

Uma brevíssima (e incompleta) introdução à memória transacional

MCZA020-13 - Programação Paralela

Emilio Francesquini

e.francesquini@ufabc.edu.br

2020.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC



- Estes slides foram preparados para o curso de **Programação Paralela na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Estes slides foram adaptados a partir do material elaborado pelo Prof. Mohamed Zahran da NYU.



Introdução

- **Memória Transacional** é uma técnica relativamente recente (e pouco conhecida) de se coordenar a execução de threads
- A ideia nasceu inspirada pelas transações de bancos de dados
- Enquanto em BDs transações tem as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade), transações em memórias transacionais oferecem **ACI**:
 - ▶ Atomicidade
 - ▶ Consistência
 - ▶ Isolamento

- Sistemas gerenciadores de bancos de dados (SGBDs) têm utilizado paralelismo por décadas
- SGBDs alcançam um bom desempenho na execução de múltiplas transações paralelas
- Cada um dos usuários de um banco de dados **não precisa se preocupar com condições de corrida**

- O modelo de programação para SGBDs é fortemente baseado em **transações**
- Transações são isoladas → para o usuário é como se ele fosse o único com acesso ao banco
 - ▶ Resultados de múltiplas transações simultâneas são indistinguíveis de execuções **serializadas** hipotéticas
- Talvez o ponto mais importante seja que **transações permitem execuções simultâneas e ainda assim garantem resultados coerentes com a utilização serializada**

- Memória pois o "BD" de programas paralelos e concorrentes é tipicamente a memória RAM
- Transacional pois oferece garantias ACI

Uma **transação** é uma sequência de ações que aparentam ser indivisíveis e instantâneas para um observador externo.

Propriedades ACID

- **Atomicidade:** Todas as ações de uma transação ou completam com sucesso ou o efeito de nenhuma delas é observável.
- **Consistência:** Dependem da aplicação.
- **Isolamento:** As ações sendo tomadas por uma transação não interferem na execução das ações das transações concorrentes.
- **Durabilidade:** Assim que uma transação finaliza, o resultado das suas ações se torna permanente.

Thread 1

```
1 begin_tx();  
2 A = A - 20;  
3 B = B + 20;  
4 A = A - B;  
5 C = C + 20;  
6 end_tx();
```

Leituras e escritas do Thread 1 às variáveis A, B e C são atômicas.

Thread 2

```
1 begin_tx();  
2 C = C - 30;  
3 A = A + 30;  
4 end_tx();
```

O Thread 2 enxerga apenas o estado anterior ou posterior (tudo ou nada) relativo à transação do Thread 1.

Inicialização

```
1 int x = 0;  
2 int y = 0;
```

Thread 1

```
1 atomic {  
2     int x = 42;  
3     int y = 42;  
4 }
```

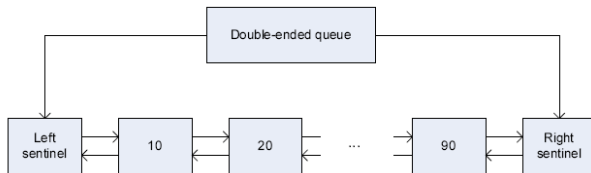
Thread 2

```
1 atomic {  
2     int tmp1 = x;  
3     int tmp2 = y;  
4 }
```

Qual é o valor correto para as variáveis `tmp1` e `tmp2`?

```
1 // Gerenciamento de transações
2 void StartTx();
3 void CommitTx();
4 void AbortTx();
5
6 //Acesso à memória
7 T ReadTX(T * addr);
8 void WriteTx(T *addr, T v);
```

Estes nomes são apenas ilustrativos. Cada implementação de TM usa nomes distintos.

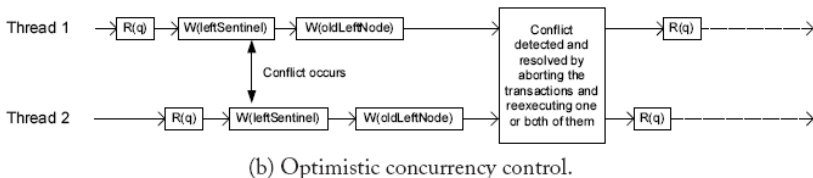
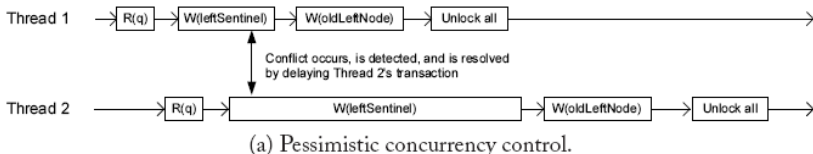


```
void PushLeft(DQueue *q, int val) {
    QNode *qn = malloc(sizeof(QNode));
    qn->val = val;
    QNode *leftSentinel = q->left;
    QNode *oldLeftNode = leftSentinel->right;
    qn->left = leftSentinel;
    qn->right = oldLeftNode;
    leftSentinel->right = qn;
    oldLeftNode->left = qn;
}
```

```
void PushLeft(DQueue *q, int val) {
    QNode *qn = malloc(sizeof(QNode));
    qn->val = val;
    do {
        StartTx();
        QNode *leftSentinel = ReadTx(&(q->left));
        QNode *oldLeftNode = ReadTx(&(leftSentinel->right));
        WriteTx(&(qn->left), leftSentinel);
        WriteTx(&(qn->right), oldLeftNode);
        WriteTx(&(leftSentinel->right), qn);
        WriteTx(&(oldLeftNode->left), qn);
    } while (!CommitTx());
}
```

Funcionamento de TM

- Um **conflito** ocorre quando ao menos duas transações efetuam ações que podem causar condições de corrida se executadas concorrentemente (Ex.: 2 escritas ou uma leitura e uma escrita)
- O conflito é **detectado** quando o sistema de TM percebe que houve um conflito
- O conflito é **resolvido** quando o sistema de TM toma alguma ação para resolver a situação (Ex.: atrasando a execução de uma das transações ou abortando uma das transações conflitantes)



Algumas implementações de TM usam um controle pessimista enquanto outras usam um controle otimista.

$R\downarrow / W\rightarrow$	Otimista	Pessimista
Otimista	TCC, TL2, SigTM	Intel C++STM Intel Java STM HASTM, Microsoft OSTM
Pessimista	LogTM	Intel C++ STM

- O que fazer com as escritas feitas até a transação efetuar um *commit*?
- **Versionamento Ávido** (= *eager version management*)
 - ▶ A transação modifica os dados na memória diretamente
 - ▶ Mantém um **undo log** com os dados sobreescritos
 - ▶ Requer um controle de concorrência pessimista
- **Versionamento Preguiçoso** (= *lazy version management*)
 - ▶ Atualizações são atrasadas até que sejam efetivadas
 - ▶ Transações mantêm cada uma um **redo log** com as escritas pendentes

- Com a abordagem pessimista é fácil, basta utilizar *locks*
- Com a abordagem otimista há vários pontos a se considerar
 - ▶ Granularidade do conflito: linha de cache, objeto, variável,...
 - ▶ Momento no qual a detecção ocorre
 - Quando a transação declara o interesse de acessar uma variável (*eager conflict detection*)
 - Na validação das operações: pode ocorrer muitas vezes durante a execução da transação
 - Na efetivação da transação: *lazy conflict detection*
 - ▶ Que tipo de acesso é tratado como conflito?
 - Entre transações concorrentes
 - Entre transações ativas e efetivadas

Thread 1

```
1 do {  
2     StartTx();  
3     WriteTx(&x, 1);  
4 } while (!CommitTx());
```

Thread 2

```
1 do {  
2     StartTx();  
3     int tmp_1 = ReadTx(&x);  
4     while (tmp_1 == 0) {}  
5 } while (!CommitTx());
```

- Ficar usando `WriteTx` e `ReadTx` é inconveniente e propenso a erros
- **Blocos atômicos** de código
 - ▶ O compilador gera automaticamente o `WriteTx` e os `ReadTX`
 - ▶ Disponível em várias linguagens de programação

```
void PushLeft(DQueue *q, int val) {
    QNode *qn = malloc(sizeof(QNode));
    qn->val = val;
    atomic {
        QNode *leftSentinel = q->left;
        QNode *oldLeftNode = leftSentinel->right;
        qn->left = leftSentinel;
        qn->right = oldLeftNode;
        leftSentinel->right = qn;
        oldLeftNode->left = qn;
    }
}
```

A principal vantagem de blocos atômicos em comparação à locks é a de que um bloco atômico não precisa identificar as variáveis que pretende acessar ou nomear a seção crítica.

Thread 1

```
1  __tm_atomic {  
2      t1 = foo();  
3  }
```

Thread 2

```
1  __tm_atomic {  
2      t2 = bar();  
3  }
```

Implementações de TM

- Existem duas principais abordagens
 - ▶ Software Transactional Memory (STM)
 - ▶ Hardware Transactional Memory (HTM)

Componentes:

- **descriptor transaccional**: mantém o estado de cada transação
- **undo-log ou redo-log**:
- **read-set ou write-set**: mantém um registro dos endereços de memória acessados (leitura ou escrita)

- O compilador instrumenta o código compilado para incluir o prólogo, epílogo e as chamadas de escrita e leitura
- O ambiente de execução mantém a lista dos acessos à memória, detecta conflitos e efetiva ou aborta a execução de transações

Antes

```
1 atomic {  
2     r = x;  
3     y = r + 1;  
4 }
```

Depois

```
1 td = getTxDesc();  
2 txBegin(td);  
3 r = txRead(td, &x);  
4 txWriteInt(td, &y, r + 1);  
5 txEnd(td);
```

Problema paralelizado

Algorithm 1 Conjugate Gradients

1: $r_0 = b - Ax_0$, $p_0 = r_0$, A spd

2: **for** $i = 0, 1, 2, \dots$ **do**

3: $\alpha_i = \frac{r_i^T r_i}{p_i^T A p_i}$

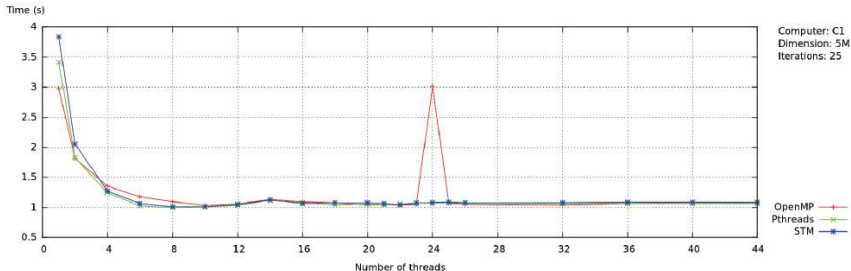
4: $x_{i+1} = x_i + \alpha_i p_i$

5: $r_{i+1} = r_i - \alpha_i A p_i$

6: $\beta_i = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$

7: $p_{i+1} = r_{i+1} + \beta_i p_i$

8: **end for**



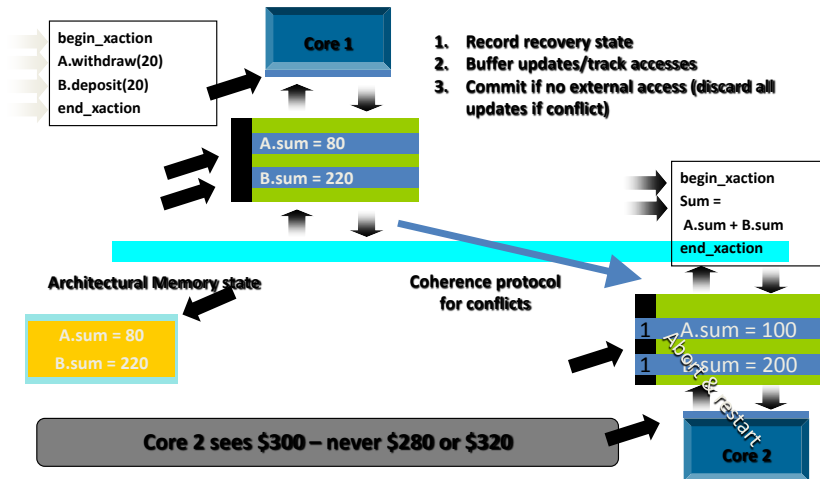
Disponível em três sabores:

- Implementação completa em hardware
- Implementação mista, com parte em hardware e parte em software
- Como extensões de hardware para acelerar a execução de STMs

O mecanismo de hardware precisa efetuar as seguintes tarefas:

- Identificar posições de memória com acesso transacionado
- Manter os read e write-sets das transações
- Detectar e resolver conflitos
- Cuidar do estado dos registradores
- Efetivar ou abortar transações

Buffering	Cache transaccional
Detecção de conflitos	Protocolo de coerência de cache
Abortar/Recuperar	Invalidar a linha de cache transaccional
Efetivar	Validar a linha de cache transaccional



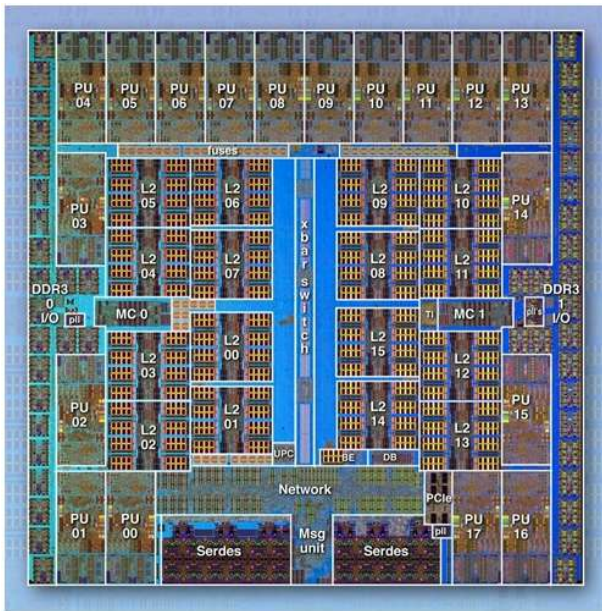
Source: Konrad Lai (Intel) slides "Transactional Memories"

■ Intel Haswell+

▶ Transactional Synchronization Extensions (TSX)

- **Hardware Lock Elision (HLE)** permite uma conversão fácil de programas baseados em locks para programas transacionais de uma maneira que é retrocompatível
- **Restricted Transactional Memory (RTM)** uma implementação mais completa do mecanismo de memória transacional.

- Utilizados no computador Sequoia no LLNL
- Um dos primeiros a ter suporte à HTM
- Processador multicore PowerPC
- 4-way SMT
- TM funciona na L2 que tem 32MB
- 18 cores
 - ▶ 16 para uso geral
 - ▶ 1 para o sistema operacional
 - ▶ 1 de "reserva"



STM:

- é mais flexível que HTM e permite a implementação de uma variedade bem maior de algoritmos
- é mais fácil de modificar e evoluir
- pode integrar mais facilmente com sistemas pré-existentes e com *features* de linguagens de alto nível tais como *garbage collection*
- tem menos limitações intrínsecas que são impostas por limites de hardware como, por exemplo, tamanho das caches

HTM:

- tem um *overhead* tipicamente menor do que STM
- depende menos de otimizações dos compiladores
- podem consumir consideravelmente menos energia
- tratam todos os acessos à memória dentro de uma transação como transacionais
- provê isolamento garantido entre transações, ainda que aplicações sejam mal comportadas
- é mais apropriado para linguagens como C/C++ que executam sem compilação dinâmica, *garbage collection*, etc

Conclusões

- TM surgiu como uma solução bem contada para simplificar a programação paralela.
- Existem diversas implementações não compatíveis entre si.
 - ▶ Melhorou um pouco com a padronização feita pelo GCC
- TM, apesar de ainda ser um terreno muito fértil para pesquisa tem, nos últimos anos, perdido um pouco da atratividade. O seu futuro parece incerto.

Exemplo de código GCC

- Disponível desde o GCC 4.7
- Para compilar é preciso utilizar a flag: `-fgnu-tm`
- Também é possível *plugar* diferentes implementações de TM

```
1  int a = 0, b = 0, c = 0;
2  int main(int argc, char *argv[]) {
3      pthread_t thr[THR_NUM];
4
5      for (int i = 0; i < THR_NUM; i++)
6          pthread_create(&thr[i], NULL, foo, NULL);
7      for (int i = 0; i < THR_NUM; i++)
8          pthread_join(thr[i], NULL);
9
10     printf("a = %d b = %d c = %d\n", a, b, c);
11     return 0;
12 }
```

```
1 void *foo(void* ignore) {
2     for (int i = 0; i < ITER_NUM; ++i) {
3         ++a;
4         b += 2;
5         c = a + b;
6     }
7     return NULL;
8 }
```

```
$ gcc tm_ex1.c -o tm_ex1 -lpthread
$ ./tm_ex1
a = 1883161 b = 4129820 c = 6012981
$ ./tm_ex1
a = 1711665 b = 3755986 c = 5467651
$
```

```
1 void *foo(void* ignore) {
2     for (int i = 0; i < ITER_NUM; ++i) {
3         __transaction_atomic {
4             ++a;
5             b += 2;
6             c = a + b;
7         }
8     }
9     return NULL;
10 }
```

```
$ gcc -fgnu-tm tm_ex1_tm.c -o tm_ex1_tm -lpthread
$ ./tm_ex1_tm
a = 40000000 b = 80000000 c = 120000000
$ ./tm_ex1_tm
a = 40000000 b = 80000000 c = 120000000
$ ./tm_ex1_tm
a = 40000000 b = 80000000 c = 120000000
$
```

- The GNU Transactional Memory Library - <https://gcc.gnu.org/onlinedocs/gcc-5.5.0/libitm.pdf>
- Technical Specification for C++ Extensions for Transactional Memory - <http://www.open-std.org/Jtc1/sc22/wg21/docs/papers/2015/n4514.pdf>