
Pintura



T₁

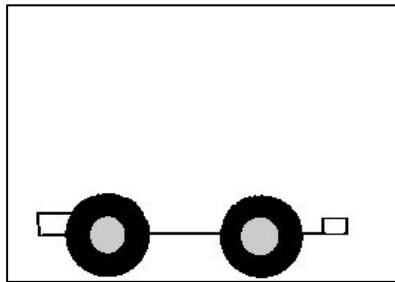
Exemplo de Pipeline: Indústria Automobilística

Etapa (1):
Assoalho



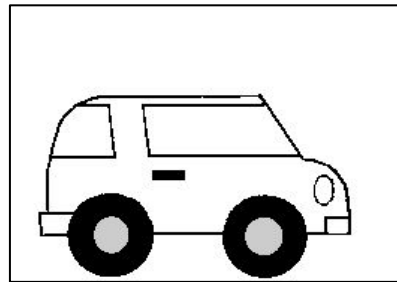
CARRO(2)

Etapa (2):
Roda

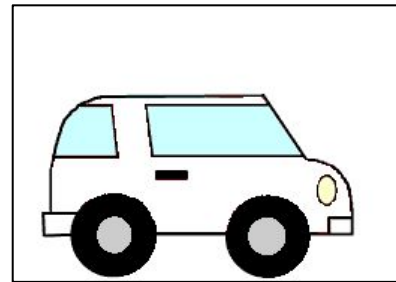


CARRO(1)

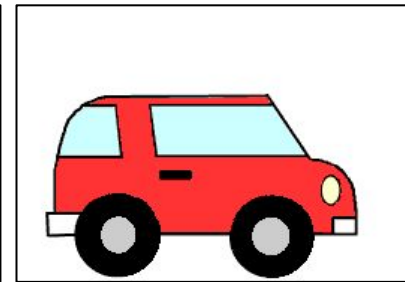
Etapa (3):
Carcaça



Etapa (4):
Vidros



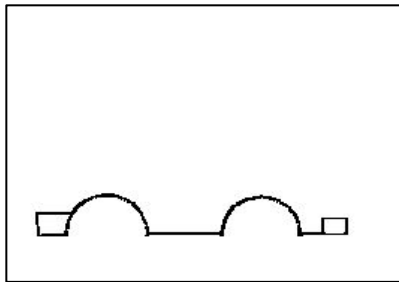
Etapa (5):
Pintura



T_2

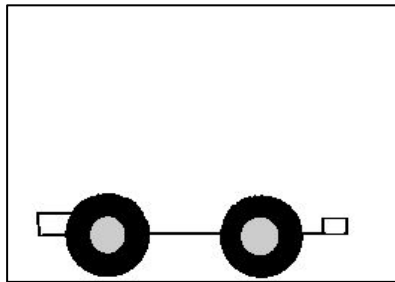
Exemplo de Pipeline: Indústria Automobilística

Etapa (1):
Assoalho



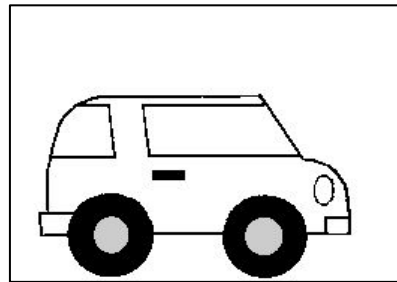
CARRO(3)

Etapa (2):
Roda



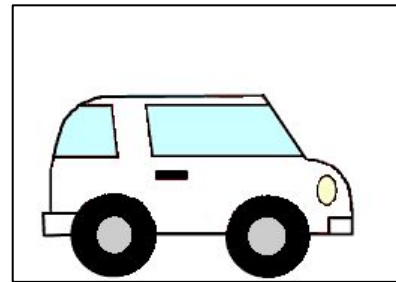
CARRO(2)

Etapa (3):
Carcaça

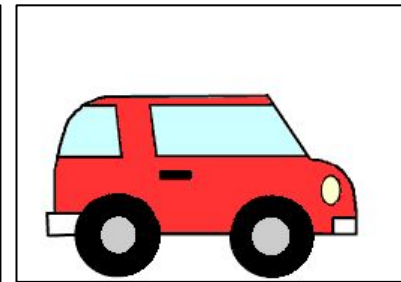


CARRO(1)

Etapa (4):
Vidros



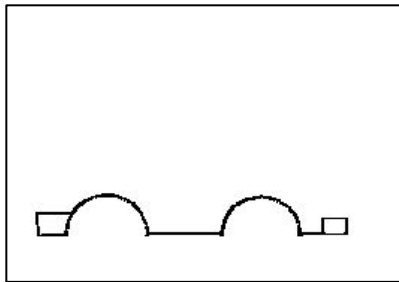
Etapa (5):
Pintura



T_3

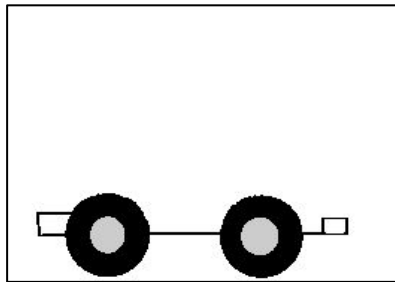
Exemplo de Pipeline: Indústria Automobilística

Etapa (1):
Assoalho



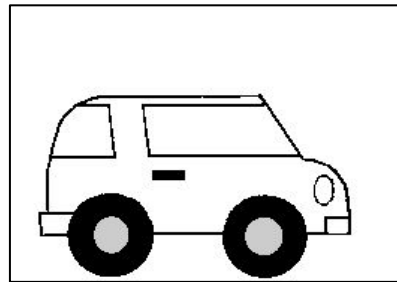
CARRO(4)

Etapa (2):
Roda



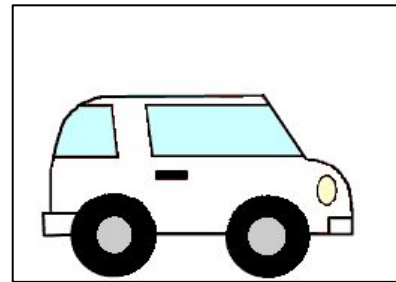
CARRO(3)

Etapa (3):
Carcaça



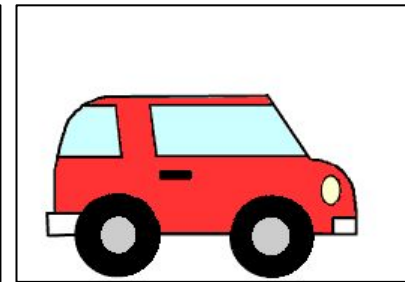
CARRO(2)

Etapa (4):
Vidros



CARRO(1)

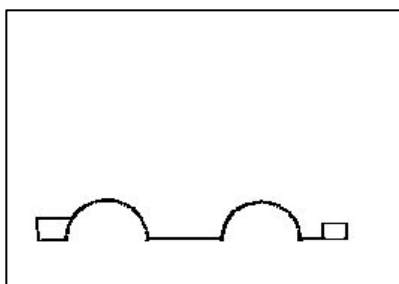
Etapa (5):
Pintura



T₄

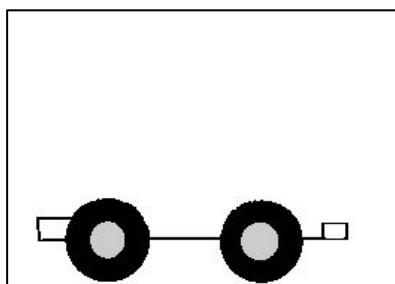
Exemplo de Pipeline: Indústria Automobilística

Etapa (1):
Assoalho



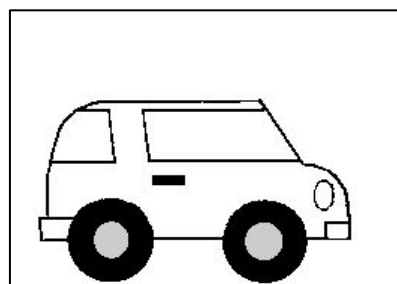
CARRO(5)

Etapa (2):
Roda



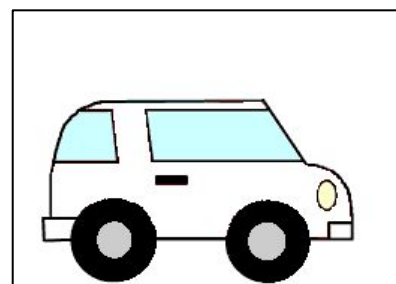
CARRO(4)

Etapa (3):
Carcaça



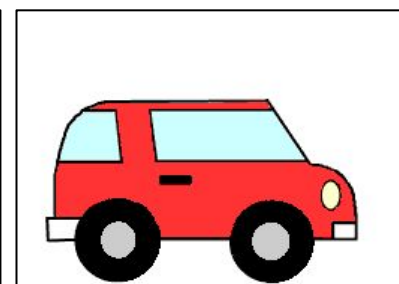
CARRO(3)

Etapa (4):
Vidros



CARRO(2)

Etapa (5):
Pintura



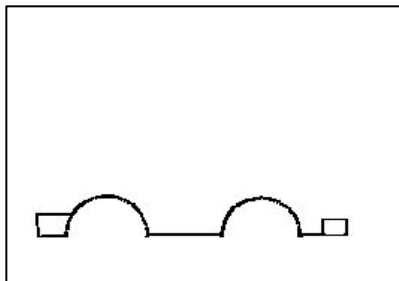
CARRO(1)

CARRO(1) pronto!!!!

T₅

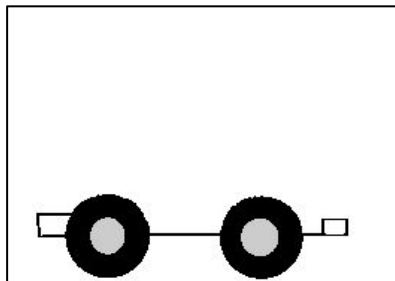
Exemplo de Pipeline: Indústria Automobilística

Etapa (1):
Assoalho



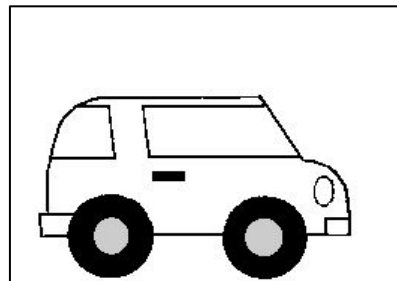
CARRO(6)

Etapa (2):
Roda



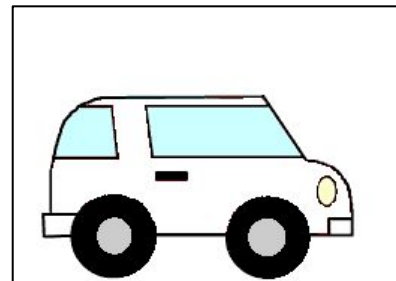
CARRO(5)

Etapa (3):
Carcaça



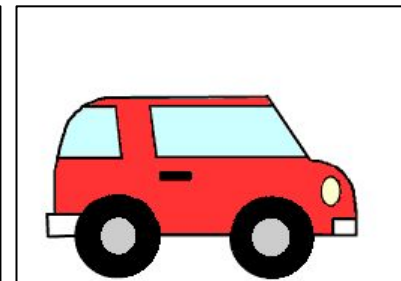
CARRO(4)

Etapa (4):
Vidros



CARRO(3)

Etapa (5):
Pintura



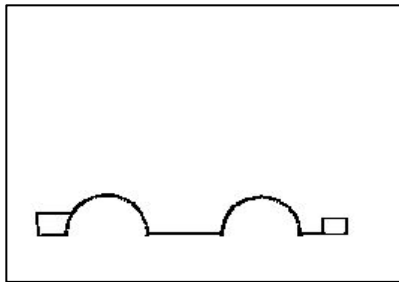
CARRO(2)

T_6

CARRO(2) pronto!!!!

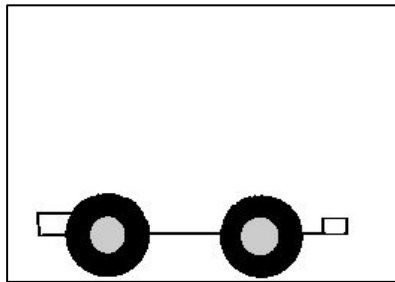
Exemplo de Pipeline: Indústria Automobilística

Etapa (1):
Assoalho



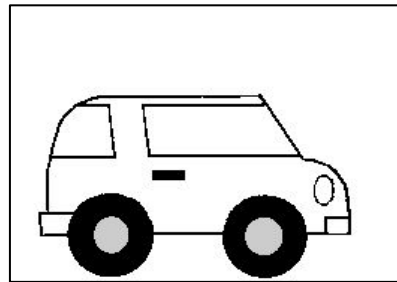
CARRO(7)

Etapa (2):
Roda



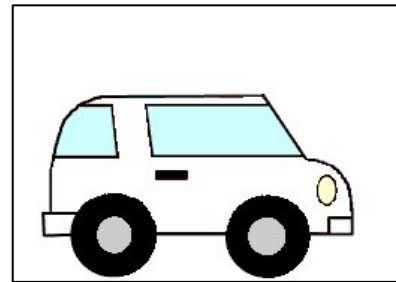
CARRO(6)

Etapa (3):
Carcaça



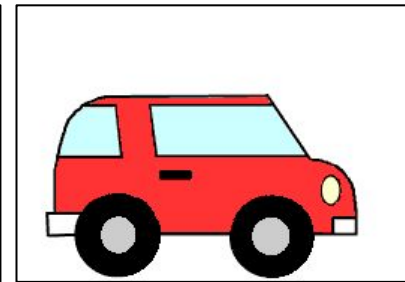
CARRO(5)

Etapa (4):
Vidros



CARRO(4)

Etapa (5):
Pintura



CARRO(3)

CARRO(3) pronto!!!!

T_7

Computação Paralela

- Problemas cuja complexidade é maior ou igual a $O(n^2)$
- Problemas com etapas independentes

Exercício Resolvido (12)

- Faça um programa (versões sequencial e paralela) que leia um texto e mostre o número de ocorrências de uma palavra no texto. Em seguida, calcule o *speedup* e eficiência

Exercício Resolvido (12)

```
//DECLARACAO DA STRING A SER PROCURADA =====
char* str = (char*) malloc(100 * sizeof(char));
strcpy(str, "Página");

//DECLARACAO E ALOCACAO DO VETOR =====
char **palavra = malloc (SIZE * sizeof(char*));
for (int i = 0; i < SIZE; ++i){
    palavra[i] = malloc (100 * sizeof(char));
}

//LEITURA DAS FRASES =====
int numPalavra = 0;
while(numPalavra < SIZE && strcmp(palavra[numPalavra++], "</html>")){
    scanf("%s", palavra[numPalavra]);
    printf("(%i): %s\n", numPalavra, palavra[numPalavra]);
}
```

10

Exercício Resolvido (12)

```
//ALGORITMO SEQUENCIAL =====  
int soma = 0;  
for(int i = 0; i < numPalavra; i++){  
    if(compareTo(palavra[i], str)){  
        soma++;  
    }  
}
```

```
bool compareTo(char* s1, char* s2){  
    bool resp = true, sair = false;  
  
    for(int i = 0; sair == false; i++){  
        if(s1[i] == '\\0'){  
            sair = true;  
            resp = (s2[i] == '\\0');  
        } else if(s1[i] != s2[i]){  
            sair = true;  
            resp = false;  
        }  
    }  
  
    return resp;  
}
```

10

10

Exercício Resolvido (12)

```
//ALGORITMO PARALELO =====
soma = 0;
omp_set_num_threads(4);
#pragma omp parallel
{
    int numPalavraLocal = numPalavra / omp_get_num_threads();
    int deslocamento = omp_get_thread_num()*numPalavraLocal;
    for(int i = deslocamento; i < (deslocamento + numPalavraLocal); i++){
        if(compareTo(palavra[i], str)){
            soma++;
        }
    }
}
```

10

Por exemplo: Se tivermos 400 palavras e 4 threads:

- numPalavraLocal será 100
- Thread 0: deslocamento = $(0 * 100)$ e término do for em $[(0 * 100) + 100]$
- Thread 1: deslocamento = $(1 * 100)$ e término do for em $[(1 * 100) + 100]$
- Thread 2: deslocamento = $(2 * 100)$ e término do for em $[(2 * 100) + 100]$
- Thread 3: deslocamento = $(3 * 100)$ e término do for em $[(3 * 100) + 100]$

Exercício Resolvido (12)

- Faça um programa (versões sequencial e paralela) que leia um texto e mostre o número de ocorrências de uma palavra no texto. Em seguida, calcule o *speedup* e eficiência

Tempo (seq): 0.000641 s

Tempo (par): 0.000395 s

$$\text{speedup} = (T_{\text{SEQ}} / T_{\text{PAR}}) \\ = 1.62$$

$$\text{eficiência} = (S_{\text{REAL}} / \\ S_{\text{IDEAL}}) \\ = 40,57\%$$

Exercício Resolvido (13)

- Repita o exercício de matriz seguindo a ideia do deslocamento da questão anterior. Em seguida, calcule o *speedup* e eficiência

Exercício Resolvido (13)

- Repita o exercício de matriz seguindo a ideia do deslocamento da questão anterior. Em seguida, calcule o *speedup* e eficiência

```
void matriz4(int **mat, int size){  
    #pragma omp parallel  
    {  
        int size_local = size / omp_get_num_threads();  
        int deslocamento = omp_get_thread_num()*size_local;  
        int inicio = deslocamento;  
        int fim = deslocamento + size_local;  
        for(int i = inicio; i < fim; i++){  
            for(int j=0; j < size; j++){  
                mat[i][j] = rand();  
            }  
        }  
    }  
}
```

11

- Introdução
- Conceitos
- **OpenMP**
 - Sobre o OpenMP
 - Compilador gcc for Linux
 - Diretivas parallel e parallel for
 - Mais conceitos básicos
 - **Diretivas sections, barrier, master e critical**
 - Escopo das Variáveis
 - Cláusula Reduction
- Algoritmo
- Algoritmo

Diretiva “*omp parallel sections*”

- Segmenta o código em seções sendo que cada seção é executada por uma única *thread*

Exemplo da Diretiva “*omp parallel sections*”

```
int main(){
    omp_set_num_threads(3);
    #pragma omp parallel sections
    {
        #pragma omp section
        for(int i = 1; i < 10; i++)
            printf("Thread(%d): %d\n", omp_get_thread_num(), i);

        #pragma omp section
        for(int i = 1; i < 10; i++)
            printf("Thread(%d): %d\n", omp_get_thread_num(), (i*10));

        #pragma omp section
        for(int i = 1; i < 10; i++)
            printf("Thread(%d): %d\n", omp_get_thread_num(), (i*100));
    }
}
```

12

Exemplo da Diretiva “*omp parallel sections*”

```
int main(){  
    omp_set_num_threads(3);  
    #pragma omp parallel sections  
    {  
        #pragma omp section
```

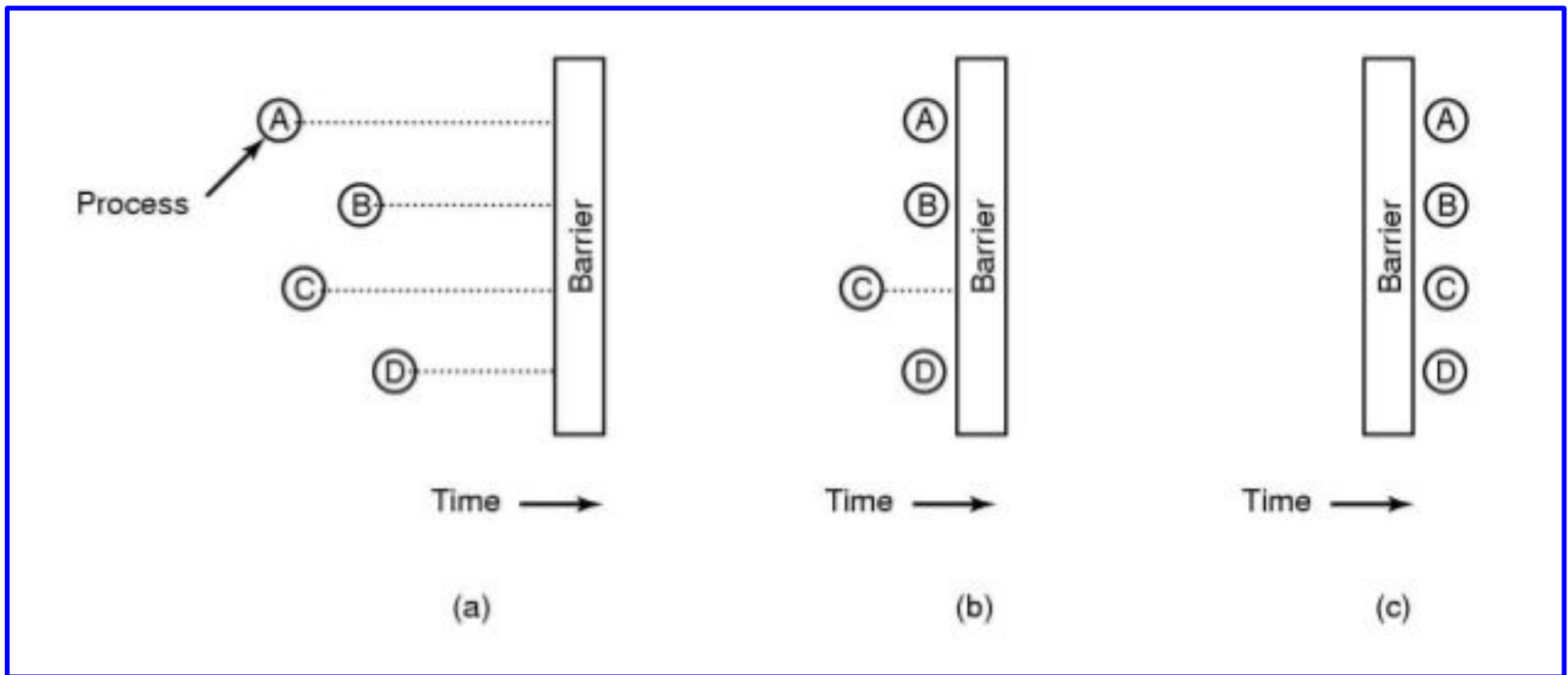
Observação (1): Se tivemos 6 núcleos disponíveis, apenas 3 deles serão utilizados; um para cada seção

Observação (2): Se tivermos apenas 2, um deles executará a terceira seção

```
        #pragma omp section  
        for(int i = 1; i < 10; i++)  
            printf("Thread(%d): %d\n", omp_get_thread_num(), (i*100));  
    }  
}
```

Diretiva “*omp barrier*”

- Impõe um ponto de sincronização no qual qualquer *thread* só pode ultrapassá-lo quando todas as demais atingirem esse ponto



Diretiva “*omp barrier*”

- Impõe um ponto de sincronização no qual qualquer *thread* só pode ultrapassá-lo quando todas as demais atingirem esse ponto



Observação: Um efeito colateral é o aumento de *overhead*

Time →

(a)

Time →

(b)

Time →

(c)

Exemplo da Diretiva “*omp barrier*”

```

omp_set_num_threads(3);
#pragma omp parallel
{
    printf("Fase 1\n");
    printf("Fase 2\n");
}

printf("\n\n===== \n\n");

#pragma omp parallel
{
    printf("Fase 1\n");
    #pragma omp barrier
    printf("Fase 2\n");
}

```

Fase 1
 Fase 2
 Fase 1
 Fase 2
 Fase 1
 Fase 2

Fase 1
 Fase 1
 Fase 1
 Fase 2
 Fase 2
 Fase 2

Diretiva “*omp master*”

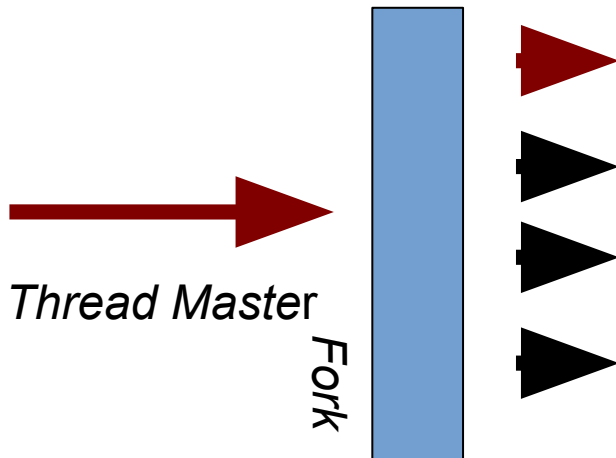
- Abre uma seção que será executada somente pela *thread master* (ID 0)

Exemplo da Diretiva “*omp master*”

```
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    printf("Thread %d\n", omp_get_thread_num());  
    #pragma omp barrier  
    #pragma omp master  
    {  
        printf("Master %d\n", omp_get_thread_num());  
    }  
}
```


Exemplo da Diretiva “*omp master*”

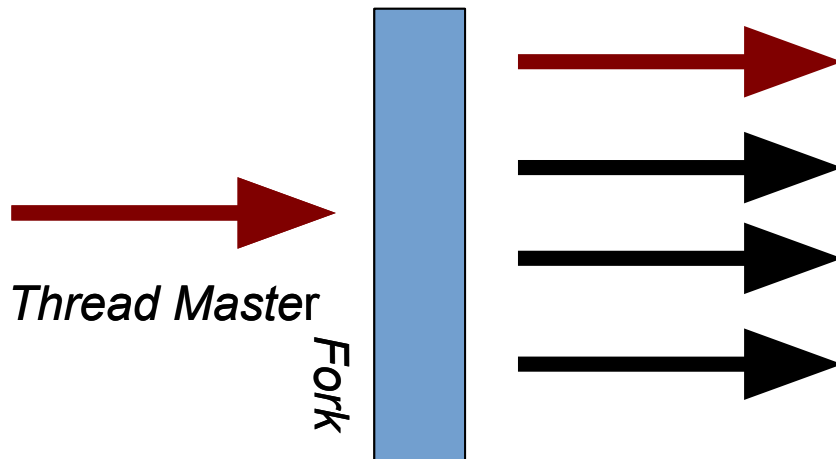
```
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    printf("Thread %d\n", omp_get_thread_num());  
    #pragma omp barrier  
    #pragma omp master  
    {  
        printf("Master %d\n", omp_get_thread_num());  
    }  
}
```



Exemplo da Diretiva “*omp master*”

```
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    printf("Thread %d\n", omp_get_thread_num());  
    #pragma omp barrier  
    #pragma omp master  
    {  
        printf("Master %d\n", omp_get_thread_num());  
    }  
}
```

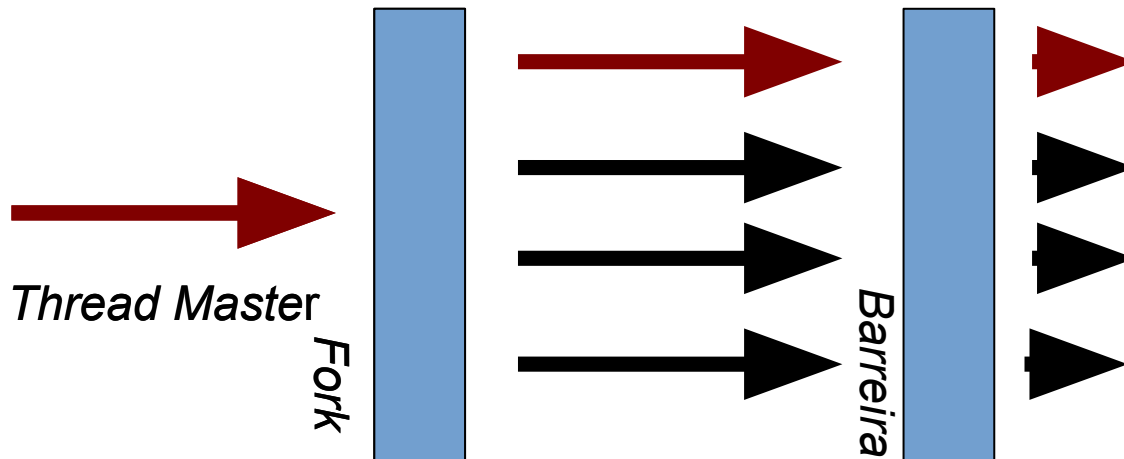
Thread 1
Thread 0
Thread 3
Thread 2



Exemplo da Diretiva “*omp master*”

```
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    printf("Thread %d\n", omp_get_thread_num());  
    #pragma omp barrier  
    #pragma omp master  
    {  
        printf("Master %d\n", omp_get_thread_num());  
    }  
}
```

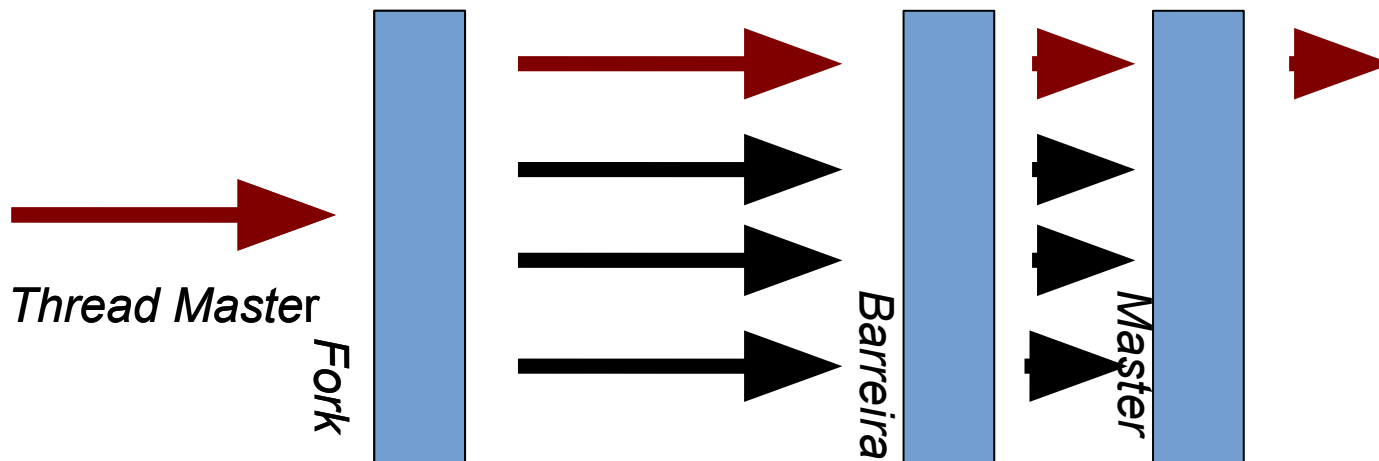
Thread 1
Thread 0
Thread 3
Thread 2



Exemplo da Diretiva “*omp master*”

```
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    printf("Thread %d\n", omp_get_thread_num());  
    #pragma omp barrier  
    #pragma omp master  
    {  
        printf("Master %d\n", omp_get_thread_num());  
    }  
}
```

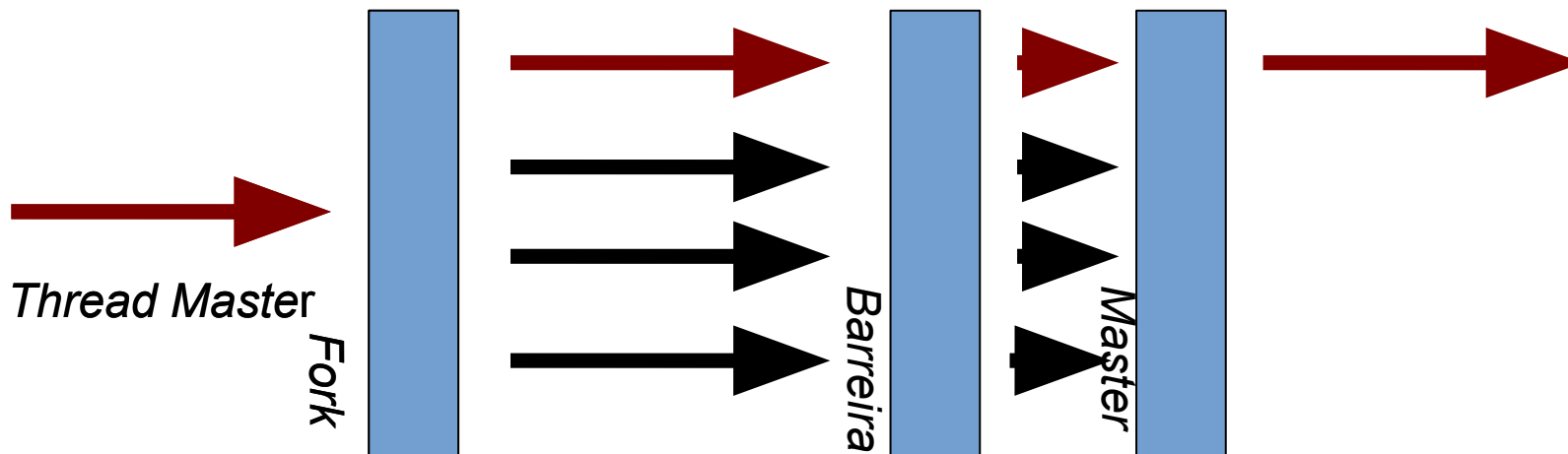
Thread 1
Thread 0
Thread 3
Thread 2



Exemplo da Diretiva “*omp master*”

```
omp_set_num_threads(4);
#pragma omp parallel
{
    printf("Thread %d\n", omp_get_thread_num());
    #pragma omp barrier
    #pragma omp master
    {
        printf("Master %d\n", omp_get_thread_num());
    }
}
```

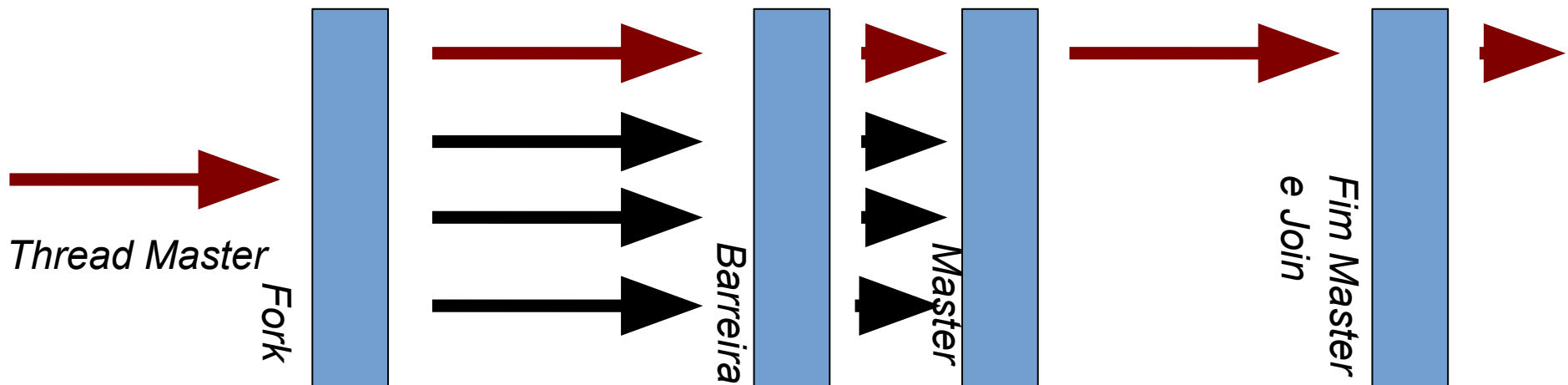
Thread 1
Thread 0
Thread 3
Thread 2
Master 0



Exemplo da Diretiva “*omp master*”

```
omp_set_num_threads(4);
#pragma omp parallel
{
    printf("Thread %d\n", omp_get_thread_num());
    #pragma omp barrier
    #pragma omp master
    {
        printf("Master %d\n", omp_get_thread_num());
    }
}
```

Thread 1
Thread 0
Thread 3
Thread 2
Master 0



Diretiva “*omp critical*”

- Cria uma região crítica na qual não é permitido o acesso simultâneo de duas *threads*. Quando uma *thread* deseja acessar tal região e essa está ocupada por outra *thread*, a primeira *thread* aguarda a segunda

Diretiva “omp critical”

```
int main() {  
    int max = 0;  
    #pragma omp parallel  
    {  
        int local = rand();  
        #pragma omp barrier  
        if(max < local)  
            max = local;  
    }  
}
```

max	local - T0	local - T1	local - T2
N/A	N/A	N/A	N/A

Geramos aleatoriamente um número inteiro para cada *thread* e o maior valor gerado entre as *threads* é armazenado na variável compartilhada **max**

Condição de corrida: *threads* escrevem na variável compartilhada

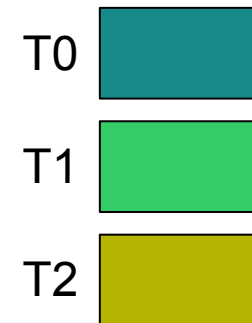
Diretiva “omp critical”

```

int main() {
    int max = 0;
    #pragma omp parallel
    {
        int local = rand();
        #pragma omp barrier
        if(max < local)
            max = local;
    }
}

```

max	local - T0	local - T1	local - T2
0	N/A	N/A	N/A



Thread Master

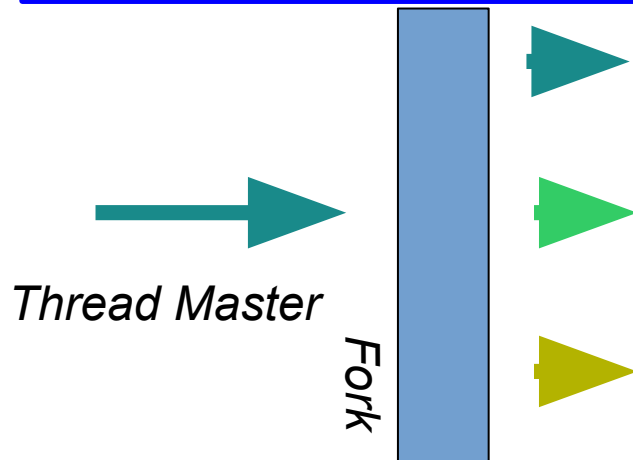
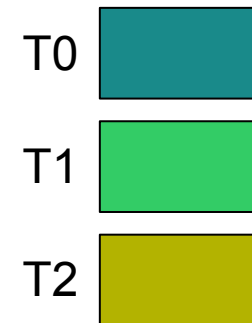
Diretiva “omp critical”

```

int main() {
    int max = 0;
    #pragma omp parallel
    {
        int local = rand();
        #pragma omp barrier
        if(max < local)
            max = local;
    }
}

```

max	local - T0	local - T1	local - T2
0	N/A	N/A	N/A



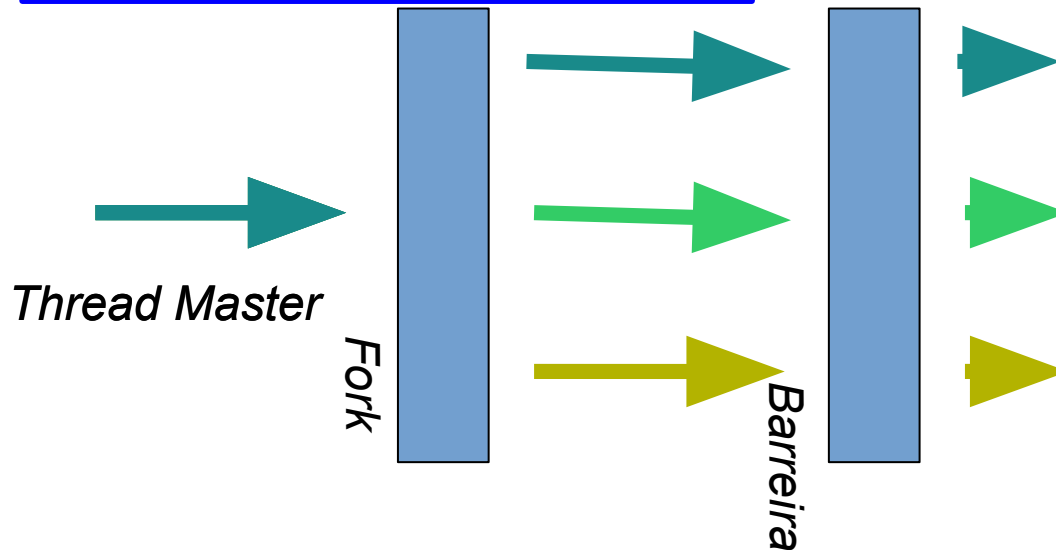
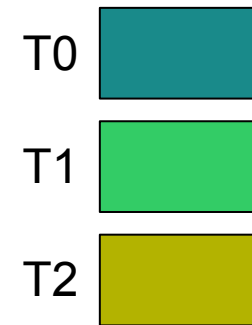
Diretiva “omp critical”

```

int main() {
    int max = 0;
    #pragma omp parallel
    {
        int local = rand();
        #pragma omp barrier
        if(max < local)
            max = local;
    }
}

```

max	local - T0	local - T1	local - T2
0	5	3	7



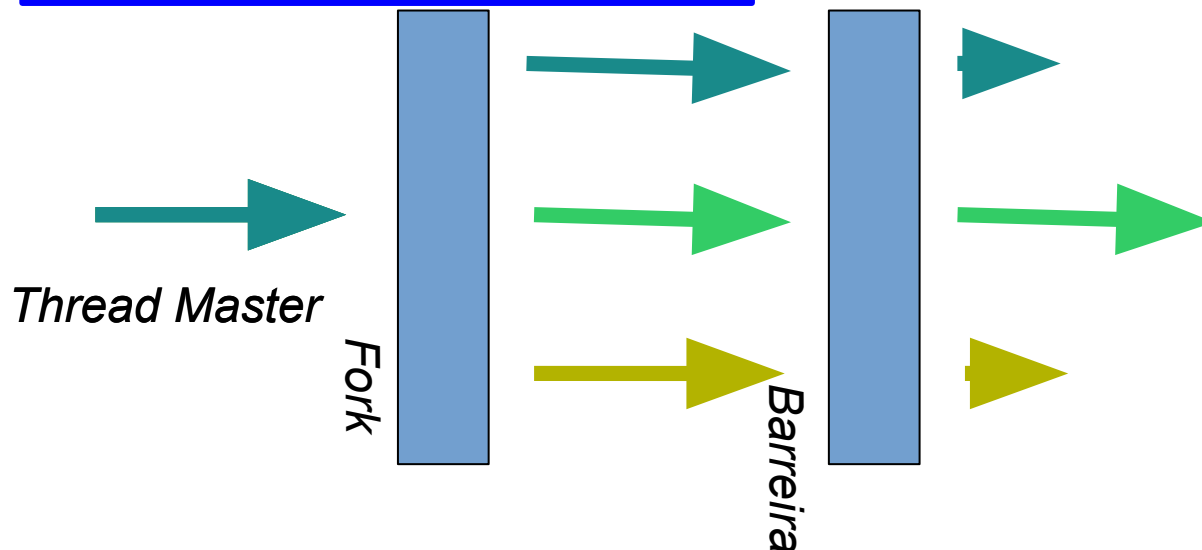
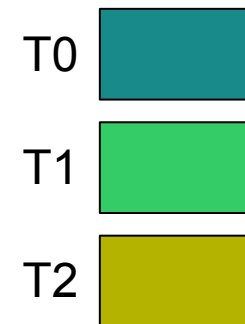
Diretiva “omp critical”

```

int main() {
    int max = 0;
    #pragma omp parallel
    {
        int local = rand();
        #pragma omp barrier
        if(max < local)
            max = local;
    }
}

```

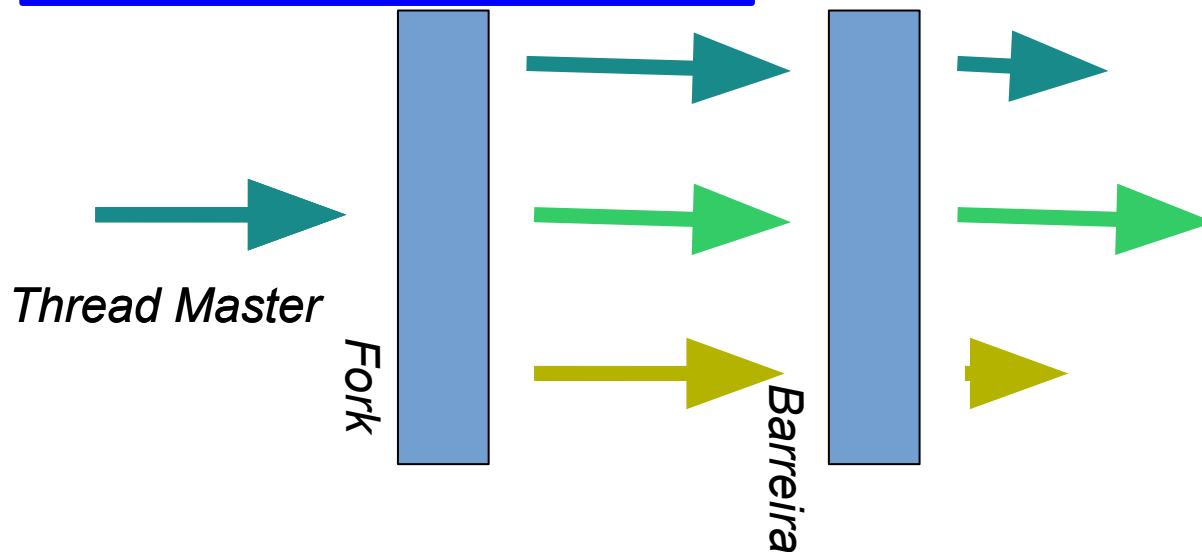
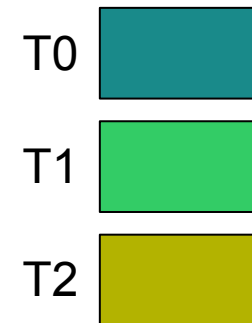
max	local - T0	local - T1	local - T2
3	5	3	7



Diretiva “omp critical”

```
int main() {  
    int max = 0;  
    #pragma omp parallel  
    {  
        int local = rand();  
        #pragma omp barrier  
        if(max < local)  
            max = local;  
    }  
}
```

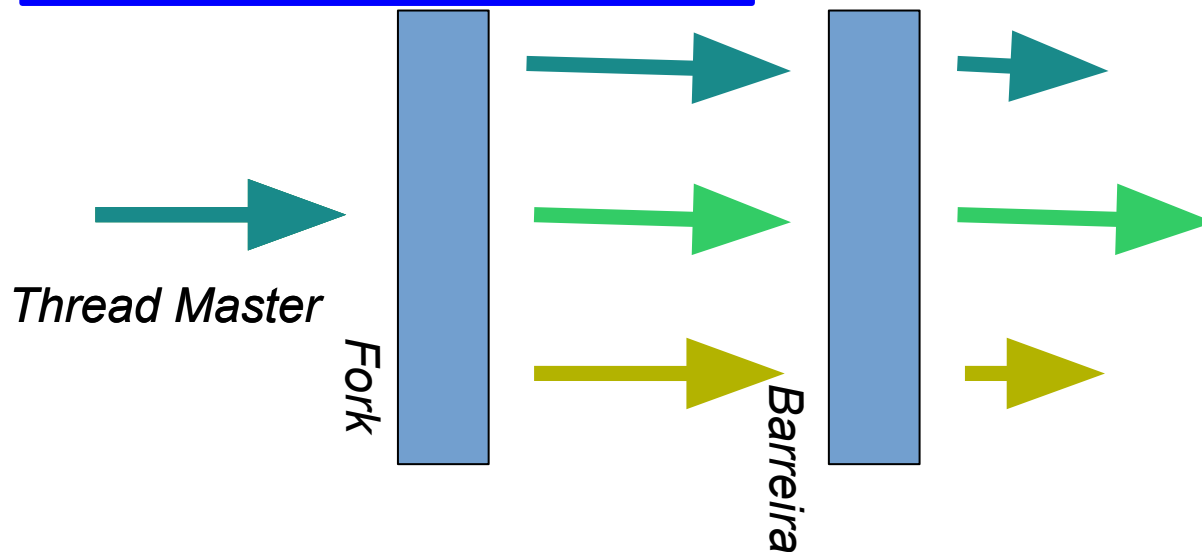
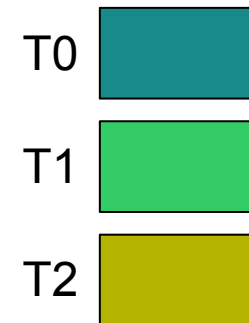
max	local - T0	local - T1	local - T2
3	5	3	7



Diretiva “omp critical”

```
int main() {  
    int max = 0;  
    #pragma omp parallel  
    {  
        int local = rand();  
        #pragma omp barrier  
        if(max < local)  
            max = local;  
    }  
}
```

max	local - T0	local - T1	local - T2
3	5	3	7



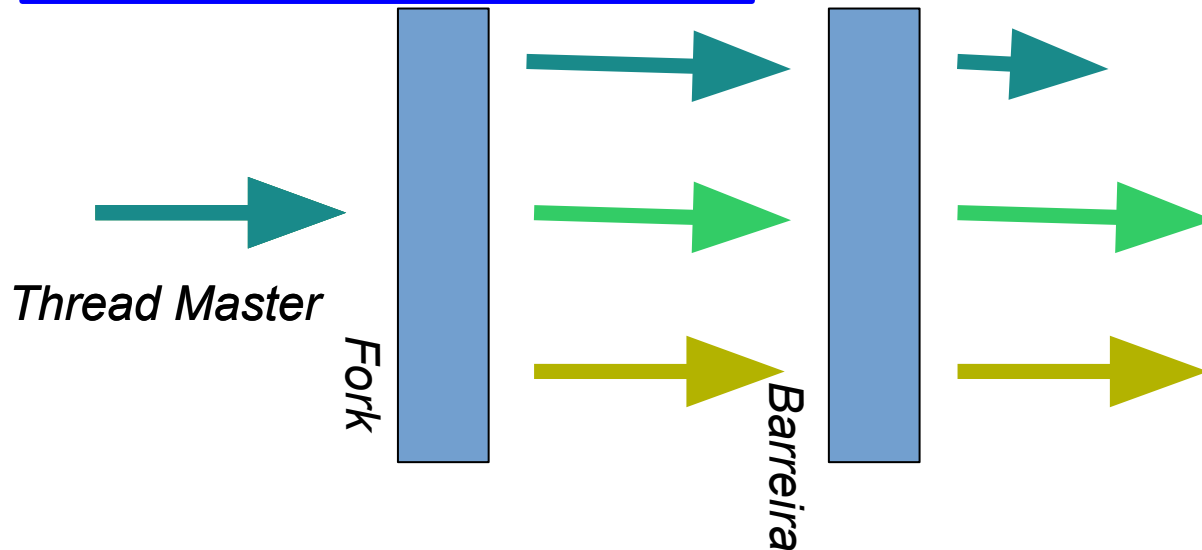
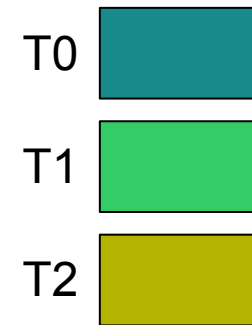
Diretiva “omp critical”

```

int main() {
    int max = 0;
    #pragma omp parallel
    {
        int local = rand();
        #pragma omp barrier
        if(max < local)
            max = local;
    }
}

```

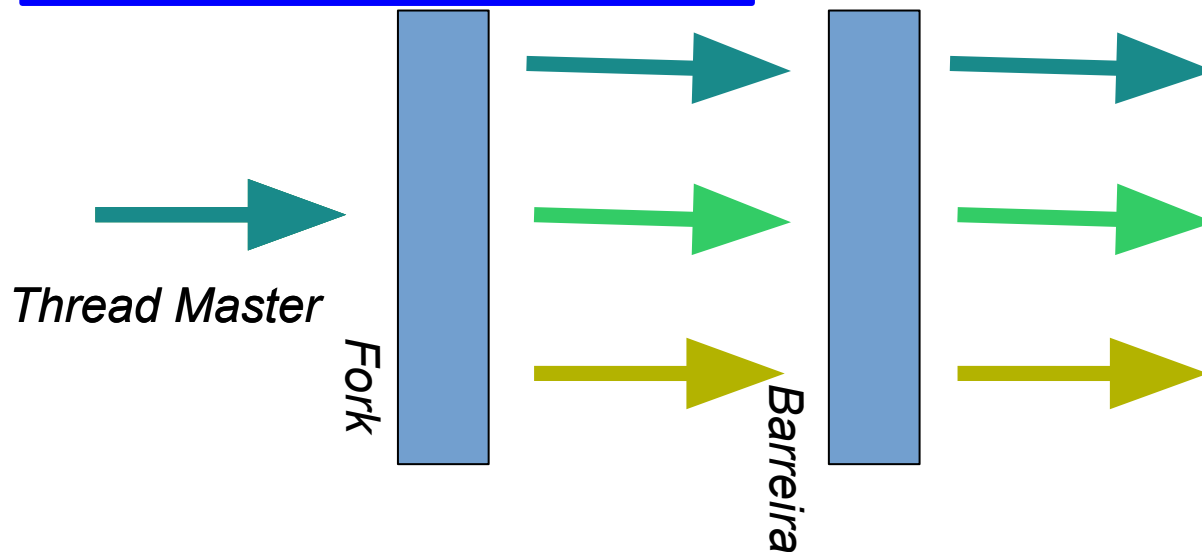
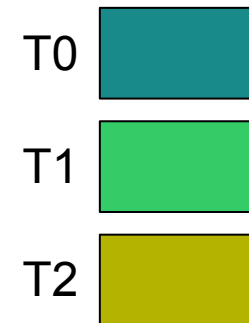
max	local - T0	local - T1	local - T2
7	5	3	7



Diretiva “omp critical”

```
int main() {  
    int max = 0;  
    #pragma omp parallel  
    {  
        int local = rand();  
        #pragma omp barrier  
        if(max < local)  
            max = local;  
    }  
}
```

max	local - T0	local - T1	local - T2
5	5	3	7



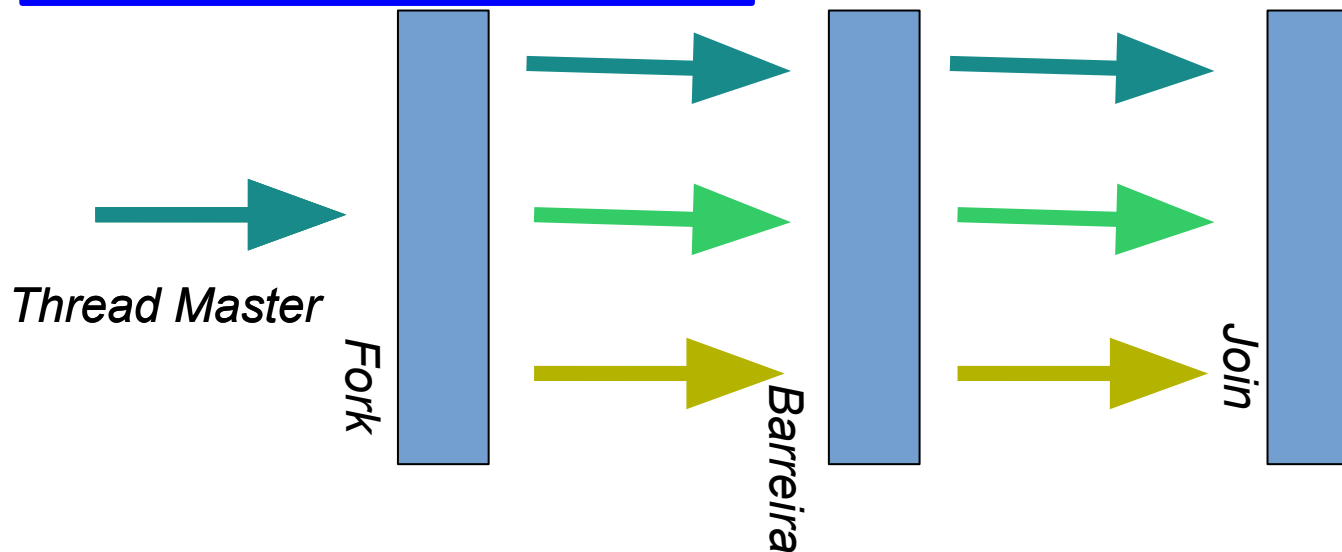
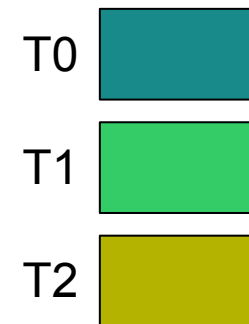
Diretiva “omp critical”

```

int main() {
    int max = 0;
    #pragma omp parallel
    {
        int local = rand();
        #pragma omp barrier
        if(max < local)
            max = local;
    }
}

```

max	local - T0	local - T1	local - T2
5	5	3	7



Diretiva “omp critical”

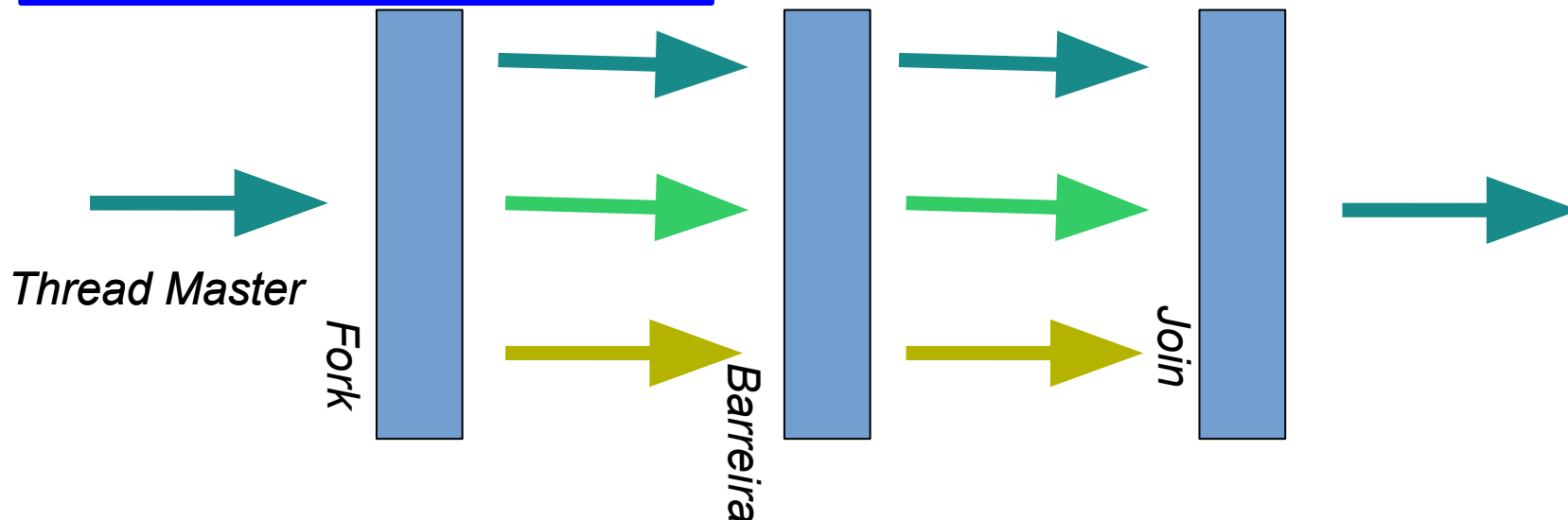
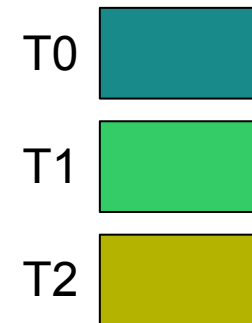
```

int main() {
    int max = 0;
    #pragma omp parallel
    {
        int local = rand();
        #pragma omp barrier
        if(max < local)
            max = local;
    }
}

```

max	local - T0	local - T1	local - T2
5	5	3	7

Resultado incorreto



Diretiva “omp critical”

```
int main() {  
    int max = 0;  
    #pragma omp parallel  
    {  
        int local = rand();  
        #pragma omp barrier  
        #pragma omp critical  
        {  
            if(max < local)  
                max = local;  
        }  
    }  
}
```

max	local - T0	local - T1	local - T2
N/A	N/A	N/A	N/A

Diretiva “omp critical”

```

int main() {
    int max = 0;
    #pragma omp parallel
    {
        int local = rand();
        #pragma omp barrier
        #pragma omp critical
        {
            if(max < local)
                max = local;
        }
    }
}

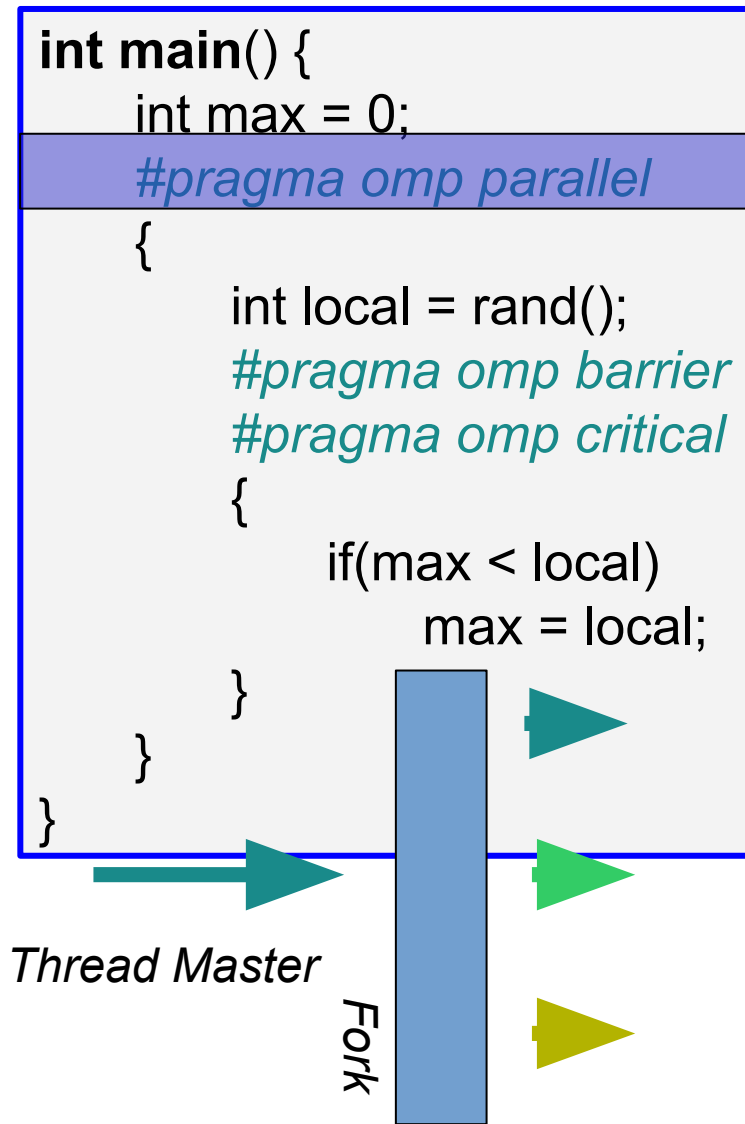
```



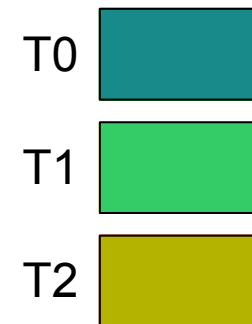
Thread Master

max	local - T0	local - T1	local - T2
0	N/A	N/A	N/A

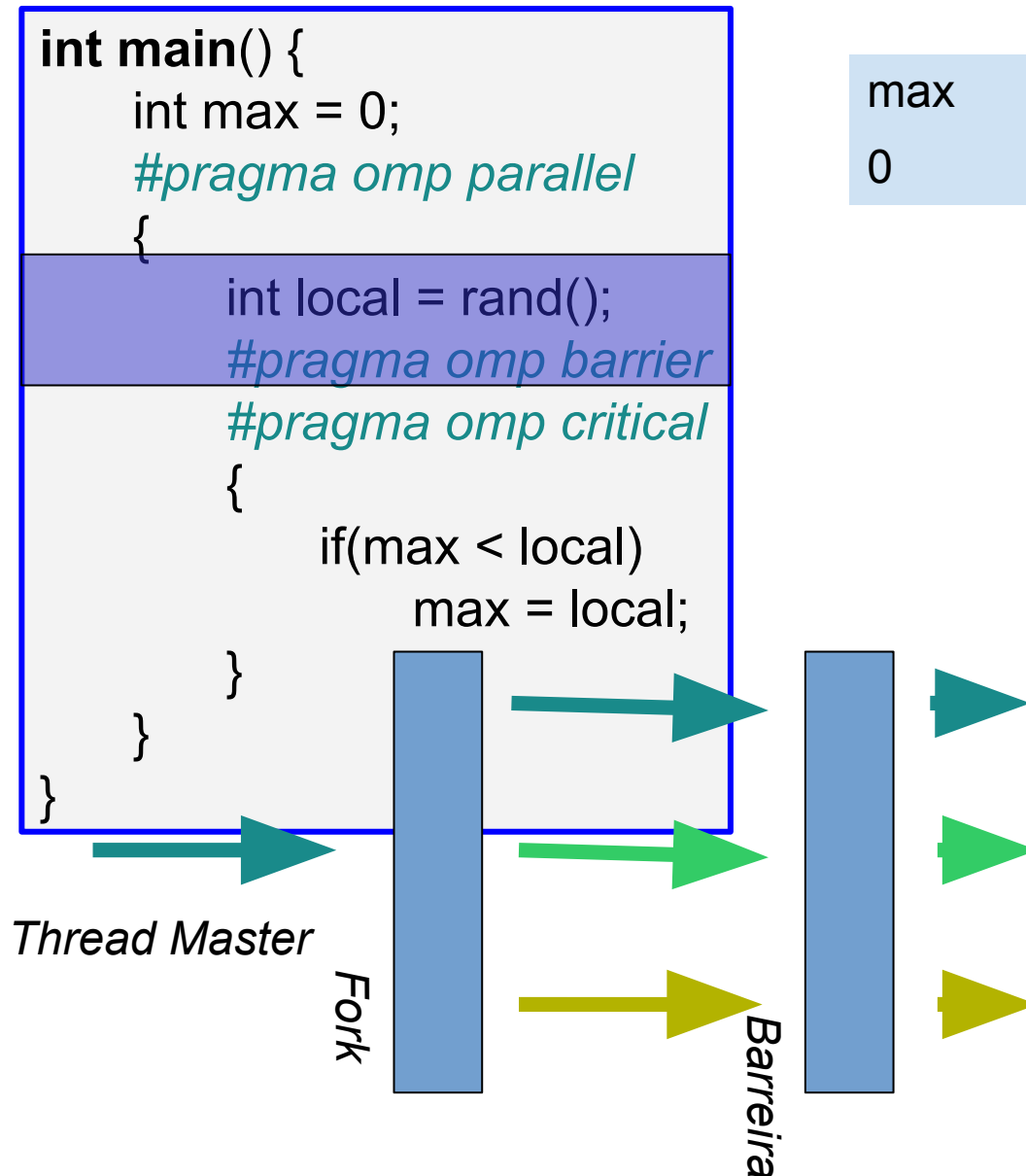
Diretiva “omp critical”



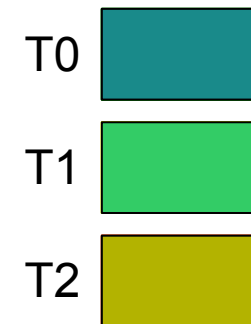
max	local - T0	local - T1	local - T2
0	N/A	N/A	N/A



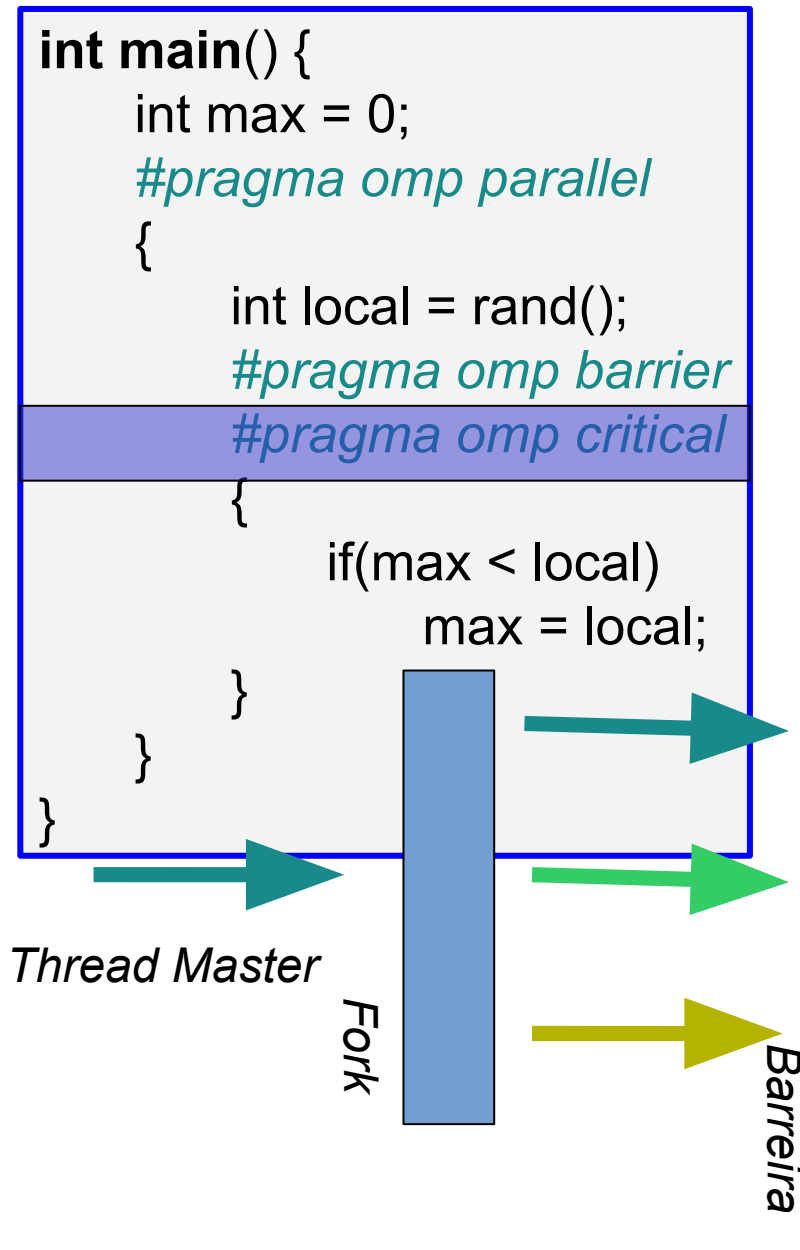
Diretiva “omp critical”



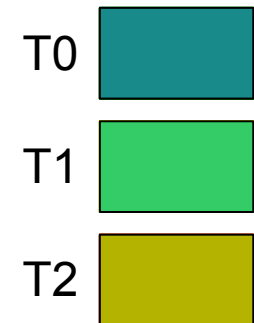
max	local - T0	local - T1	local - T2
0	5	3	7



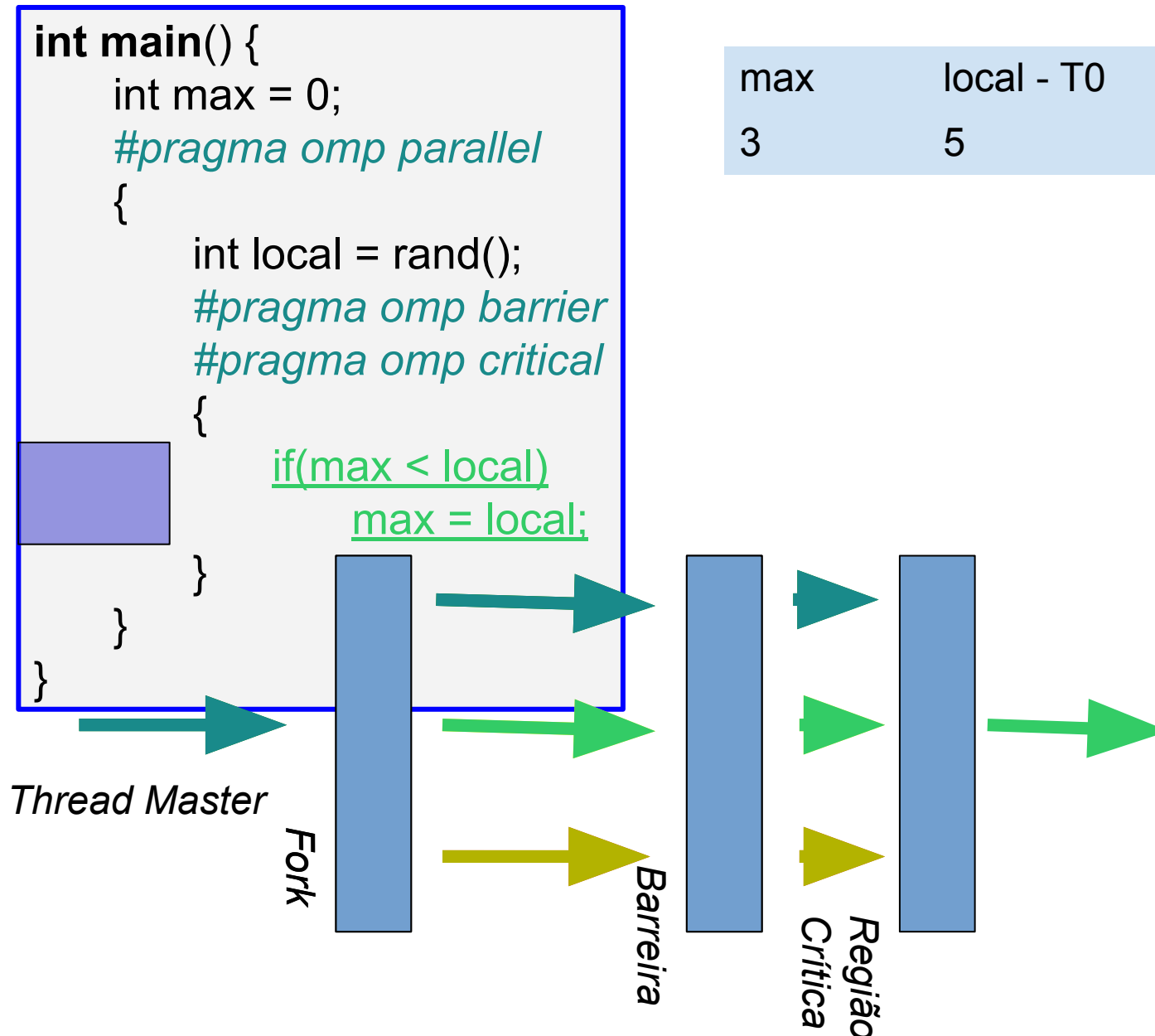
Diretiva “omp critical”



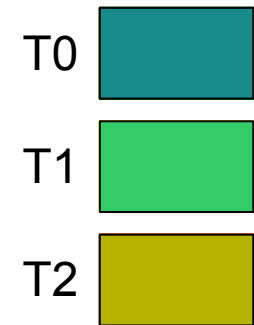
max	local - T0	local - T1	local - T2
0	5	3	7



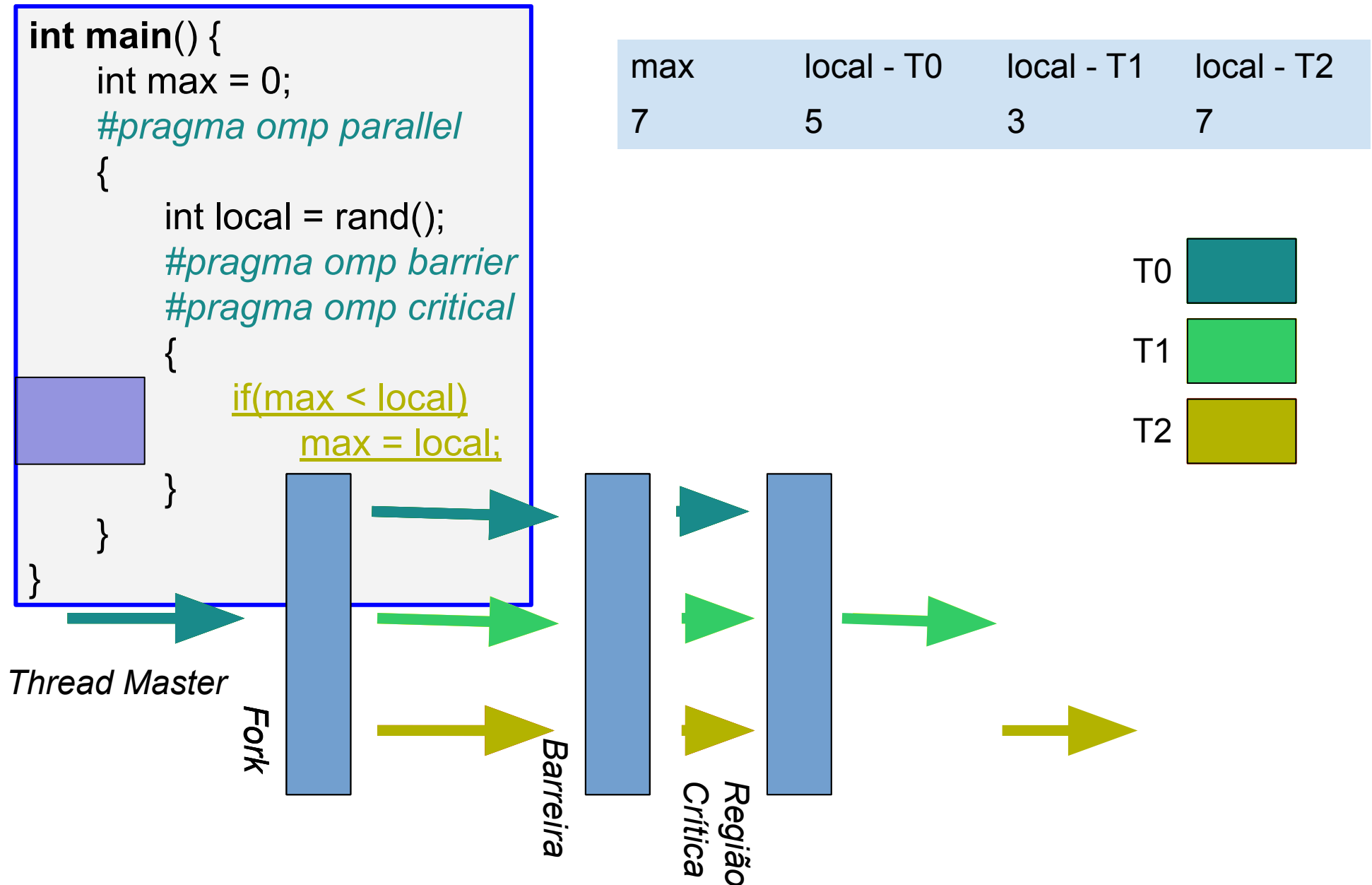
Diretiva “omp critical”



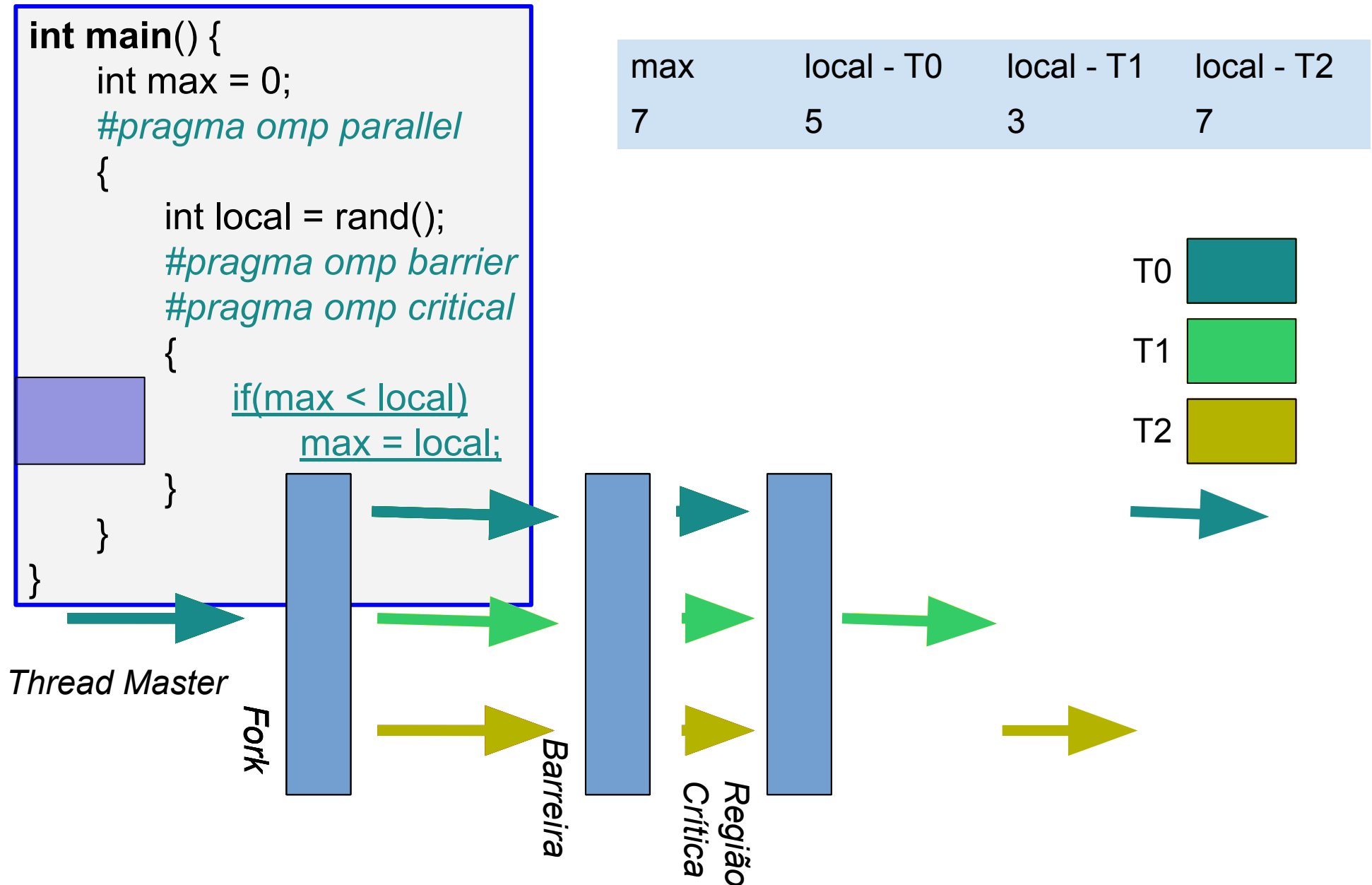
max	local - T0	local - T1	local - T2
3	5	3	7



Diretiva “omp critical”



Diretiva “omp critical”



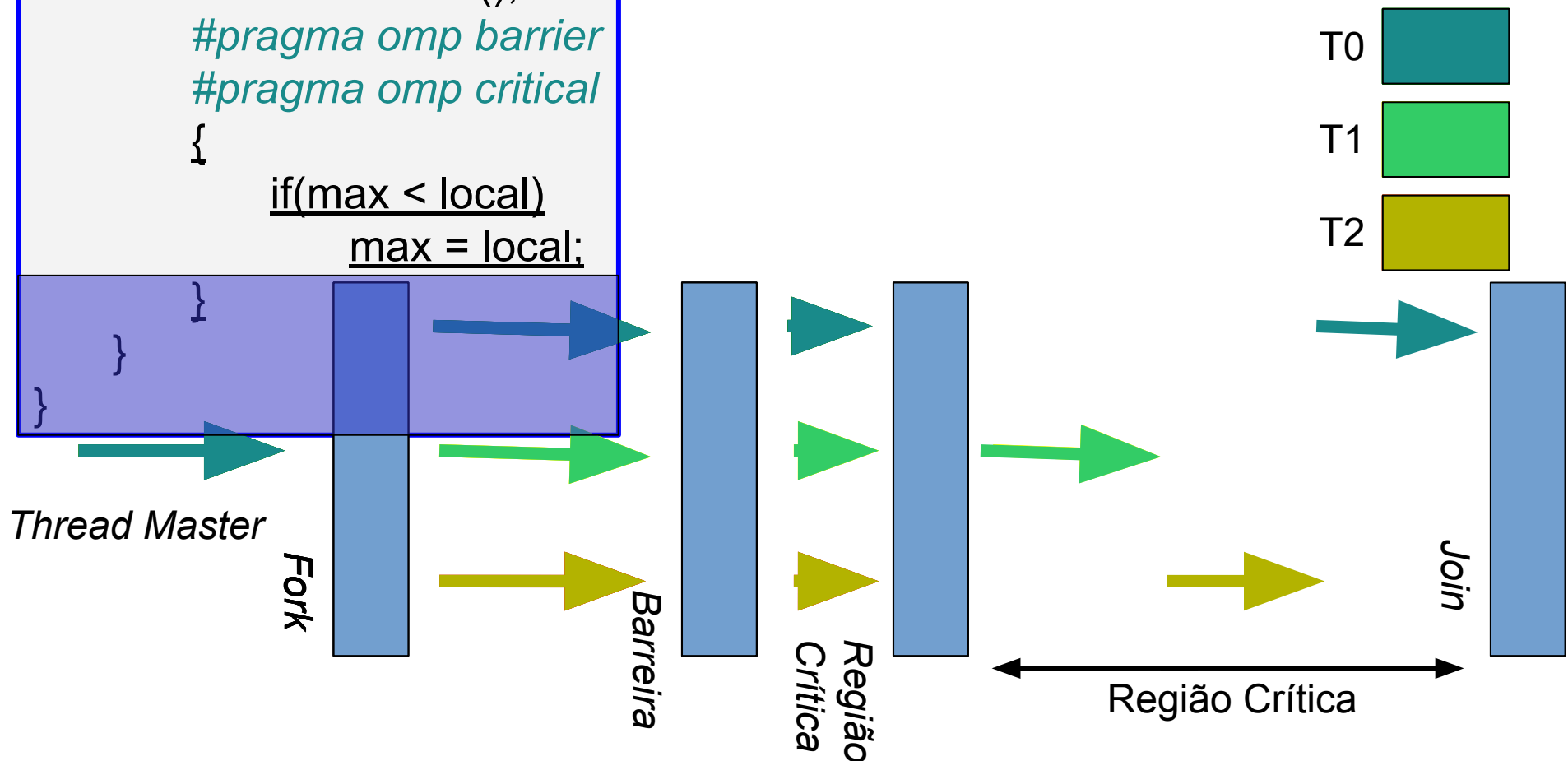
Diretiva “omp critical”

```

int main() {
    int max = 0;
    #pragma omp parallel
    {
        int local = rand();
        #pragma omp barrier
        #pragma omp critical
        {
            if(max < local)
                max = local;
        }
    }
}

```

max	local - T0	local - T1	local - T2
7	5	3	7



Exercício Resolvido (14)

- Paralelize o código abaixo usando a diretiva “*omp critical*”. Em seguida, apresente o *speedup* e a eficiência obtidos com a paralelização

```
int atividade(int **mat, int SIZE, int x){  
    int i, j, cont;  
    for(i = 0; i < SIZE; i++)  
        for(j=0; j < SIZE; j++)  
            if (mat[i][j] == x)  
                cont++;  
    return cont;  
}
```

Exercício Resolvido (15)

- Repita o Exercício Resolvido (12), resolvendo o problema existente no mesmo usando a diretiva “*omp critical*”. Em seguida, apresente o *speedup* e a eficiência obtidos com a paralelização

- Introdução
- Conceitos
- **OpenMP**
 - Sobre o OpenMP
 - Compilador gcc for Linux
 - Diretivas parallel e parallel for
 - Mais conceitos básicos
 - Diretivas sections, barrier, master e critical
 - **Escopo das Variáveis**
 - Cláusula Reduction
- Algoritmo
- Algoritmo

Escopo das Variáveis

```
int main() {  
    int x, y;  
    #pragma omp parallel  
    {  
        int z;  
    }  
}  
x: compartilhada  
y: compartilhada  
z: privada
```

```
int main() {  
    int x, y;  
    #pragma omp parallel private (x)  
    {  
        int z;  
    }  
}  
x: privada  
y: compartilhada  
z: privada
```

```
int main() {  
    int x, y;  
    #pragma omp parallel for  
    for(y=0; y < 5; y++)  
    {  
        int z;  
    }  
}  
x: compartilhada  
y: privada  
z: privada
```

- Introdução
- Conceitos
- **OpenMP**
 - Sobre o OpenMP
 - Compilador gcc for Linux
 - Diretivas parallel e parallel for
 - Mais conceitos básicos
 - Diretivas sections, barrier, master e critical
 - Escopo das Variáveis
 - **Cláusula Reduction**
- Algoritmo
- Algoritmo

Cláusula Reduction

- Aplicada em operações do tipo “*all-to-one*”
- Por exemplo, quando temos uma soma compartilhada, podemos usar a redução na variável soma, permitindo em cada *thread* tenha uma cópia local da variável e quando saímos da seção paralela, as somas locais serão automaticamente adicionadas na variável

Cláusula Reduction

Versão Serial

```
int i, sum = 0;
for(i = 0; i < N; i++){
    sum+= rand();
}
```

Versão Paralela

```
int i, sum = 0;
#pragma omp parallel for
for(i = 0; i < N; i++){
    sum+= rand();
}
```

Versão Paralela Usando Reduction

```
int i, sum = 0;
#pragma omp parallel for reduction (+:sum)
for(i = 0; i < N; i++){
    sum+= rand();
}
```

Exercício Resolvido (16)

- Repita o Exercício Resolvido (12), resolvendo o problema existente no mesmo usando uma *cláusula reduction*. Em seguida, apresente o *speedup* e a eficiência obtidos com a paralelização

- Introdução
- Conceitos Básicos
- OpenMP
- **Algoritmo Odd Even Paralelo**
- Algoritmo Quicksort Paralelo

Algoritmo *Odd Even* Sequencial

- Derivado do *Bubble sort*
- Dividido por duas fases: *odd* (ímpar) e *even* (par)
- O **laço externo** executa n vezes e quando i é ímpar, estamos na fase *odd*, caso contrário, na fase *even*
- Dentro do laço externo há dois internos. Para cada fase é executado apenas um dos laços internos e todas as comparações dentro desses laço é executado com diferentes elementos

Algoritmo *Odd Even* Sequencial

```
void oddEvenSort(int a[], int size){  
    for(int i = 0; i < size; i++) {  
        if(i%2 == 1)  
            for(int j = 1; j < size-1; j += 2){  
                if(a[j] > a[j+1]){  
                    swap(j, j+1);  
                }  
            }  
        else  
            for(int j = 0; j < size-1; j += 2){  
                if(a[j] > a[j+1]){  
                    swap(j, j+1);  
                }  
            }  
    }  
}
```

Exercício Resolvido (17)

- Paralelize o *Odd Even*. Em seguida, apresente o *speedup* e a eficiência obtidos com a paralelização

Dica: Comparações da mesma fase, são concorrentes

- Introdução
- Conceitos Básicos
- OpenMP
- Algoritmo Odd Even Paralelo
- **Algoritmo Quicksort Paralelo**

Quicksort Sequencial

- Tem um elemento chamado pivô, gerado pela função **particao()**
- Após a partição, à esquerda do pivô teremos elementos menores ou iguais a ele e, à direita, maiores ou iguais ao mesmo
- O algoritmo tem duas chamadas recursivas sendo que a primeira recebe os elementos da esquerda e a outra os da direita

Quicksort Sequencial

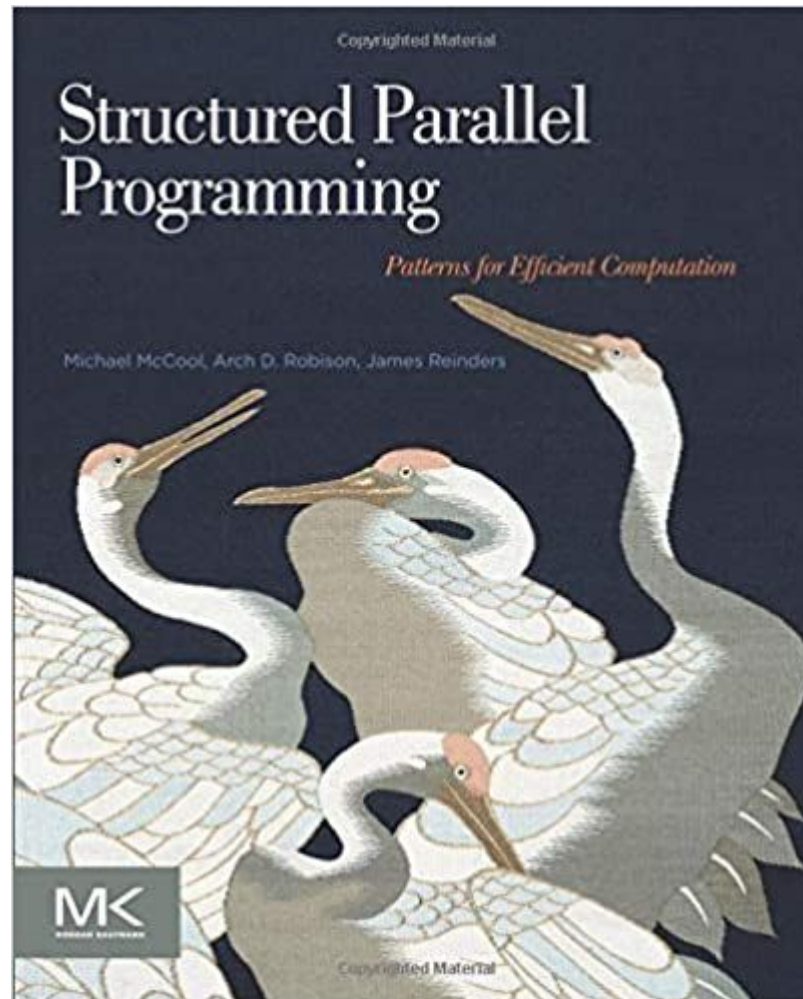
```
void quicksort(int vet[], int esq, int dir) {  
    int r;  
    if (dir > esq) {  
        pivo = particao(vet, esq, dir);  
        quicksort(vet, esq, pivo - 1);  
        quicksort(vet, pivo + 1, dir);  
    }  
}
```

Exercício Resolvido (18)

- Paralelize o *Quicksort*. Em seguida, apresente o *speedup* e a eficiência obtidos com a paralelização

Referências

- Michael McCool, James Reinders, Arch Robison, Structured parallel programming: patterns for efficient computation, Elsevier, 2012



Referências

- Andrew S. Tanenbaum, Herbert Bos, Sistemas Operacionais Modernos, Pearson, 4ª edição, 2015

