Uma breve história de Memória Transacional

Alexandro Baldassin (IGCE-UNESP)

email: alexandro.baldassin@unesp.br

Seminários em Computação, UFABC

14 de agosto de 2019

UNESP

- Present in 24 cities in the state of São Paulo

Created in 1976

Programs of studies
Undergraduate programs: 179
Graduate programs: 118
Research master degrees: 117
Doctorate degrees: 93

Students
Undergraduate: 35,666
Graduate: 11,043

Teaching staff
3,553 (93% with P.h.D.)

CS Graduate Program

- The graduate program in CS was created in 2006 (Master program), 2018 (PhD program)
  - Faculty: 23 professors
- Multicampus
  - Sao Jose do Rio Preto, Rio Claro, Bauru, Presidente Prudente
- Areas
  - Distributed Systems & Computer Architecture
  - Software Engineering & Data Base Systems,
  - Image Processing & Computer Vision
  - Mathematics & Computational Intelligence

Sobre o palestrante

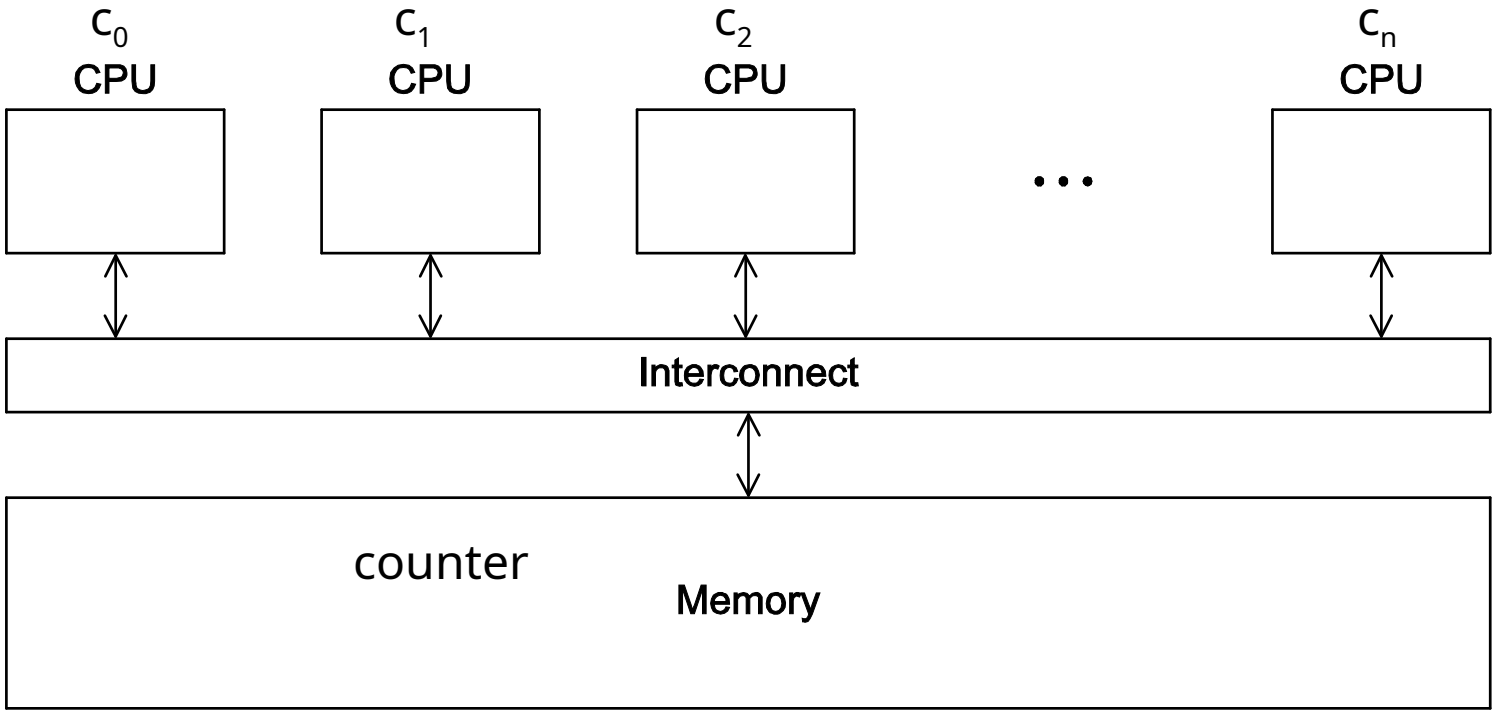
- Mestrado e doutorado no Instituto de Computação, UNICAMP (2005, 2009)
- Microsoft Research (2008 e 2010)
  - STM.NET
  - Revisions
- Professor no IGCE/DEMAC, UNESP – Rio Claro desde 2010 (atualmente professor associado)
- Área de pesquisa: Arquitetura de Computadores e Paralelismo

Programação paralela

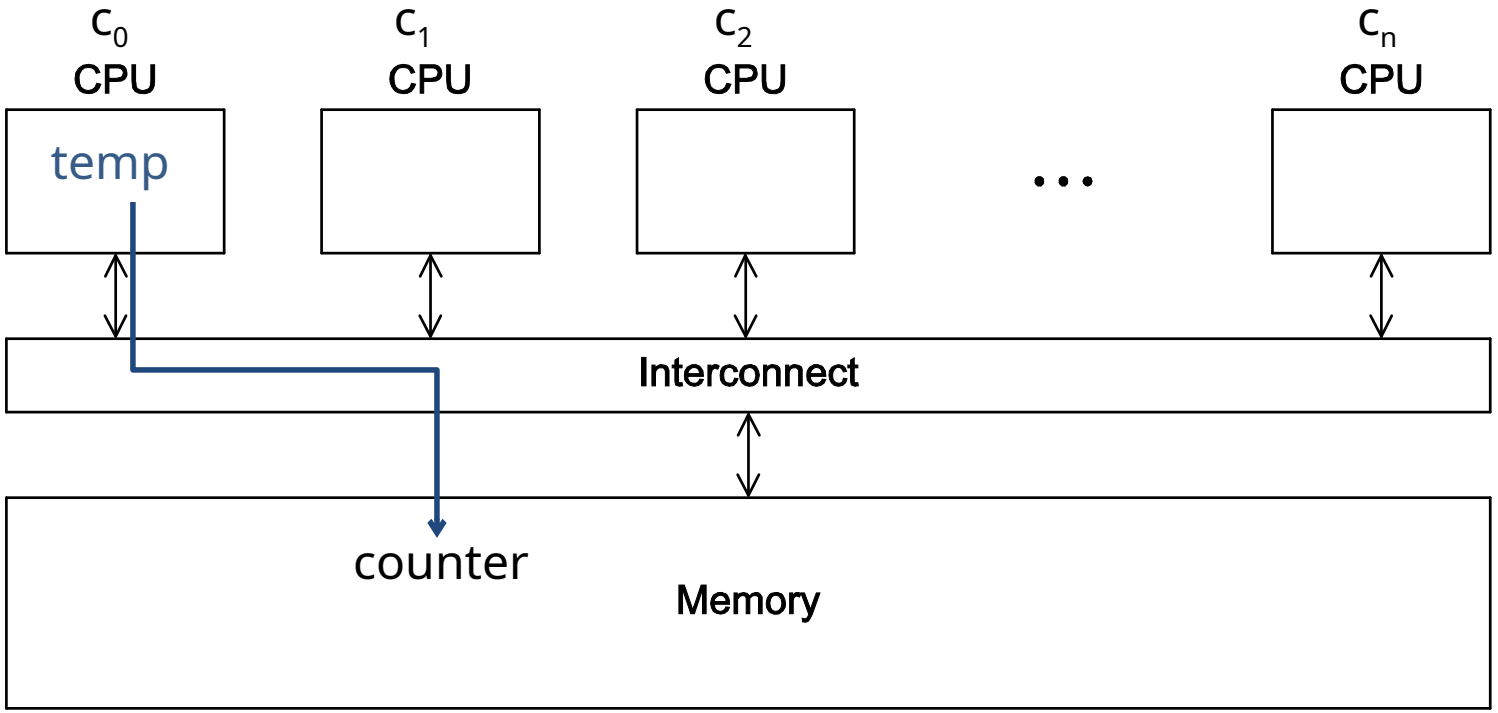
- Dois modelos principais (relativos à comunicação)
  - Passagem de mensagem (primitivas *send* e *receive*)
  - Memória compartilhada

- Memória compartilhada
  - Processo cria várias threads de execução
  - Threads compartilham espaço de endereçamento
  - Comunicação é feita diretamente através de leituras e escritas em memória compartilhada
    - Acessos concorrentes de leitura e escrita podem causar inconsistências (condições de corrida)
    - Sincronização é usada para evitar tais cenários

Sistema de memória compartilhada

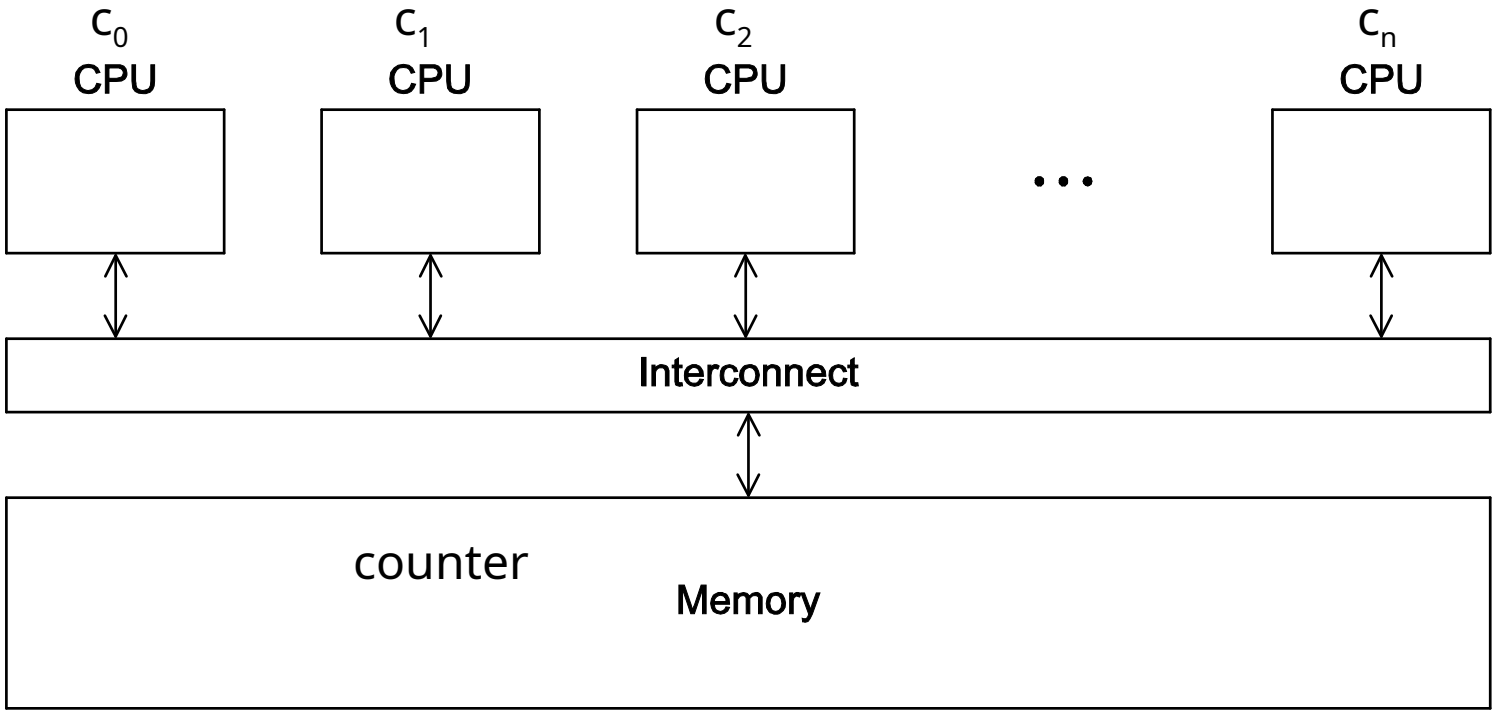


Sistema de memória compartilhada

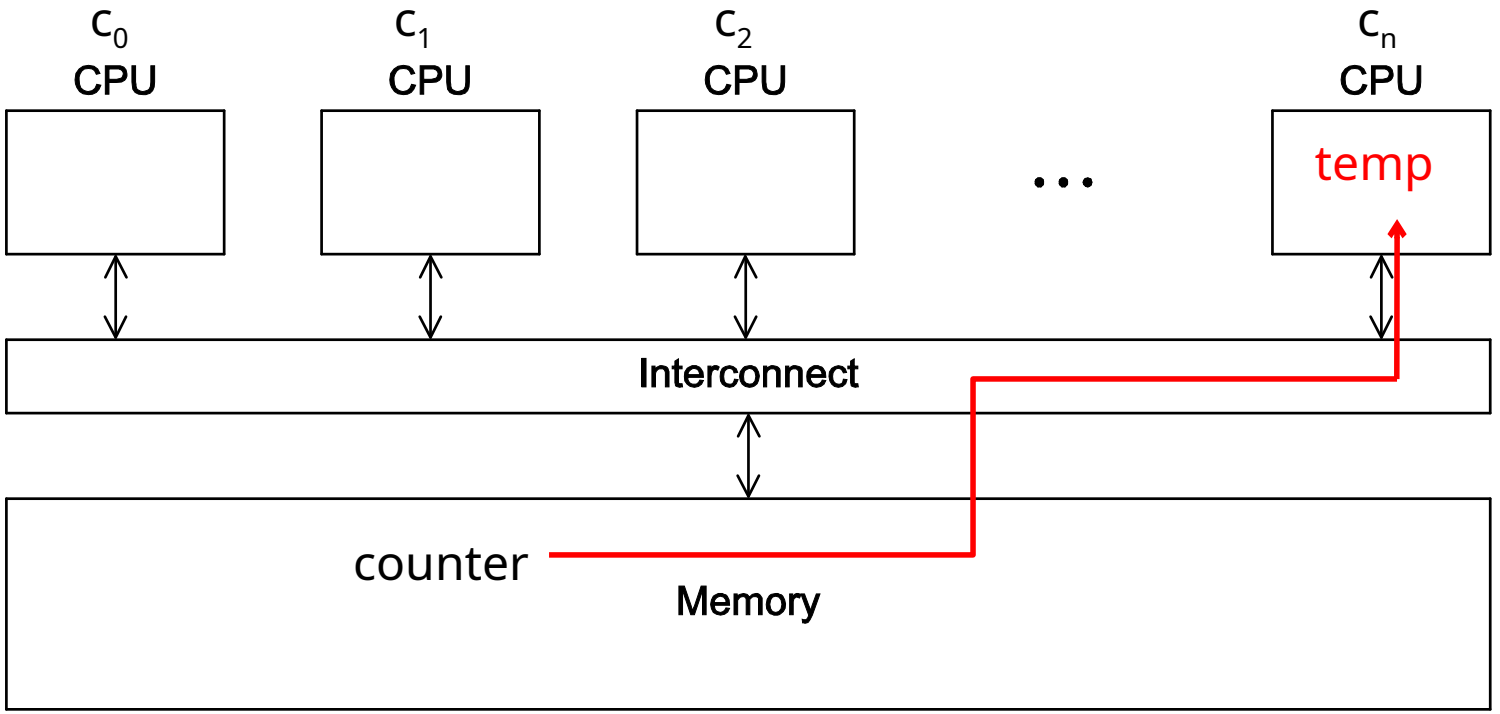


store (counter, temp)

Sistema de memória compartilhada

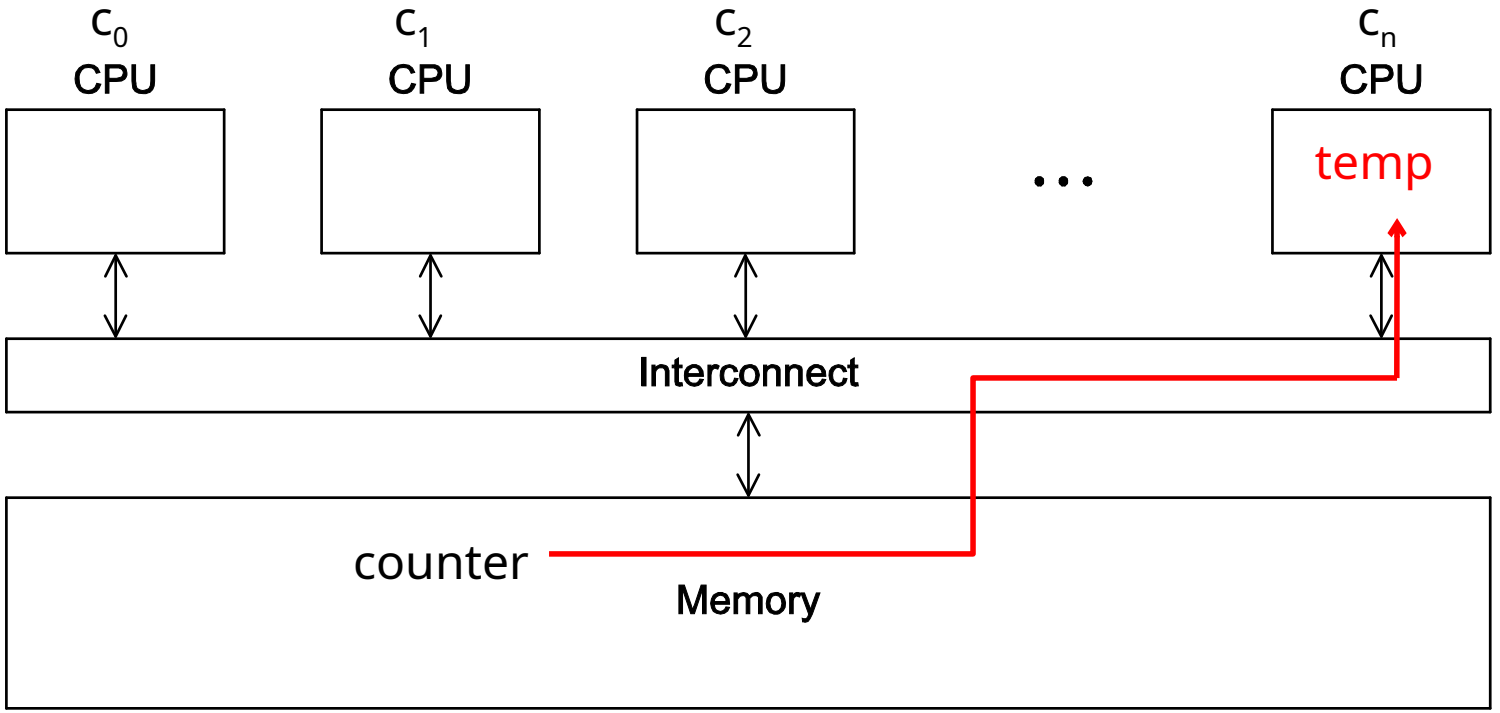


Sistema de memória compartilhada



load (temp, counter)

Sistema de memória compartilhada



- Usado como modelo de execução de Pthreads e OpenMP

Sincronização

- Necessária para evitar intercalações inconsistentes de execução

```
shared counter = 0;
```

Thread 1

```
void work()
{
  counter++;
}
```

Thread 2

```
void work()
{
  counter++;
}
```

Qual o resultado esperado após a primeira execução?

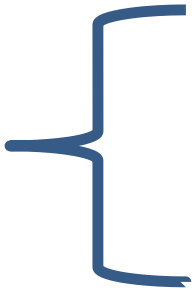
Sincronização

- Necessária para evitar intercalações inconsistentes de execução

```
shared counter = 0;
```

Thread 1

```
void work()
{
  counter++;
}
```



```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

Qual o resultado esperado após a primeira execução?

Sincronização

```
shared counter = 0;
```

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

Sincronização

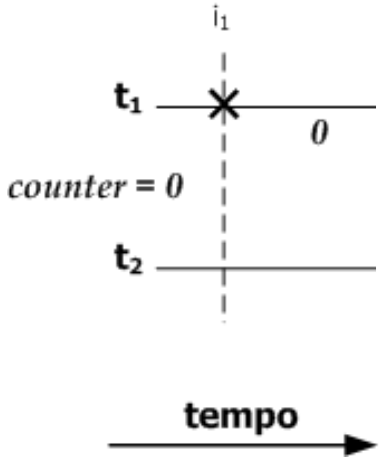
```
shared counter = 0;
```

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



Sincronização

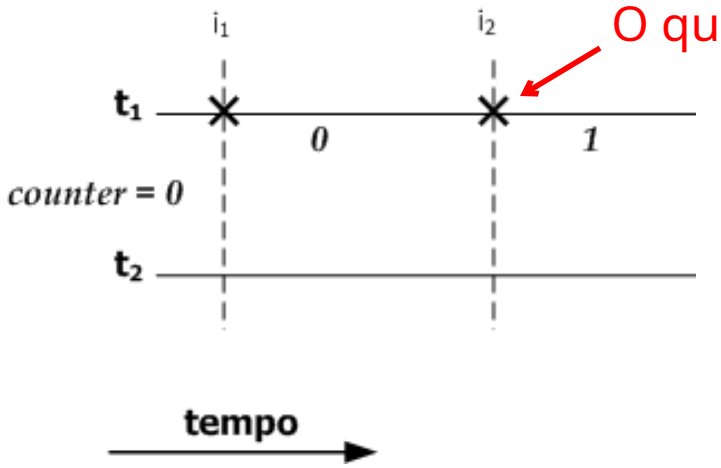
```
shared counter = 0;
```

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



O que ocorreu aqui?

Sincronização

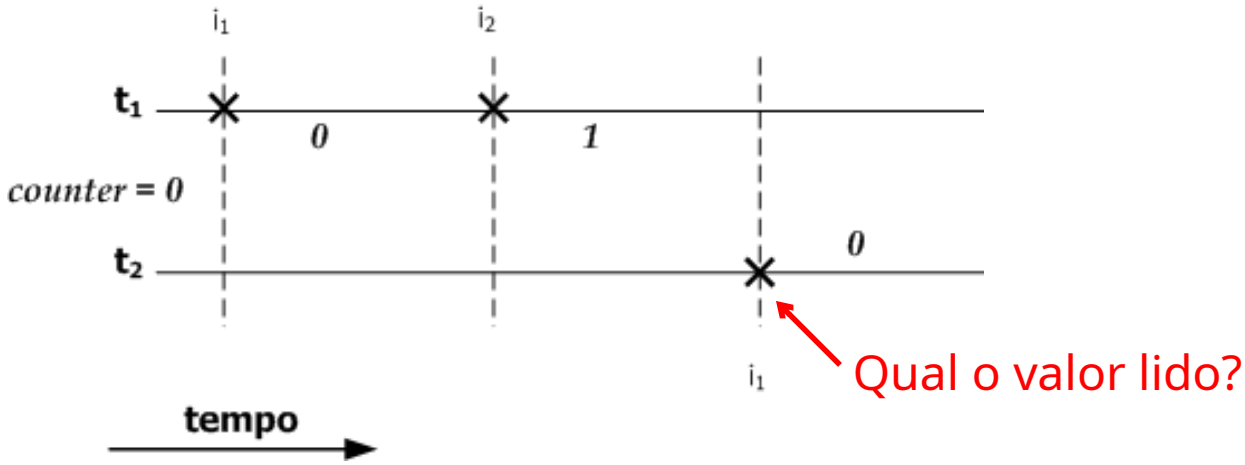
```
shared counter = 0;
```

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



Qual o valor lido?

Sincronização

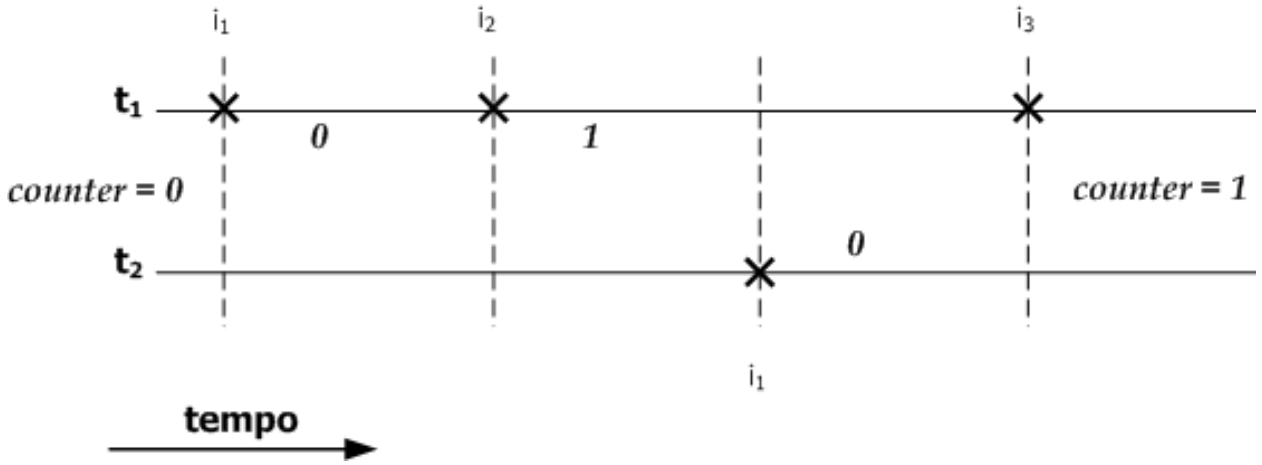
```
shared counter = 0;
```

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



Sincronização

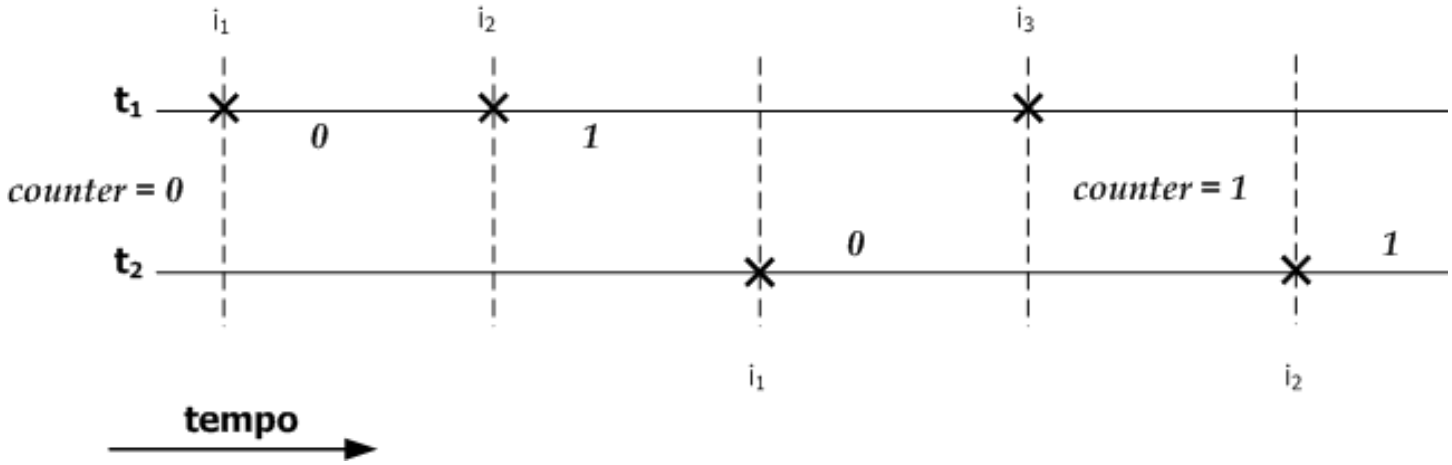
```
shared counter = 0;
```

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



Sincronização

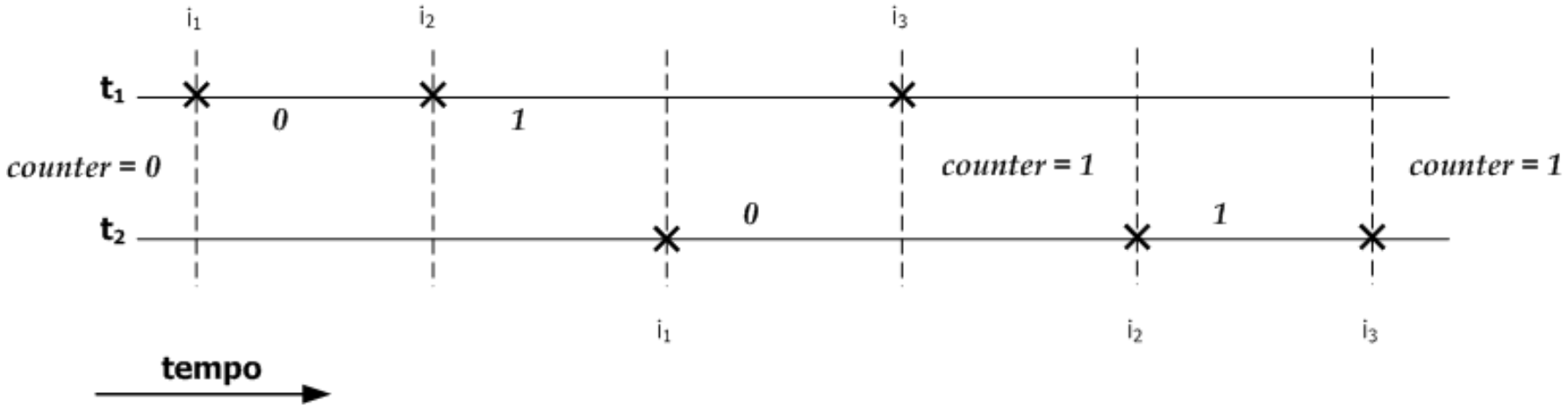
```
shared counter = 0;
```

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



Sincronização

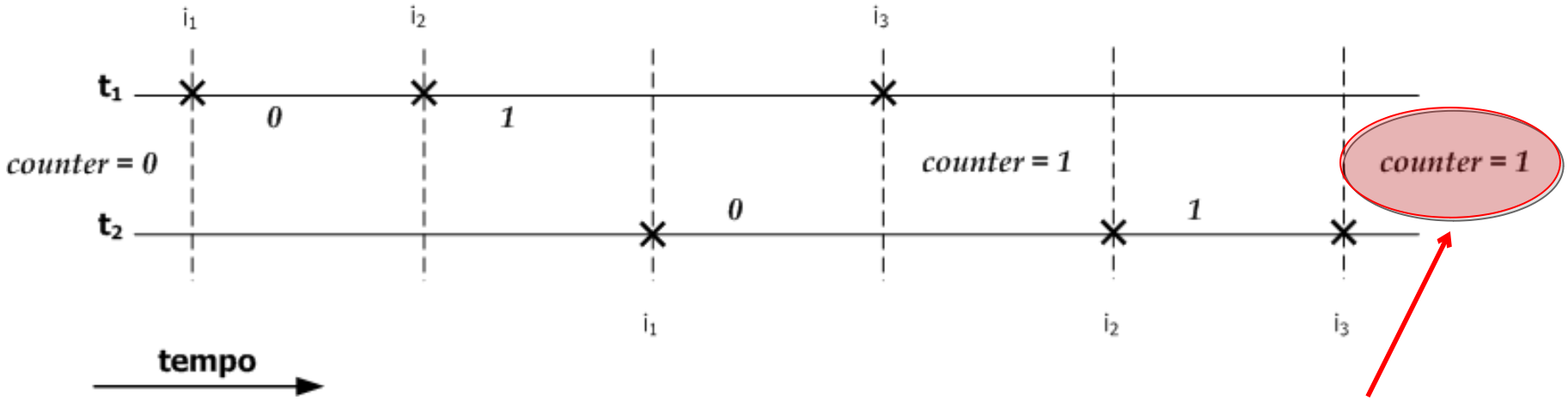
```
shared counter = 0;
```

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



Está correto?

Mecanismos de Sincronização

- Bloqueantes
  - bloqueios (*locks*) ←
  - variáveis de condição (*condition variables*)
  - semáforos/monitores
- Não-bloqueantes
  - livre de espera (*wait-free*)
  - livre de trava (*lock-free*)
  - livre de obstrução (*obstruction-free*)

Lista ordenada – bloqueio global

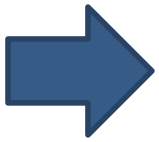
```
public boolean add(int item) {
   Node pred, curr;

   pred = head;
   curr = pred.next;
   while (curr.key < item) {
     pred = curr;
     curr = curr.next;
   }
   if (item != curr.key) {
     Node node = new Node(item);
     node.next = curr;
     pred.next = node;
     return true;
   }
   else return false;
}
```

Lista ordenada – bloqueio global

```
public boolean add(int item) {
   Node pred, curr;

   pred = head;
   curr = pred.next;
   while (curr.key < item) {
     pred = curr;
     curr = curr.next;
   }
   if (item != curr.key) {
     Node node = new Node(item);
     node.next = curr;
     pred.next = node;
     return true;
   }
   else return false;
}
```



```
public boolean add(int item) {
   Node pred, curr;
   boolean valid = false;

   lock.lock();
     pred = head;
     curr = pred.next;
     while (curr.key < item) {
       pred = curr;
       curr = curr.next;
     }
     if (item != curr.key) {
       Node node = new Node(item);
       node.next = curr;
       pred.next = node;
       valid = true;
     }
   lock.unlock();
   return valid;
}
```

Lista ordenada – bloqueio global

- Ideia do bloqueio global
  - Antes de iniciar o trecho de código que altera a lista, adquirir o bloqueio (**lock**)
  - Após trecho de código, liberar o bloqueio (**unlock**)
  - Funciona?
- Solução simples, mas não escala!
  - Operações são serializadas

Serializando

```
public boolean add(int item) {
   Node pred, curr;
   boolean valid = false;

   lock.lock();
     pred = head;
     curr = pred.next;
     while (curr.key < item) {
       pred = curr;
       curr = curr.next;
     }
     if (item != curr.key) {
       Node node = new Node(item);
       node.next = curr;
       pred.next = node;
       valid = true;
     }
   lock.unlock();
   return valid;
}
```

- Como melhorar a solução?
  - Sugestões?

```
public boolean add(int item) {
   Node pred, curr;
   boolean valid = false;

   lock.lock();
     pred = head;
     curr = pred.next;
     while (curr.key < item) {
       pred = curr;
       curr = curr.next;
     }
     if (item != curr.key) {
       Node node = new Node(item);
       node.next = curr;
       pred.next = node;
       valid = true;
     }
   lock.unlock();
   return valid;
}
```

Lista ordenada – bloqueios finos

- Antes de alterar um nó, uma *thread* necessita adquirir os bloqueios para o nó atual e o próximo
- Note que as threads envolvidas precisam adquirir os bloqueios na **mesma ordem** para evitar o risco de **deadlock**
- Não é trivial provar a corretude!
- Exemplo de código para a operação de inserção ...

Lista ordenada – bloqueios finos

```
public boolean add(int item) {
   boolean valid = false;
   head.lock();
     Node pred = head;
     Node curr = pred.next;
     curr.lock();
       while (curr.key < item) {
         pred.unlock();
         pred = curr;
         curr = curr.next;
         curr.lock();
       }
       if (item != curr.key) {
         Node newNode = new Node(item);
         newNode.next = curr;
         pred.next = newNode;
         valid = true;
       }
     curr.unlock();
   pred.unlock();
   return valid;
}
```

Lista ordenada – bloqueios finos

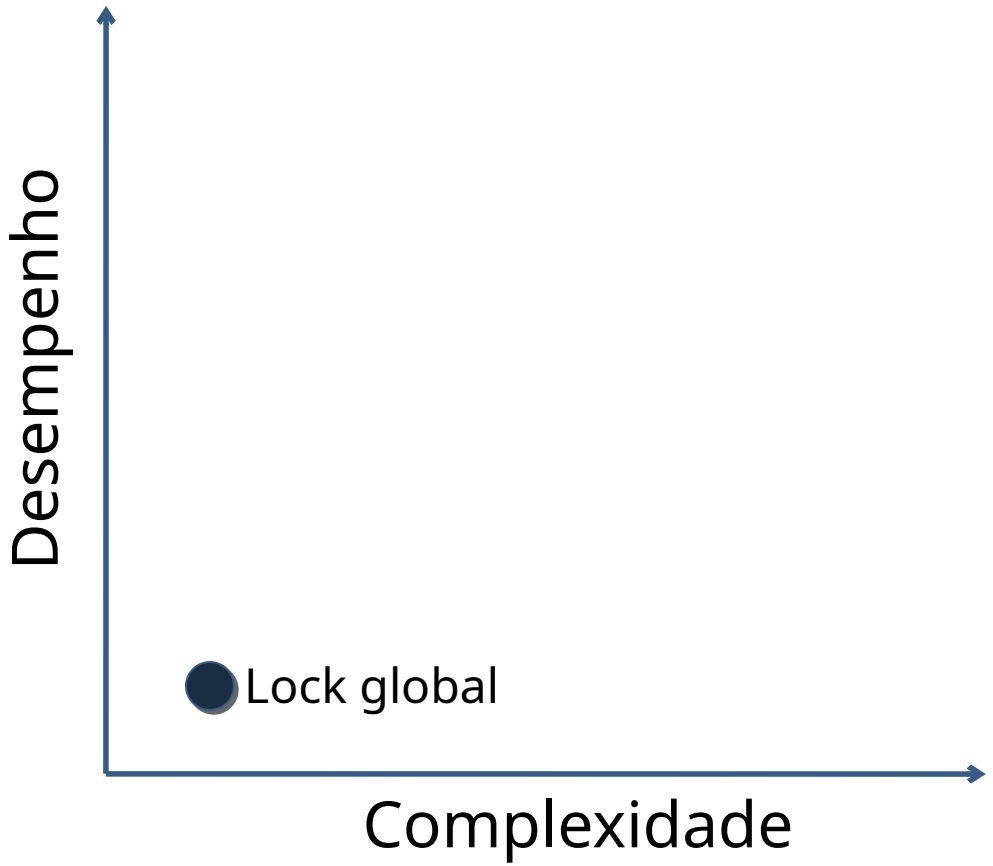
```
public boolean add(int item) {
   boolean valid = false;
   head.lock();
     Node pred = head;
     Node curr = pred.next;
     curr.lock();
       while (curr.key < item) {
         pred.unlock();
         pred = curr;
         curr = curr.next;
         curr.lock();
       }
       if (item != curr.key) {
         Node newNode = new Node(item);
         newNode.next = curr;
         pred.next = newNode;
         valid = true;
       }
     curr.unlock();
   pred.unlock();
   return valid;
}
```

Grande parte do código é específico para sincronização (6 de 18 linhas = ~33%)

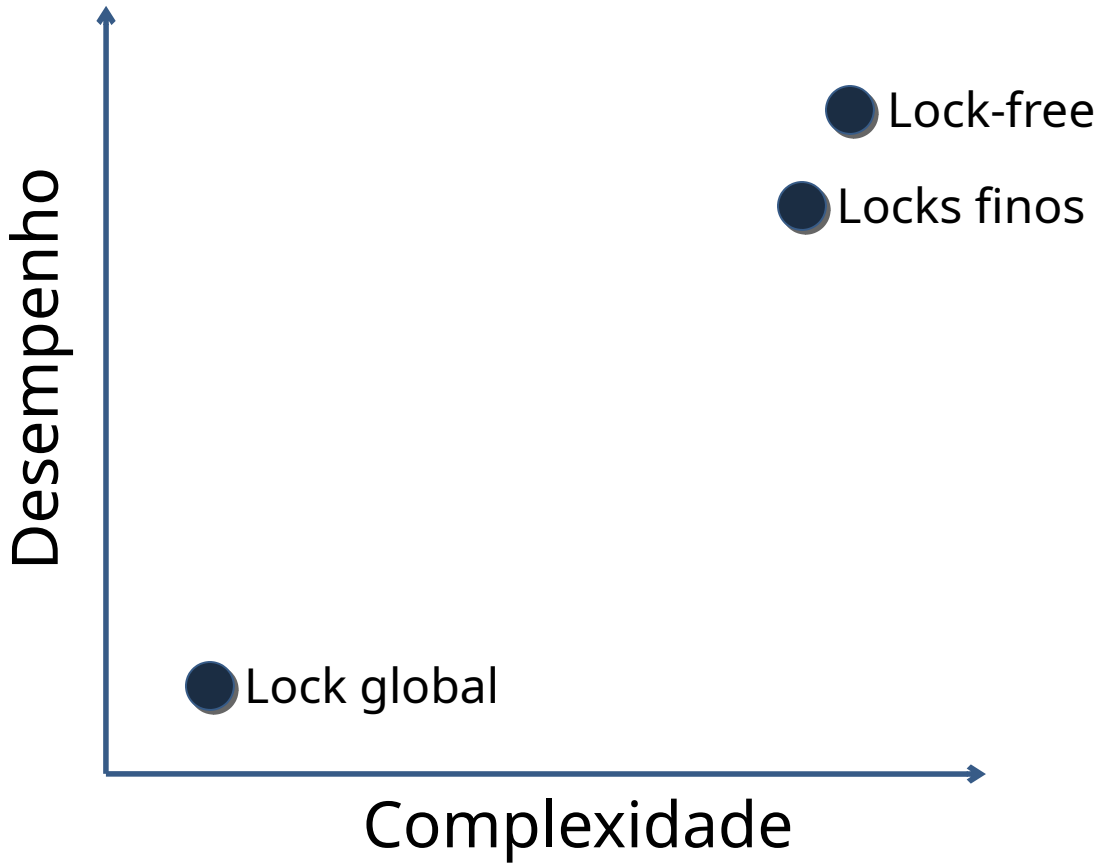
Problemas com bloqueios finos

- Risco alto de deadlock
  - Diferentes bloqueios adquiridos em diferentes ordens
- Operações **lock** e **unlock** custosas
  - Geralmente envolvem alguma forma de *syscall*
- Dificuldade relacionada a engenharia de software
  - Como encapsular um método com bloqueios?
  - Como compor código?

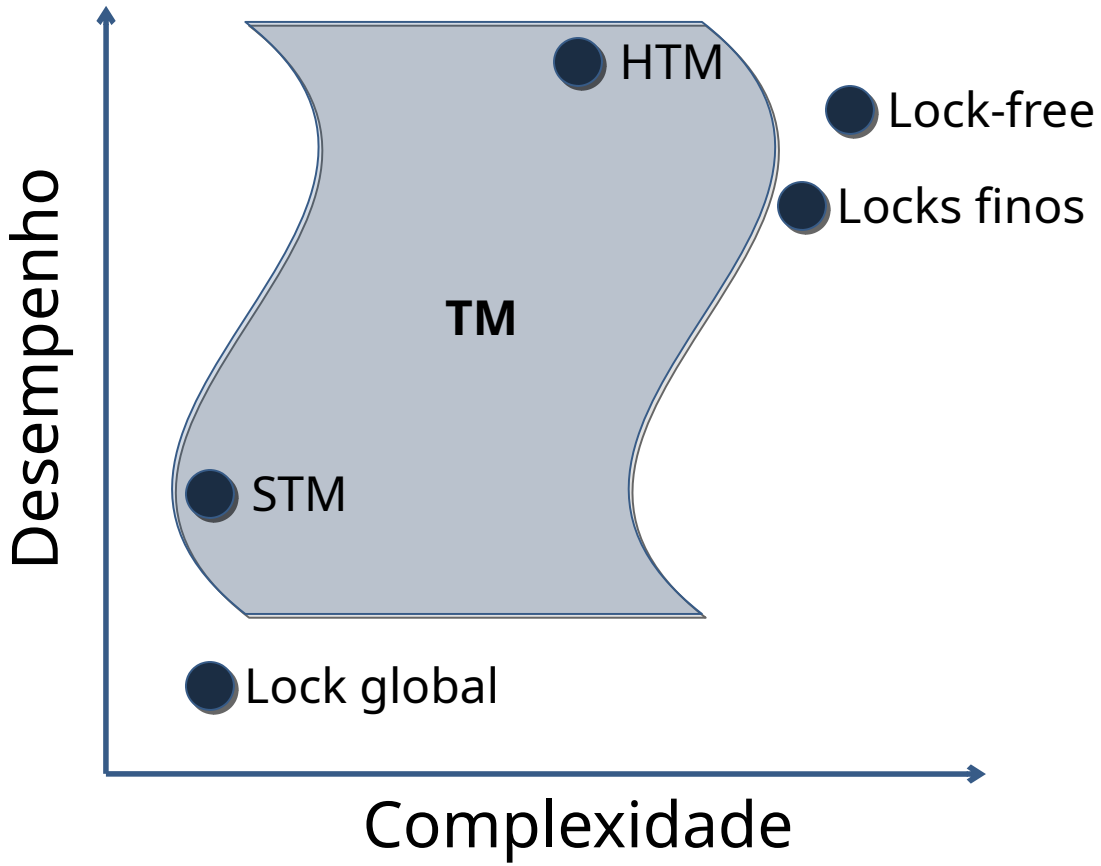
Programação concorrente



Programação concorrente



Programação concorrente



Ideia principal

- Similar ao modelo de transações em Banco de Dados

```
START TRANSACTION;

   UPDATE BankAccount SET Balance=Balance-5 WHERE Owner='James';
   UPDATE BankAccount SET Balance=Balance+5 WHERE Owner='Gord';

COMMIT;
```

Ideia principal

- Similar ao modelo de transações em Banco de Dados

```
START TRANSACTION;

   UPDATE BankAccount SET Balance=Balance-5 WHERE Owner='James';
   UPDATE BankAccount SET Balance=Balance+5 WHERE Owner='Gord';

COMMIT;
```

```
atomic {

  account("James").subtract(5);
  account("Gord").add(5);

}
```

reads and writes from/to
volatile memory

Memória Transacional (TM)

- No modelo transacional, programadores usam o conceito de **transação** como abstração
  - **A**tomicidade
  - **C**onsistência
  - **I**solamento

- Vantagens
  - Nível de abstração maior
  - Potencial ganho de desempenho
    - Dependente de implementação (visto mais adiante)
  - Composição de código

Memória Transacional (TM)

- No modelo transacional, programadores usam o conceito de **transação** como abstração
  - **A**tomicidade
  - **C**onsistência
  - **I**solamento
- Vantagens
  - Nível de abstração maior
  - Potencial ganho de desempenho
    - Dependente de implementação (visto mais adiante)
  - Composição de código

Detalhes de *como* realizar a sincronização são movidos do programador para o sistema de execução

Blocos atômicos

- O termo bloco atômico geralmente é usado quando o enfoque é sobre o suporte em linguagens
- Memória transacional (TM) é uma forma de se implementar blocos atômicos
- Nomenclatura ainda não consolidada!

Programando com blocos atômicos

- Programador delimita a região que deve ser executada atomicamente
  - Exemplo com lista ligada

```
public boolean add(int item) {
   Node pred, curr;
   boolean valid = false;

   atomic {
     pred = head;
     curr = pred.next;
     while (curr.key < item) {
       pred = curr;
       curr = curr.next;
     }
     if (item != curr.key) {
       Node node = new Node(item);
       node.next = curr;
       pred.next = node;
       valid = true;
     }
   }
   if (valid) return true;
   return false;
}
```

Programando com blocos atômicos

- Programador delimita a região que deve ser executada atomicamente
  - Exemplo com lista ligada

```
public boolean add(int item) {
   Node pred, curr;
   boolean valid = false;

   atomic {
     pred = head;
     curr = pred.next;
     while (curr.key < item) {
       pred = curr;
       curr = curr.next;
     }
     if (item != curr.key) {
       Node node = new Node(item);
       node.next = curr;
       pred.next = node;
       valid = true;
     }
   }
   if (valid) return true;
   return false;
}
```

- Sistema de execução (pode ser hardware ou software) cuida de garantir atomicidade, isolamento e consistência

Implementando blocos atômicos

- A implementação de blocos atômicos como transações usa dois conceitos principais

- **Versionamento de dados**
  - Os dados temporários (especulativos) usados pela transação precisam ser mantidos em algum local
  - Essencial para garantiar atomicidade e consistência

- **Isolamento da execução**
  - É necessário um mecanismo para detectar e resolver os conflitos entre transações

Implementação de TM

- O suporte transacional pode ser realizado em hardware, software ou uma mescla de ambos (híbrido)

- Hardware (HTM)
  - Melhor desempenho
  - Problemas com virtualização (espaço, tempo)

- Software (STM)
  - Desempenho depende muito da aplicação
  - Extremamente flexível
  - Ideal para testar novas ideias

Hardware versus Software

- Suporte em software geralmente é feito através de uma biblioteca
  - API típica
    - **TxStart**, **TxCommit**, **TxAbort**, **TxLoad**, **TxStore**
  - Exemplos: TinySTM, SwisSTM, NOrec
- Hardware geralmente adiciona novas instruções no ISA
  - Exemplo – Intel RTM
    - **XBEGIN <fallback-addr>**
    - **XEND**
    - **XABORT <cause>**

Exemplo: HW vs SW

```
atomic {
   x++;
}
```

Exemplo: HW vs SW

```
atomic {
   x++;
}
```

```
jmp_buf checkpoint;

setjmp(checkpoint);

TxStart(&checkpoint);

   int temp;
   temp = TxLoad((intptr_t *)(void *)&x);
   temp++;
   TxStore((intptr_t *)(void *)&x,
(intptr_t)temp);

TxCommit();
```

STM

Exemplo: HW vs SW

```
atomic {
   x++;
}
```

```
jmp_buf checkpoint;

setjmp(checkpoint);

TxStart(&checkpoint);

   int temp;
   temp = TxLoad((intptr_t *)(void *)&x);
   temp++;
   TxStore((intptr_t *)(void *)&x,
(intptr_t)temp);

TxCommit();
```

STM

```
if ((status = _xbegin()) ==
_XBEGIN_STARTED) {
   x++;
   _xend();
} else {
      // software fallback path
}
```

HTM

Principais eventos

1977	Lomet propõe ***atomic actions*** no contexto de estruturação de processos

Principais eventos

1977	Lomet propõe ***atomic actions*** no contexto de estruturação de processos
1993	Herlihy & Moss cunham o termo ***transactional memory***; no mesmo ano, um grupo da IBM propõe o Oklahoma update para o mesmo fim

Principais eventos

1977	Lomet propõe ***atomic actions*** no contexto de estruturação de processos
1993	Herlihy & Moss cunham o termo ***transactional memory***; no mesmo ano, um grupo da IBM propõe o Oklahoma update para o mesmo fim
1995	Shavit e Touitou criam a ***STM***

Principais eventos

1977	Lomet propõe ***atomic actions*** no contexto de estruturação de processos
1993	Herlihy & Moss cunham o termo ***transactional memory***; no mesmo ano, um grupo da IBM propõe o Oklahoma update para o mesmo fim
1995	Shavit e Touitou criam a ***STM***
2003	De forma independente, Herlihy et al. e Harris & Fraser propõem ***implementações eficientes*** de STM

Principais eventos

1977	Lomet propõe ***atomic actions*** no contexto de estruturação de processos
1993	Herlihy & Moss cunham o termo ***transactional memory***; no mesmo ano, um grupo da IBM propõe o Oklahoma update para o mesmo fim
1995	Shavit e Touitou criam a ***STM***
2003	De forma independente, Herlihy et al. e Harris & Fraser propõem ***implementações eficientes*** de STM
2011	Primeira ***implementação comercial de HTM***: IBM Blue Gene

Principais eventos

1977	Lomet propõe ***atomic actions*** no contexto de estruturação de processos
1993	Herlihy & Moss cunham o termo ***transactional memory***; no mesmo ano, um grupo da IBM propõe o Oklahoma update para o mesmo fim
1995	Shavit e Touitou criam a ***STM***
2003	De forma independente, Herlihy et al. e Harris & Fraser propõem ***implementações eficientes*** de STM
2011	Primeira ***implementação comercial de HTM***: IBM Blue Gene
2012	Intel anuncia ***TSX*** – IBM tambẽm fornece implementação de HTM na sua linha ***Power***

Principais eventos

1977	Lomet propõe ***atomic actions*** no contexto de estruturação de processos
1993	Herlihy & Moss cunham o termo ***transactional memory***; no mesmo ano, um grupo da IBM propõe o Oklahoma update para o mesmo fim
1995	Shavit e Touitou criam a ***STM***
2003	De forma independente, Herlihy et al. e Harris & Fraser propõem ***implementações eficientes*** de STM
2011	Primeira ***implementação comercial de HTM***: IBM Blue Gene
2012	Intel anuncia ***TSX*** – IBM tambẽm fornece implementação de HTM na sua linha ***Power***
2014	***GCC*** implementa suporte para TM

Resumo – HTM

- Nos últimos anos, especificações e implementações reais estão surgindo (Intel, IBM)
- Os primeiros resultados apontam que apenas o suporte em hardware não é suficiente
- Nota-se que o papel do software vai ser crucial para o bom desempenho (caminhos alternativos caso a transação não possa ser efetivada em hardware)
- Grande desafio: disponibilidade de aplicações transacionais

TM – Bom para quê?

- Estruturas de dados cuja escalabilidade é ruim com abordagens baseadas em bloqueios
  - Exemplo: árvore rubro-negra
- Aplicações nas quais o uso de bloqueio é muito conservativo
- Aplicações irregulares (uso extensivo de ponteiros)
  - Algoritmos baseados em grafos

Concluindo ...

- Memória Transacional (bloco atômico) está ganhando aceitação como um novo paradigma de programação paralela
  - Drafts para inclusão em C++ em preparação
- Duas grandes empresas na área lançaram processadores contendo extensões para TMs (IBM e Intel)
- Boas oportunidades para realização de pesquisa!

OBRIGADO!