Tópicos Especiais Em Redes E Sistemas Distribuídos: Programação Paralela

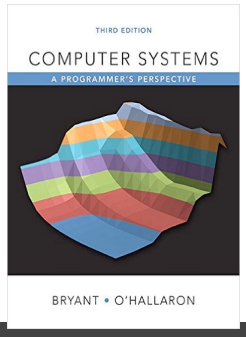
- Contas que você precisa criar
- Corretude na programação paralela

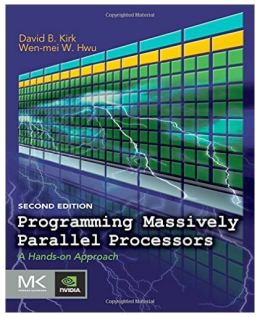
http://octopus-code.org/wiki/Main_Page

O Curso

De Programação paralela

Bibliografia básica





Contas que vai precisar

1. Udacity: https://br.udacity.com/course/intro-to-parallel-programming--cs344/
2. MPI: http://www2.lcad.cefetmg.br/acesso-cluster/
3. CUDA: conta na 200.131.37.139 porta 2200

TÓPICOS ESPECIAIS EM REDES E SISTEMAS DISTRIBUÍDOS: PROGRAMAÇÃO PARALELA

Corretude na programação paralela

Slides baseados em:

Mary Hall, CS 4230 - Parallel Programming, University of Utah.

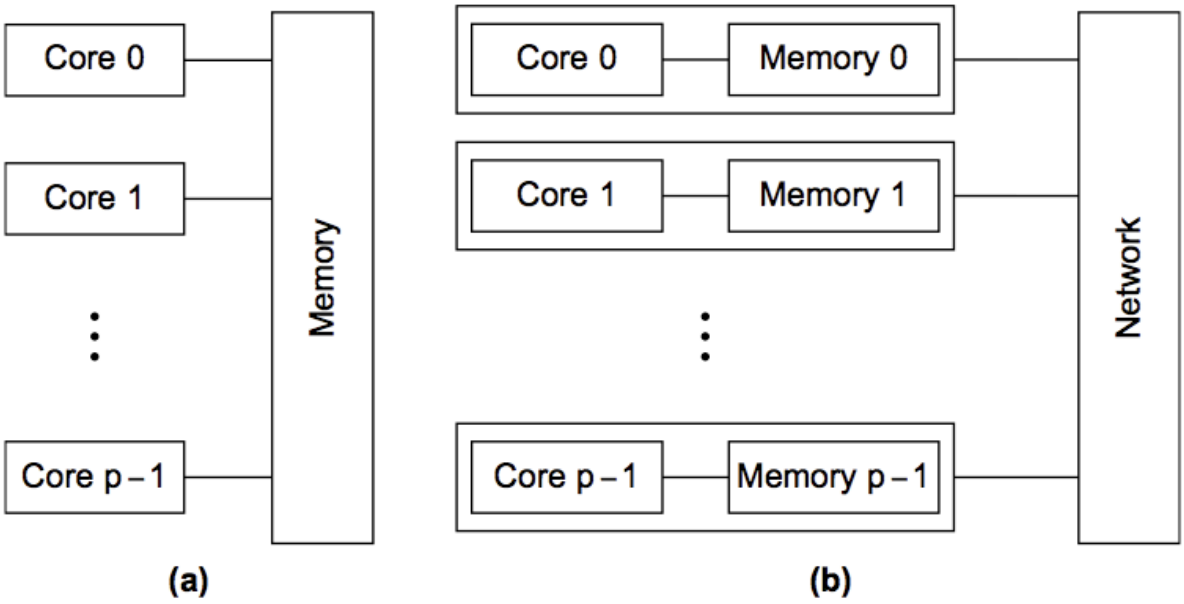
Fernando Silva , Parallel and Distributed Programming: foundations

Capítulo 1 livro: Pacheco

Memória compartilhada vs. distribuída

- **Sistema de memória compartilhada: os cores** podem compartilhar o acesso a memória
- Sistema de memória distribuída: cada core tem a sua própria memória privada, a comunicação entre cores é explícita , ex. Através de mensagens pela rede.

Memória compartilhada vs. distribuída



Programando Arquiteturas de Memória Distribuída

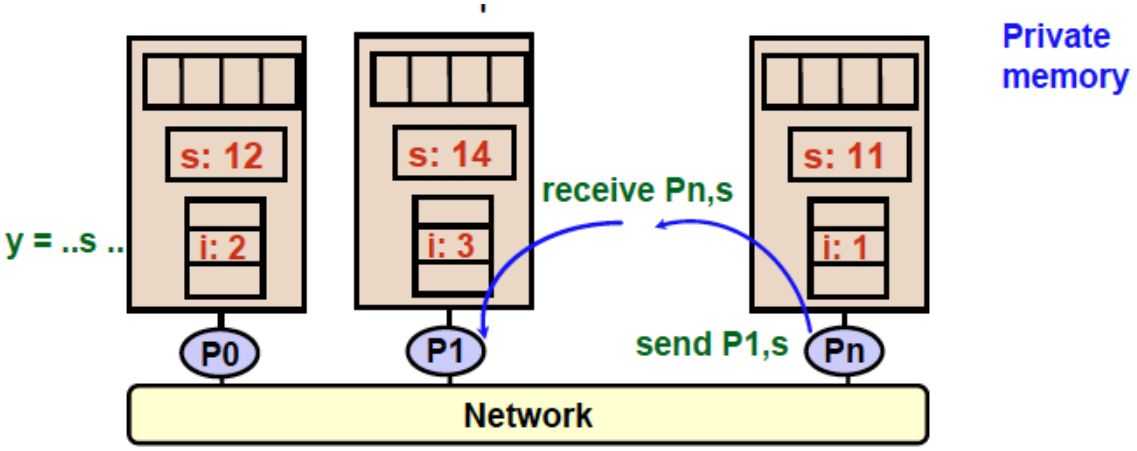
Um programa de memória distribuída consiste em processos com nome.

• Processo é uma thread de controle, mais espaço de endereços local: Não há dados compartilhados.

• Dados logicamente compartilhados são particionados entre os processos locais.

• Processos se comunicam por pares de envio / recebimento explícito

• A coordenação está implícita em cada evento de comunicação.



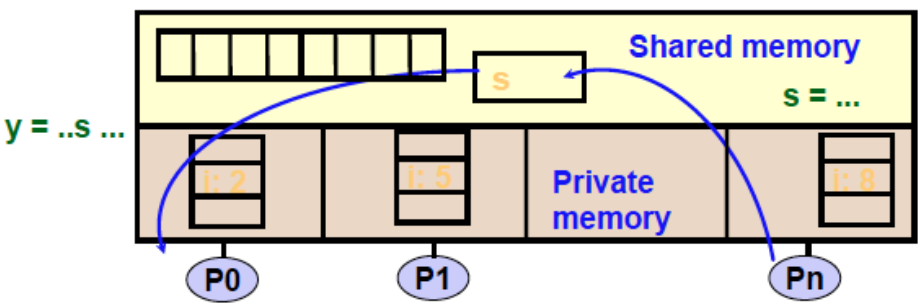
Programando Arquiteturas de memória compartilhada

Um programa de memória compartilhada é um conjunto de threads de controle. Threads são criadas no início do programa ou, eventualmente, de forma dinâmica.

Cada segmento tem:

- variáveis privadas, por exemplo, variáveis de pilha locais
- um conjunto de variáveis compartilhadas, por exemplo, variáveis estáticas, blocos comuns compartilhados, ou o heap global.

Threads se comunicam implicitamente escrevendo e lendo variáveis compartilhadas, e se coordenam através de construções de sincronização.



Visão: o que é uma Computação

Computação como seqüência de estados, com transições entre os estados que chamaremos de passos.

Operações atômicas: <e> Indivisíveis, executam até o final ou não executam. Não podem ser desmembradas, o estado não pode ser modificado no meio da execução, estados intermédios não são visíveis de fora do processo

Na maioria dos computadores atuais, referências a memória e atribuições (loads/stores) de palavras são atômicas. Operações de ponto flutuante não são.

Histórias

Estado de um programa concorrente: valores das variáveis do programa.
Processos executam sequências de instruções.
Instruções estão compostas por uma sequência de ações atômicas: ações indivisíveis que examinam ou mudam o estado (s_i) do programa.
Durante a execução do programa essas sequencias de ações atômicas são intercaladas. Cada execução de um programa concorrente produz uma história.
História (trace): Execução particular de um programa concorrente: $s_0 \xrightarrow[a_1]{} s_1 \xrightarrow[a_2]{} \ldots \xrightarrow[a_3]{} s_{n-1}$
(Onde s_i é o estado do programa, a_i uma ação atômica)

Conjunto de todas as histórias possíveis: $(mp)!/(m!)^p$

Concorrência e não determinismo

A execução sequencial é determinística: para a mesma entrada sempre a mesma saída. Computador como instrumento útil.

- DYSEAC, Univac, IBM 1953-1956: Paralelismo (abstrato) com 1 processador (por interrupção)
- Univac-LARC, Burroughs (1956 - 1962): Paralelismo real (multiprocessamento)
- O que mudou? Não determinismo!!

Histórias: exercício

x = 0

p	q
x++	x++

A instrução x++ é equivalente à sequência de instruções atômicas:
LOAD x //(temp = x)
ADD #1 //(temp = temp + 1)
SAVE x //(x = temp)

escreva todas as historias possíveis e o valor final da variável x em cada caso. Pode assumir que leituras e escritas nas variáveis são operações atômicas, e que a ordem das instruções dentro de cada thread é preservada no código gerado pelo compilador.

Condições de corrida (data races)

- Falha em que o resultado de um processo depende da sequência em que as instruções são executadas
- Exemplo: Duas threads executam o mesmo código. Qual deve ser o resultado final?

```
for (i = 0; i < niters; i++)
        cnt++;
```

Condições de corrida (data races)

- Código asm:

for (i = 0; i < niters; i++)
cnt++;

	movl (%rdi), %ecx	Head
	movl $0, %edx	
	cmpl %ecx, %edx	
	jge .L13	
.L11		
	movl cnt (%rip), %eax	L_i
	incl %eax	U_i
	movl %eax, cnt(%rip)	S_i
	incl %edx	Tail
	cmpl %ecx, %edx	
	jl .L11	
.L13		

	Thr	Instr	%eax1	%eax2	cnt
1	1	H1	-	-	0
2	1	L1	0	-	0
3	1	U1	1	-	0
4	1	S1	1	-	1
5	2	H2	-	-	1
6	2	L2	-	1	1
7	2	U2	-	2	1
8	2	S2	-	2	2
9	2	T2	-	2	2
10	1	T1	1	-	2

Condições de corrida (data races)

- Código asm:

	movl (%rdi), %ecx	Head
	movl $0, %edx	
	cmpl %ecx, %edx	
	jge .L13	
.L11		
	movl cnt (%rip), %eax	L_i
	incl %eax	U_i
	movl %eax, cnt(%rip)	S_i
	incl %edx	Tail
	cmpl %ecx, %edx	
	jl .L11	
.L13		

	Thr	Instr	%eax1	%eax2	cnt
1	1	H1	-	-	0
2	1	L1	0	-	0
3	1	U1	1	-	0
4	2	H2	-	-	0

Condições de corrida (data races)

- Código asm:

	movl (%rdi), %ecx	Head
	movl $0, %edx	
	cmpl %ecx, %edx	
	jge .L13	
.L11		
	movl cnt (%rip), %eax	L_i
	incl %eax	U_i
	movl %eax, cnt(%rip)	S_i
	incl %edx	Tail
	cmpl %ecx, %edx	
	jl .L11	
.L13		

	Thr	Instr	%eax1	%eax2	cnt
1	1	H1	-	-	0
2	1	L1	0	-	0
3	1	U1	1	-	0
4	2	H2	-	-	0
5	2	L2	-	0	0
6	1	S1	1	-	1

Condições de corrida (data races)

- Código asm:

	movl (%rdi), %ecx	Head
	movl $0, %edx	
	cmpl %ecx, %edx	
	jge .L13	
.L11		
	movl cnt (%rip), %eax	L_i
	incl %eax	U_i
	movl %eax, cnt(%rip)	S_i
	incl %edx	Tail
	cmpl %ecx, %edx	
	jl .L11	
.L13		

	Thr	Instr	%eax1	%eax2	cnt
1	1	H1	-	-	0
2	1	L1	0	-	0
3	1	U1	1	-	0
4	2	H2	-	-	0
5	2	L2	-	0	0
6	1	S1	1	-	1
7	1	T1	1	-	1
8	2	U2	-	1	1

Condições de corrida (data races)

- Código asm:

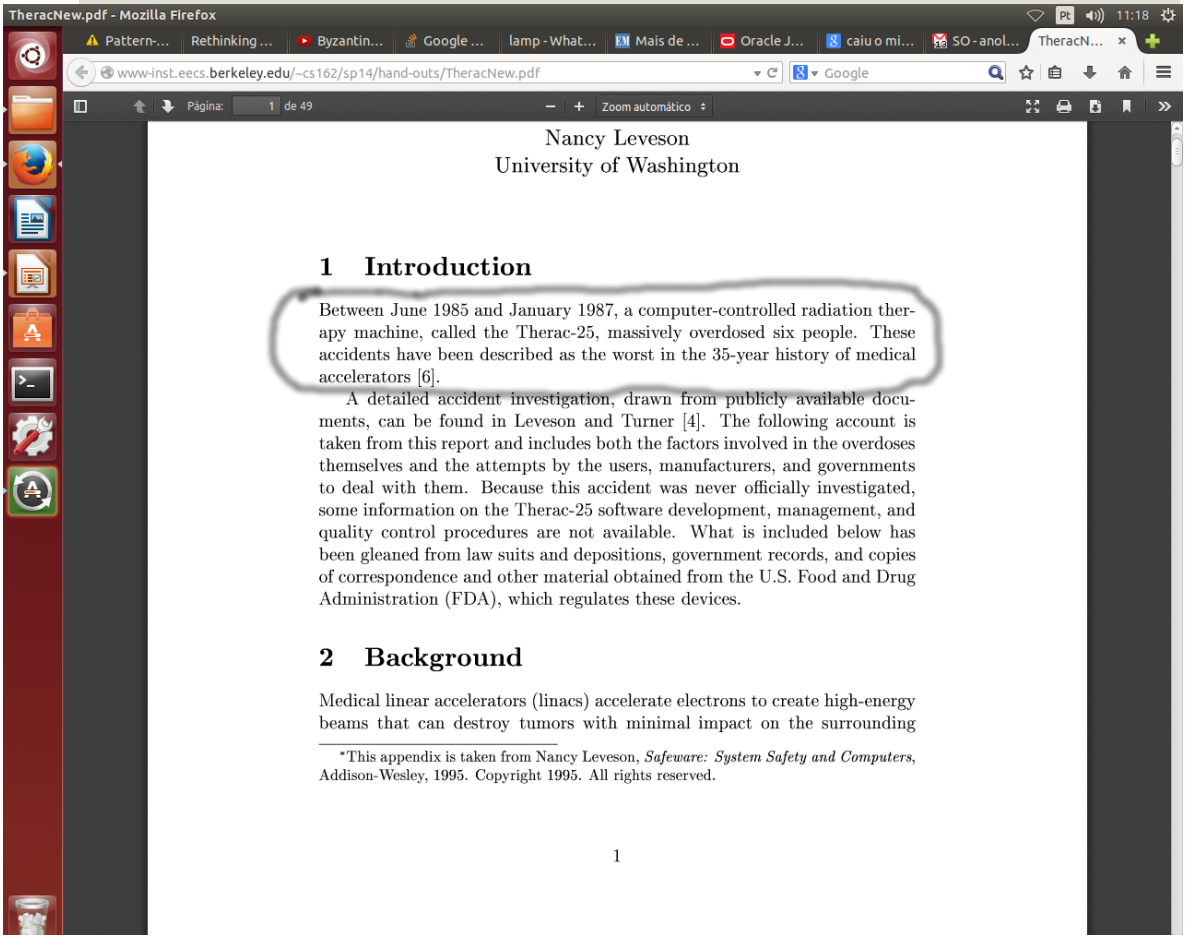
	movl (%rdi), %ecx	Head
	movl $0, %edx	
	cmpl %ecx, %edx	
	jge .L13	
.L11		
	movl cnt (%rip), %eax	L_i
	incl %eax	U_i
	movl %eax, cnt(%rip)	S_i
	incl %edx	Tail
	cmpl %ecx, %edx	
	jl .L11	
.L13		

	Thr	Instr	%eax1	%eax2	cnt
1	1	H1	-	-	0
2	1	L1	0	-	0
3	1	U1	1	-	0
4	2	H2	-	-	0
5	2	L2	-	0	0
6	1	S1	1	-	1
7	1	T1	1	-	1
8	2	U2	-	1	1
9	2	S2	-	1	1
10	2	T2	-	1	1

Não determinismo

- Lamport, 1986: ``A relação causal ente as operações lógicas é invalidada devido ao interleaving (intercalação) arbitrário das operações''

Sincronização: problema da exclusão mutua



Therac-25: Depuração em programas concorrentes não é garantia

- **Therac-25**, equipo de radioterapia médica:
- 1983, foram feitos análises de segurança no aparelho que excluiram qualquer falha de software, sob a premisa de: ``**Programming errors have been reduced by extensive testing on a hardware simulator and under field conditions on teletherapy units.** "; (Depuração nao é segura! Por que?)
- Junho 1985 e Janeiro 1987: 6 acidentes conhecidos envolvendo overdose massiva de radiação sob um conjunto específico de condições devido a erros no software (concorrente) de controle; (Corretude é imprescindível!)
- Natureza **esporádica** dos erros. (De novo, depuração não é segura!)

Corretude

Outro caso: **The Bug Heard 'round the World**," ACM SIGSOFT Software Engineering Notes, Vol 6 No 5, Oct 1981
http://www-inst.eecs.berkeley.edu/~cs162/sp14/hand-outs/garman_bug_81.pdf

"Learning from Mistakes— A Comprehensive Study on Real World Concurrency Bug Characteristics" ASPLOS 2008. http://www.cs.columbia.edu/~junfeng/09fa-e6998/papers/concurrency-bugs.pdf

(1) Almost all (97%) of the examined non-deadlock bugs belong to one of the two simple bug patterns: atomicity-violation or order-violation*

(2) About one third (32%) of the examined non-deadlock bugs are order-violation bugs, which are not well addressed in previous work.

(7) Almost all (97%) of the examined deadlock bugs involve two threads circularly waiting for at most two resources.

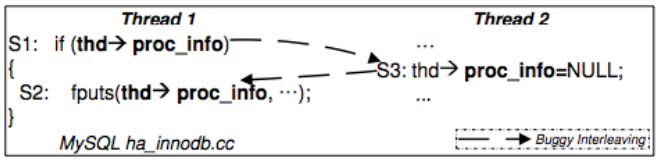


Figure 1. An atomicity violation bug from MySQL.

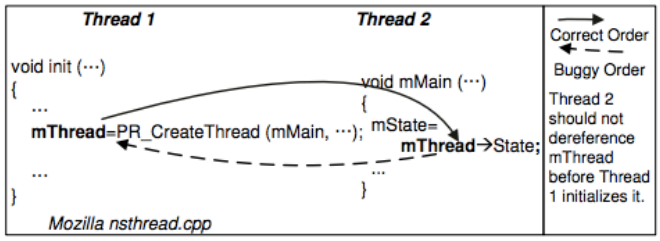


Figure 2. An order violation bug from Mozilla. The program fails to enforce the programmer's order intention: thread 2 should not read `mThread` until thread 1 initializes `mThread`. Note that, this bug could be fixed by making `PR_CreateThread` atomic with the write to `mThread`. However, our bug pattern categorization is based on root cause, regardless of possible fix strategies.

"Learning from Mistakes— A Comprehensive Study on Real World Concurrency Bug Characteristics" ASPLOS 2008. http://www.cs.columbia.edu/~junfeng/09fa-e6998/papers/concurrency-bugs.pdf

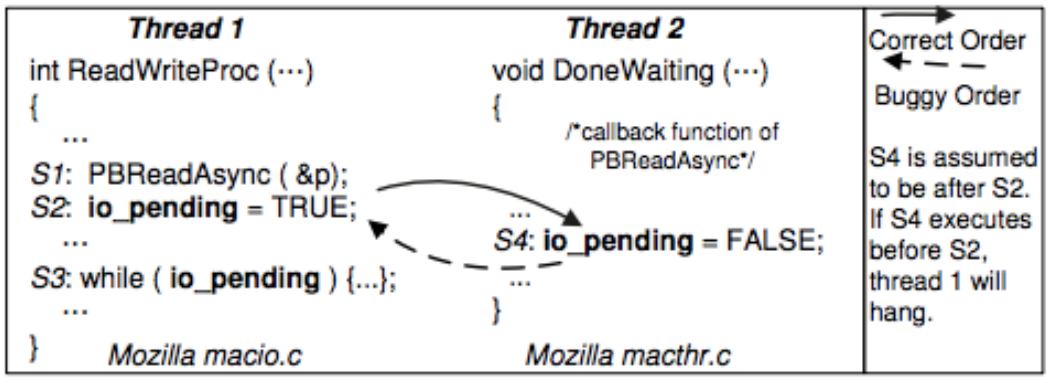


Figure 4. A write-write order violation bug from Mozilla. The program fails to enforce the programmer's order intention: thread 2 is expected to write `io_pending` to be FALSE some time after thread 1 initializes it to be TRUE. Note that, this bug could be fixed by making S1 and S2 atomic. However, our bug pattern categorization is based on root cause, regardless of possible fix strategies.

Sincronização em memoria compartilhada

- Melhor se não precisar compartilhar nada (sincronização é complexa e cara!)
- Senão:
  - Operações atómicas;
  - Exclusão mútua;
  - Primitivas de sincronização (semáforos e mutexes), algoritmos lock-free e wait-free;
  - **O mais importante! Projetar cuidadosa e incrementalmente seu programa**

Como resolver o problema: Exemplo simples (p. 4 do Pacheco)

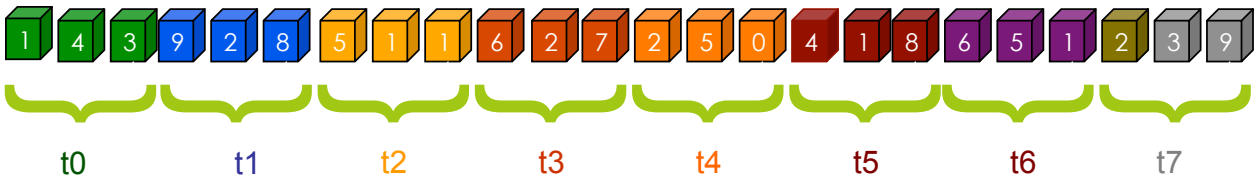
- Compute n values and add them together.
- Serial solution:

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

- Parallel formulation?

Version 1: Computation Partitioning

- Suppose each core computes a partial sum on n/t consecutive elements (t is the number of threads or processors)
- Example: n = 24 and t = 8, threads are numbered from 0 to 3




```
int block_length_per_thread = n/t;
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++)  {
    x = Compute_next_value(…);
    sum += x;
}
```

What Happened?

- Dependence on sum across iterations/threads
  - But reordering ok since operations on sum are associative
- Load/increment/store must be done *atomically* to preserve sequential meaning
- Definitions:
  - Atomicity: a set of operations is atomic if either they all execute or none executes. Thus, there is no way to see the results of a partial execution.
  - Mutual exclusion: at most one thread can execute the code at any time

Version 2: Add Locks

- Insert mutual exclusion (mutex) so that only one thread at a time is loading/incrementing/storing count atomically

```
int block_length_per_thread = n/t;
mutex m;
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++)  {
     my_x = Compute_next_value(…);
     mutex_lock(m);
     sum += my_x;
     mutex_unlock(m);
}
```

Correct now. Done?

Version 3: Increase Granularity

- Version 3:
  - Lock only to update final sum from private copy

```
int block_length_per_thread = n/t;
mutex m;
int my_sum;
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++)  {
     my_x = Compute_next_value(…);
     my_sum += my_x;
}
mutex_lock(m);
sum += my_sum;
mutex_unlock(m);
```


Version 4: Eliminate lock

- Version 4 (bottom of page 4 in textbook):
  - "Master" processor accumulates result

```
int block_length_per_thread = n/t;
mutex m;
shared my_sum[t];
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++)  {
     my_x = Compute_next_value(…);
     my_sum[id] += my_x;
}
if (id == 0) { // master thread
   sum = my_sum[0];
   for (i=1; i<t; i++) sum += my_sum[i];
}
```

Correct? Why not?

More Synchronization: Barriers

- Incorrect if master thread begins accumulating final result before other threads are done
- How can we force the master to wait until the threads are ready?
- Definition:
  - ***Barreira*** : Construção de sincronização entre múltiplas threads que garante que as threads esperam até todas chegarem na barreira antes da computação prosseguir. Depois disso todas continuam.
  - Usada to block threads from proceeding beyond a program point until all of the participating threads has reached the barrier.
  - Implementation of barriers?

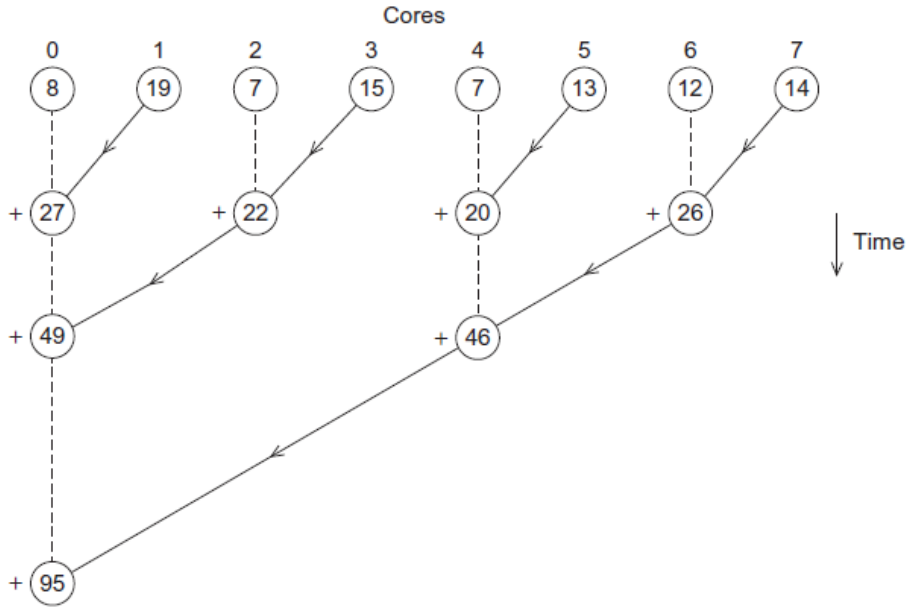
Version 5: Eliminate lock, but add barrier

- Version 5 (bottom of page 4 in textbook):
  - "Master" processor accumulates result

```
int block_length_per_thread = n/t;
mutex m;
shared my_sum[t];
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++)  {
     my_x = Compute_next_value(…);
     my_sum[t] += x;
}
Synchronize_cores(); // barrier for all participating threads
if (id == 0) { // master thread
   sum = my_sum[0];
   for (i=1; i<t; i++) sum += my_sum[t];
}
```

Now it's correct!

Version 6 (para casa): Multiple cores forming a global sum



Programação Paralela 2017.02