CSCI-GA.3033-012
Multicore Processors: Architecture & Programming

Lecture 11: Transactional Memories

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

What Are We Talking About?

- Incorporating transactions in parallel programming → computations wrapped in transactions
- A alternative way to coordinate concurrent threads
- Characteristics of a transaction: ACI
  - Atomicity
  - Consistency
  - Isolation

Databases!!

- Database systems have successfully exploited parallel hardware for decades.
- Databases achieve good performance by executing many queries simultaneously and by running queries on multiple processors when possible.
- The author of an individual query need not worry about this parallelism!

Databases!!

- DB programming model → transactions
- Computation executes as if it was the only computation accessing the DB.
- results indistinguishable from the situation in which the transactions run one after the other → serializability
- Transactions allow concurrent operations to access a common DB and still produce predictable, reproducible results.

Why Don't we Learn from DB?

Transactional Memory

In multicore, the main data storage during execution is typically the memory.

A transaction is a sequence of actions that appears indivisible and instantaneous to an outside observer.

Failure Atomicity

All constituent actions in a transaction complete successfully, or that none of these actions appear to start executing.

Consistency

Application dependent

Isolation

transactions do not interfere with each other while they are running, regardless of whether or not they are executing in parallel

Durability

once a transaction commits, its result is permanent

ACID Properties

Example

Thread 1

```
begin_xaction
A = A – 20
B = B + 20
A = A – B
C = C + 20
end_xaction
```

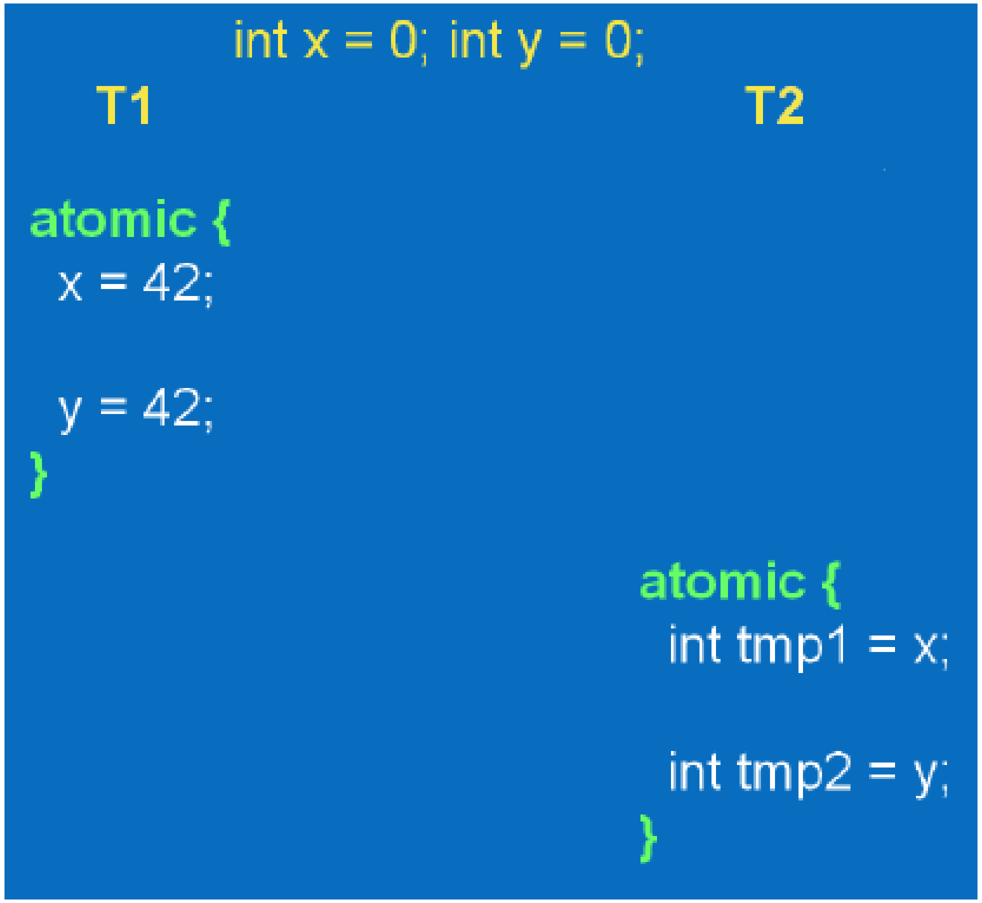
THREAD 1'S ACCESSES AND UPDATES TO A, B, C ARE ATOMIC

Thread 2

```
begin_xaction
C = C - 30
A = A + 30
end_xaction
```

THREAD 2 SEES EITHER "ALL" OR "NONE" OF THREAD 1'S UPDATES

Another Example



```
           int x = 0; int y = 0;
   T1                                 T2

atomic {
  x = 42;

  y = 42;
}

                                  atomic {
                                    int tmp1 = x;

                                    int tmp2 = y;
                                  }
```

Which execution is correct?

Who Uses TM?

- Programmer
- Compiler designer to implement some high-level language features
- Important: Beside using TM for parallel programming, it can also be used in:
  - error recovery
  - real-time programming
  - multitasking

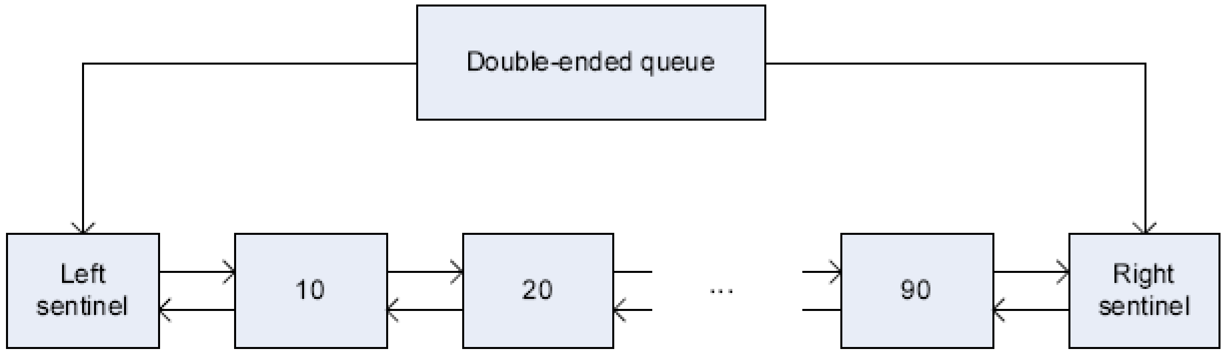
Basic TM

```
// Transaction management
void StartTx();
bool CommitTx();
void AbortTx();
```

```
// Data access
T ReadTx(T *addr);
void WriteTx(T *addr, T v);
```

Important: Different implementations of TM may have different names for functions.

Simple Example



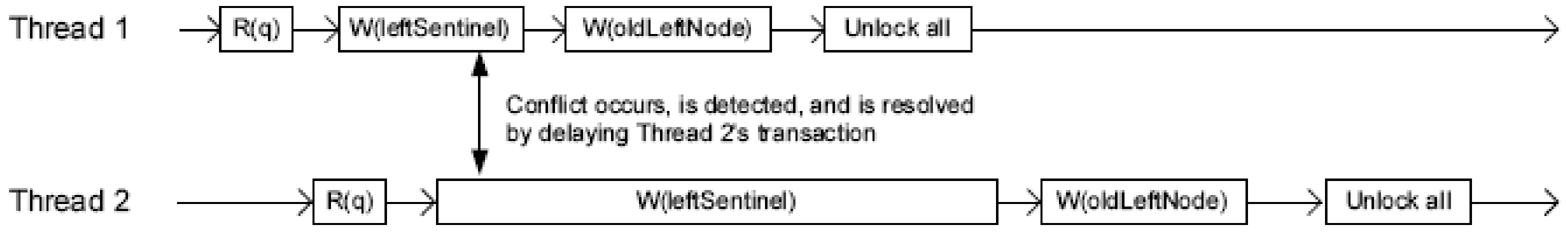
```
void PushLeft(DQueue *q, int val) {
  QNode *qn = malloc(sizeof(QNode));
  qn->val = val;
  QNode *leftSentinel = q->left;
  QNode *oldLeftNode = leftSentinel->right;
  qn->left = leftSentinel;
  qn->right = oldLeftNode;
  leftSentinel->right = qn;
  oldLeftNode->left = qn;
}
```

```
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());
}
```

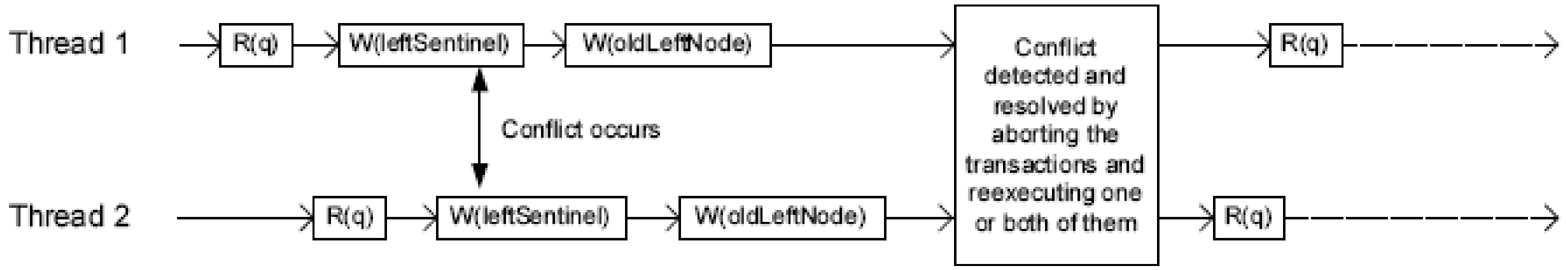
Concurrency Control: Conflict-Detection-Resolution

- A conflict occurs when two transactions perform conflicting operations on the same piece of (2 writes, or a read and a write)
- The conflict is detected when the underlying TM system determines that the conflict has occurred.
- The conflict is resolved when the underlying system or code in a transaction takes some action to avoid the conflict—e.g., by delaying or aborting one of the conflicting transactions.

Concurrency Control: Conflict-Detection-Resolution

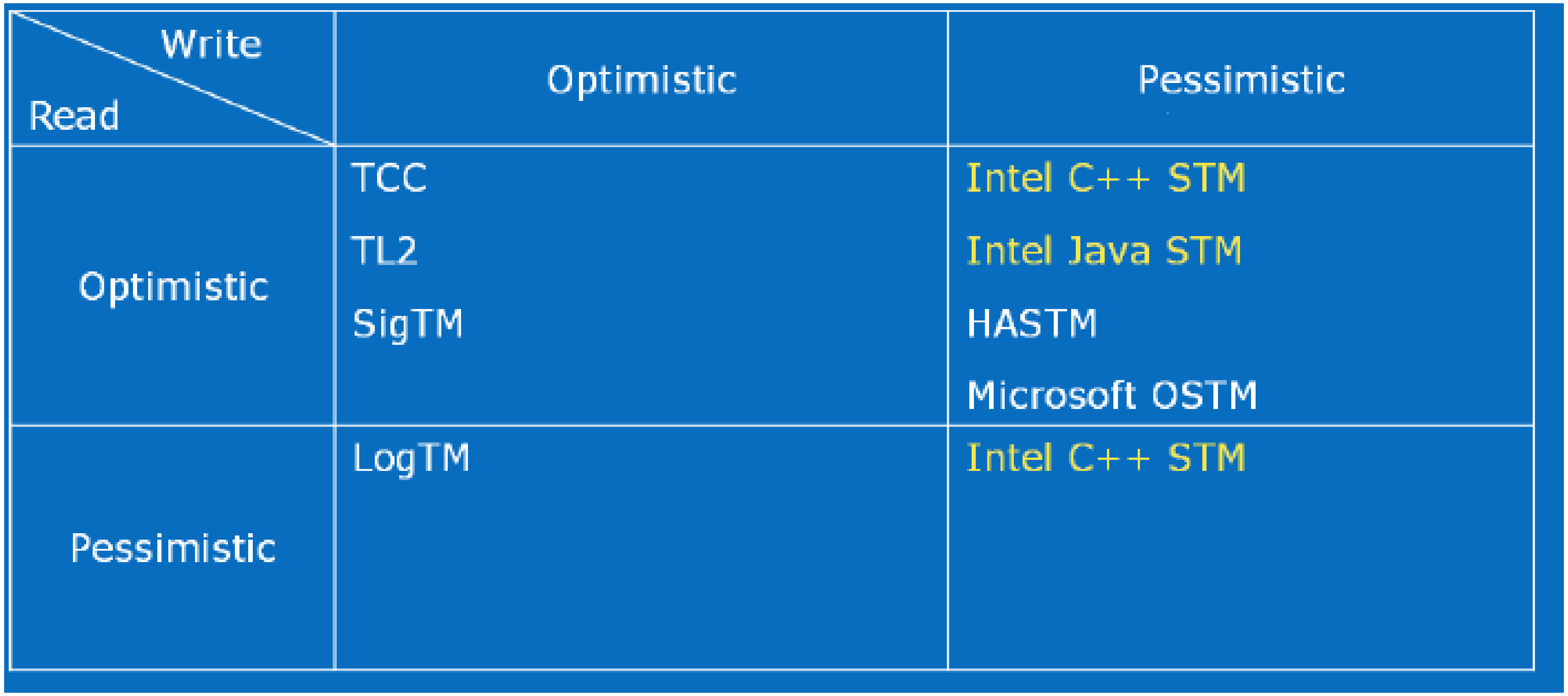


(a) Pessimistic concurrency control.



(b) Optimistic concurrency control.

Some TM implementations use Pessimistic control, others use Optimistic control.



Write / Read	Optimistic	Pessimistic
Optimistic	TCC TL2 SigTM	Intel C++ STM Intel Java STM HASTM Microsoft OSTM
Pessimistic	LogTM	Intel C++ STM

Example of Available TMs

Version Management

- What to do about writes before a transaction commits?
- Eager version management
  - The transaction directly modifies the data in memory
  - Keeps an undo-log holding overwritten data
  - Requires pessimistic concurrency control
- Lazy version management
  - updates are delayed until a transaction commits
  - transaction maintains its tentative writes in a transaction-private redo-log

Conflict Detection

- With pessimistic approach it is easy → locks!
- With optimistic approach, there are several issues:
  - Granularity of conflict (cache line, objects, …)
  - The time at which detection occurs:
    - When transaction declares its intend to access the data → eager conflict detection
    - On validation: can occur several times during transaction lifetime
    - On commit → lazy conflict detection
  - Which kind of access is treated as conflicts?
    - Among concurrent transactions
    - Between active and committed transactions

What Can Go Wrong Here?

```
// Thread 1
do {
  StartTx();
  WriteTx(&x, 1);
} while (!CommitTx());
```

```
// Thread 2
do {
  StartTx();
  int tmp_1 = ReadTx(&x);
  while (tmp_1 == 0) { }
} while (!CommitTx());
```

Can We Make Things Simpler?

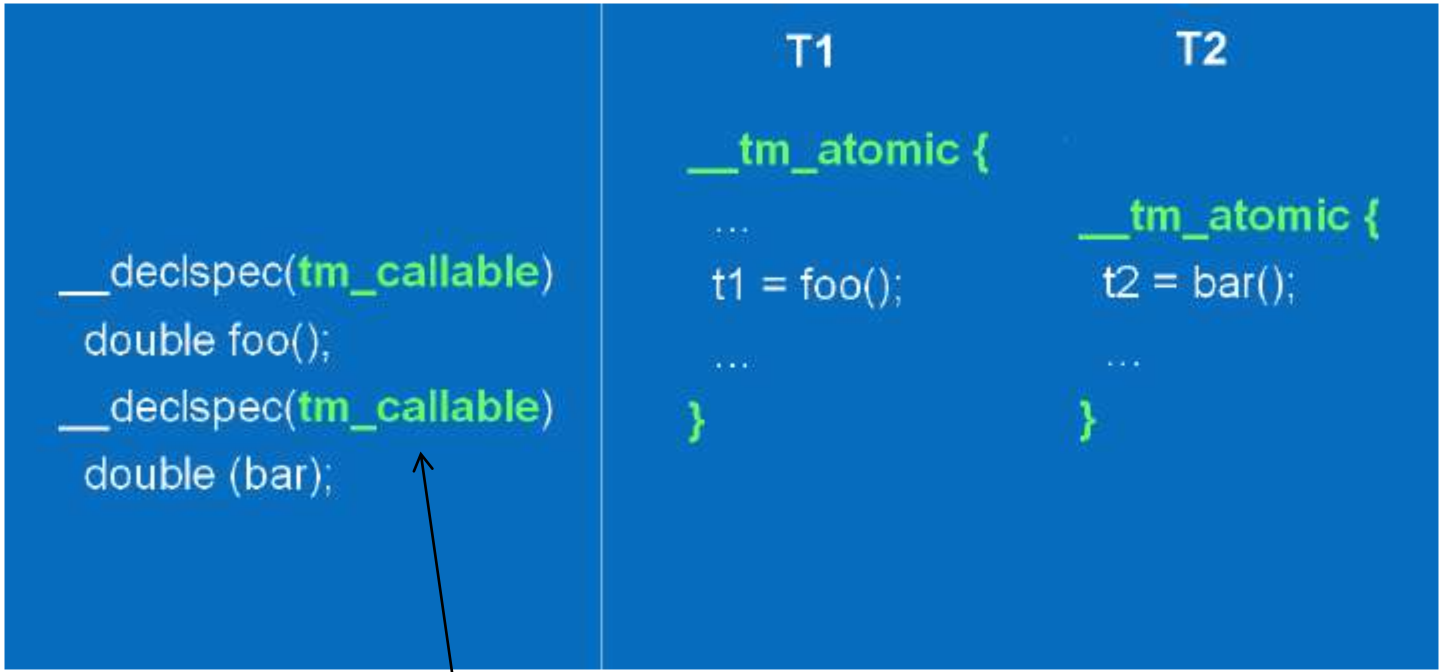
- Things look very verbose hence error prone
- Instead of WriteTx and ReadTx can we have something simpler?
- Atomic block of statements
  - getting rid of WriteTx and ReadTx
  - Implemented for many languages

Atomic Blocks

```
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 key advantage of atomic blocks over lock-based critical sections is that the atomic block does not need to name the shared resources that it intends to access or synchronize with.

Example of TM with C/C++



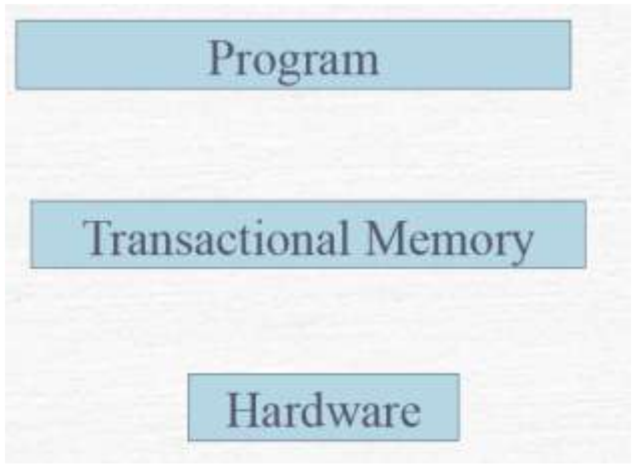
```
                                  T1                    T2

                              __tm_atomic {
                                                    __tm_atomic {
                                ...
__declspec(tm_callable)         t1 = foo();           t2 = bar();
  double foo();
                                ...                   ...
__declspec(tm_callable)       }                     }
  double (bar);
```

- Can be called transactionally
- The above is Windows version.
- The Linux version: __attribute__(tm_callable) double foo();

How to Provide the *Illusion* of Transactions?

Software Transactional Memory (STM)

Hardware Transactional Memory (HTM)



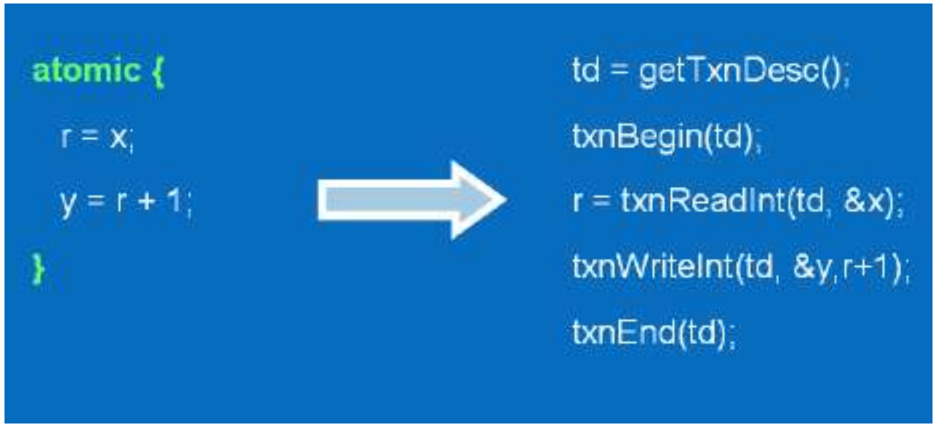
Software Transactional Memory (STM)

Components:

- transaction descriptor: is the per-transaction data structure that keeps track of the state of the transaction
- Undo-log or Redo-log
- read-set or write-set: tracks the memory locations that the transaction has read from or written to

STM

- Compiler instruments code with transaction prolog, epilog, and read/write function.
- Runtime tracks memory accesses, detects conflicts, and commits/aborts execution.

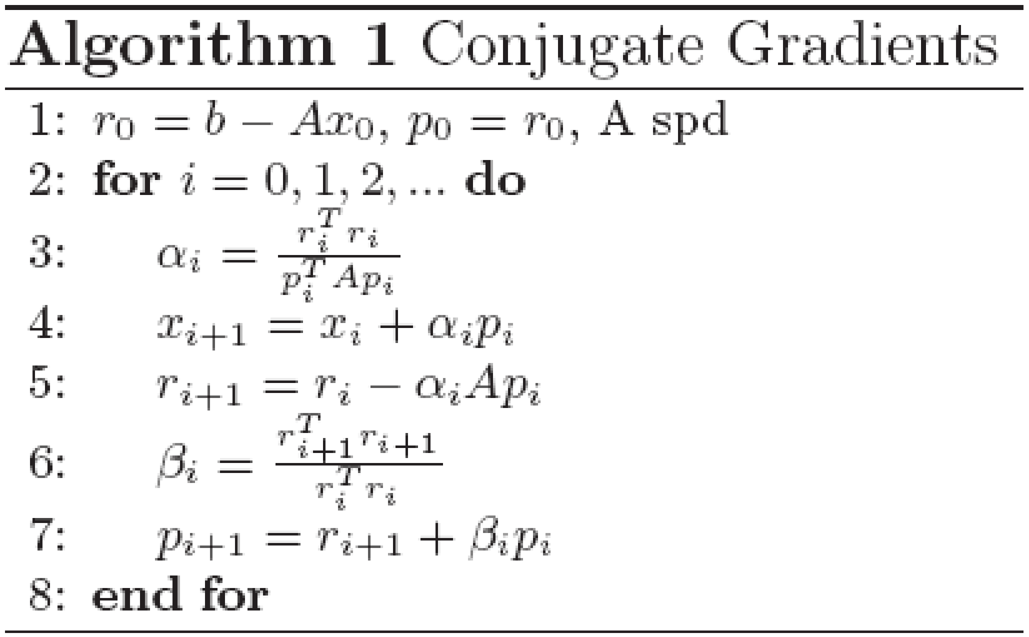


```
atomic {
  r = x;
  y = r + 1;
}
```

```
td = getTxnDesc();
txnBegin(td);
r = txnReadInt(td, &x);
txnWriteInt(td, &y, r+1);
txnEnd(td);
```

STM vs OpenMP vs Pthreads

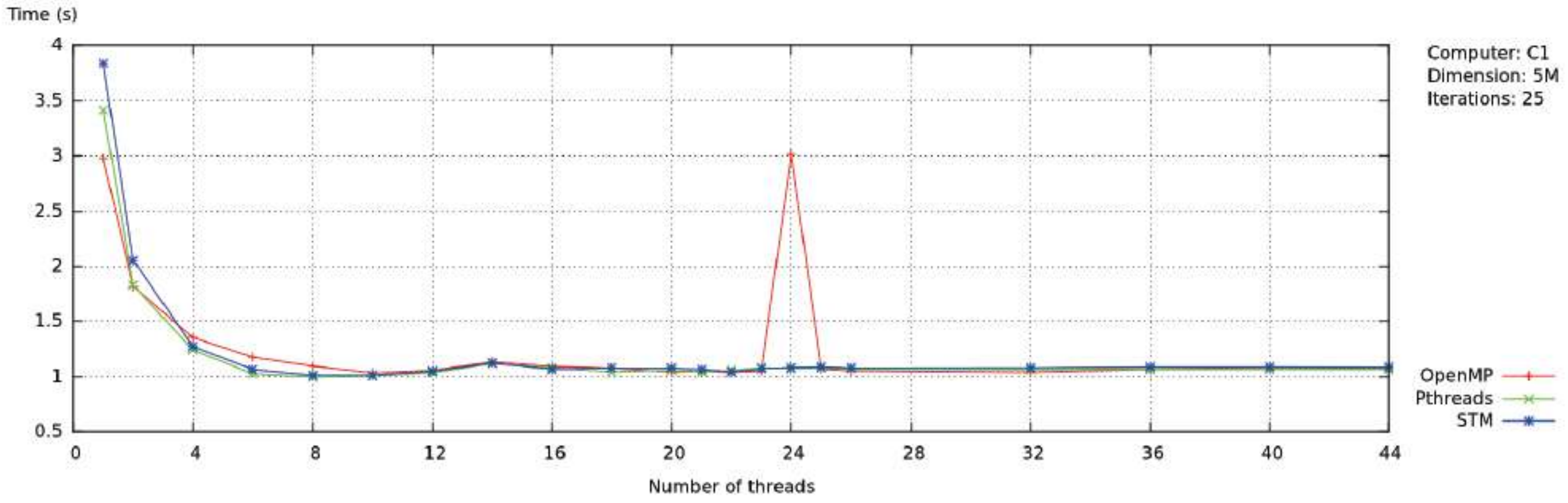
Problem to Parallelize:



Algorithm 1 Conjugate Gradients

```
1: $r_0 = b - Ax_0$, $p_0 = r_0$, A spd
2: **for** $i = 0, 1, 2, \ldots$ **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**
```

STM vs OpenMP vs Pthreads



Hardware Transactional Memory (HTM)

- Three flavors
  - Full implementation of TM in hardware
  - Allowing hardware transactions to coexist with software transactions
  - hardware extension to provide speed-up to parts of software TM

HTM

- HTM must perform the following functions
  - identify memory locations for transactional accesses
  - manage the read-sets and write-sets of the transactions
  - detect and resolve data conflicts,
  - manage architectural register state
  - commit or abort transactions

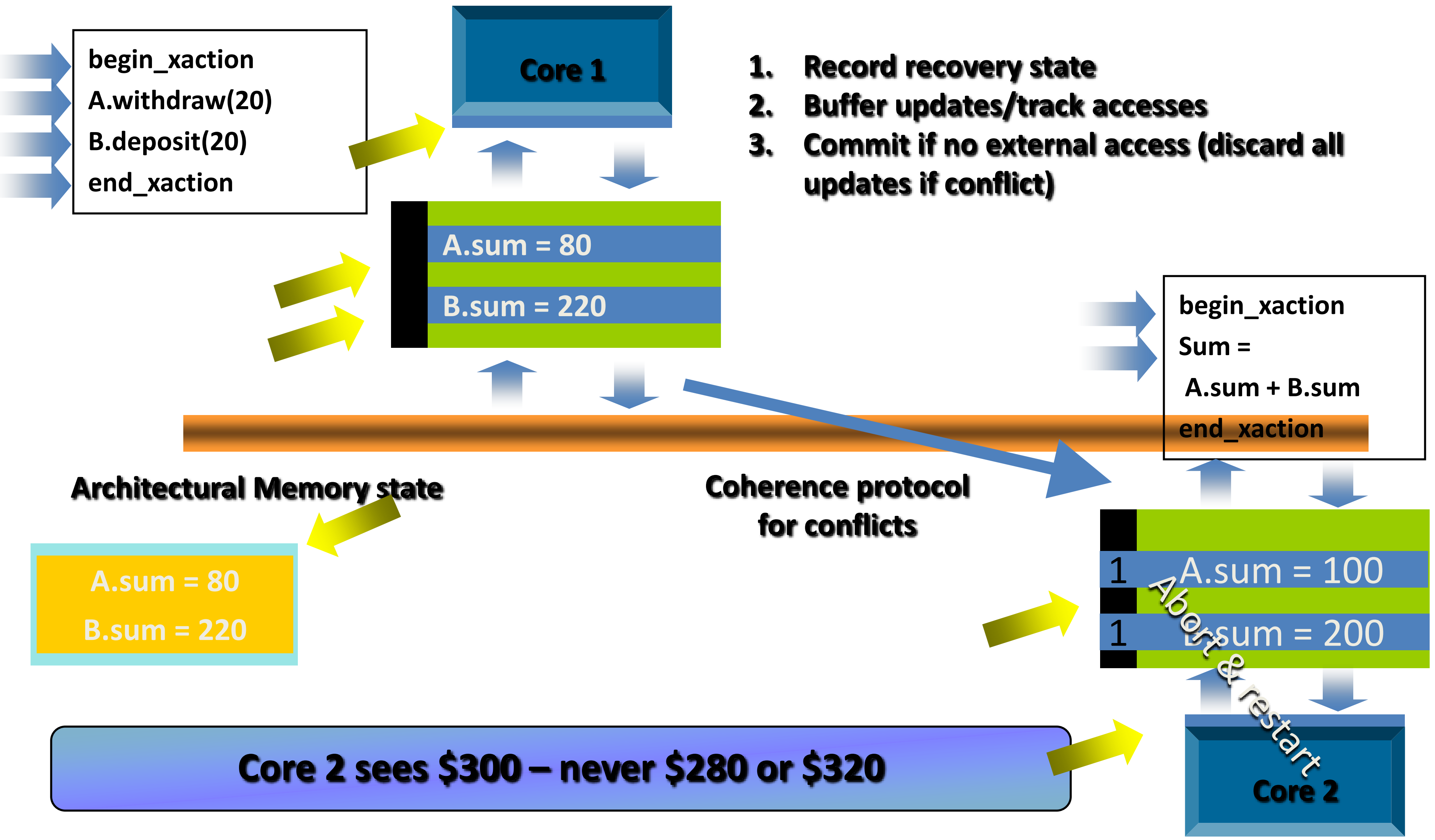
Requirements for Supporting Transactions

Buffering	Transactional cache
Conflict detection	Cache coherence protocol
Abort/Recovery	Invalidate transactional cache line
Commit	Validate transactional cache line

HTM

- Extensions to the instruction set
- Tracking read-sets and buffering write-sets is done using caches and buffers
- Coherence messages trigger conflict detection
- Nearly all conventional HTM proposals perform eager conflict detection

Hardware Support for Performance



1. Record recovery state
2. Buffer updates/track accesses
3. Commit if no external access (discard all updates if conflict)

Core 2 sees $300 – never $280 or $320

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

Transactional memory going mainstream with Intel Haswell

Intel has announced that its **Haswell architecture**, due to ship some time in 2013, will include hardware support for transactional memory.

Transactional memory going mainstream with Intel Haswell

Haswell's transactional support, which Intel is calling **Transactional Synchronization Extensions** (TSX), come in two parts:

• **Hardware Lock Elision** (HLE) allows easy conversion of lock-based programs into transactional programs in a way that's backwards compatible with current processors.

• **Restricted Transactional Memory** (RTM) is a more complete transactional memory implementation.

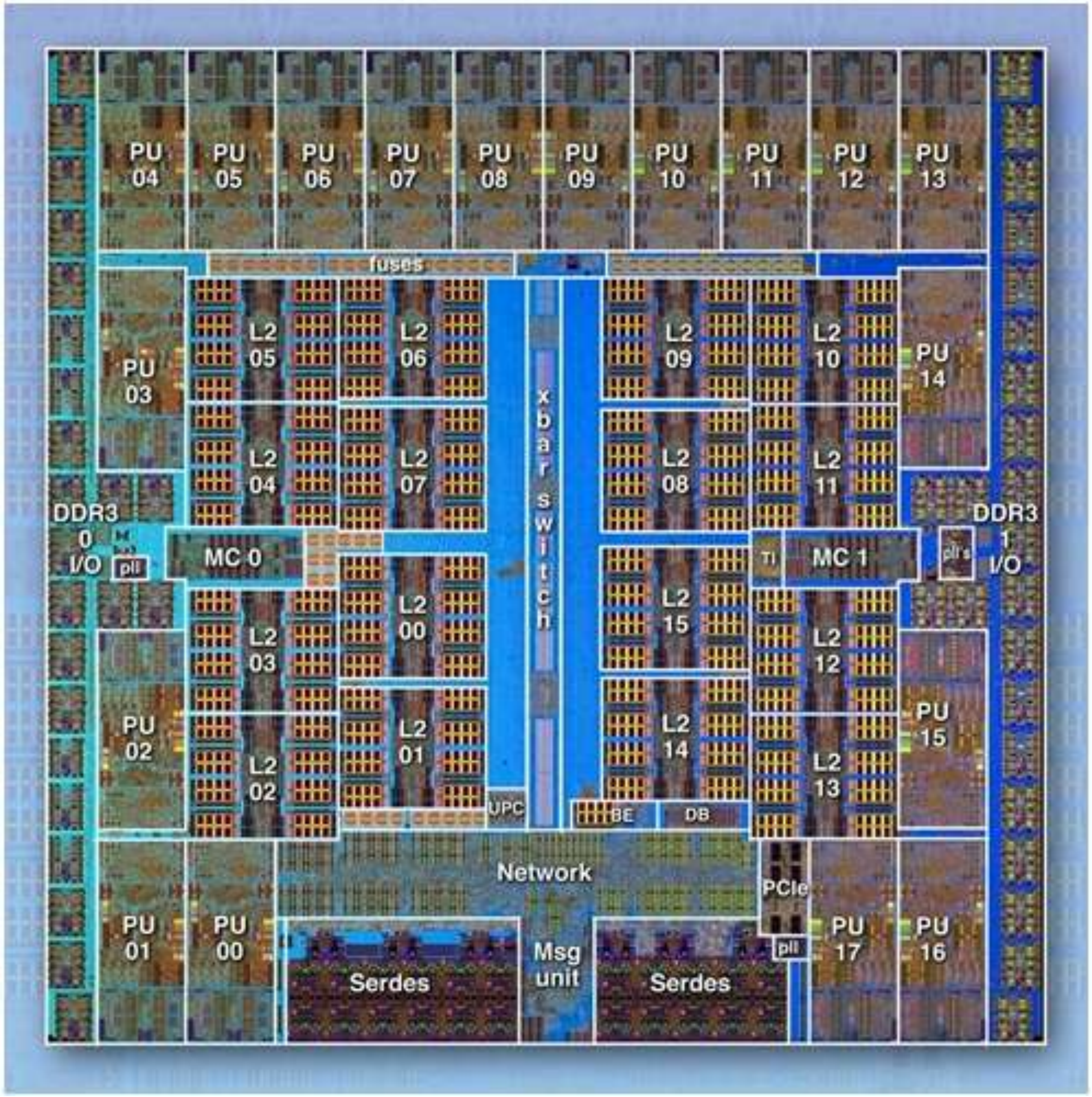
IBM Blue Gene/Q

The BlueGene/Q processors that will power the 20 petaflops Sequoia supercomputer being built by IBM for Lawrence Livermore National Labs will be the first commercial processors to include hardware support for transactional memory.

IBM Blue Gene/Q

- multicore 64-bit PowerPC-based system-on-chip
- based on IBM‘s 4-way multithreaded PowerPC A2 design
- 1.47 billion transistor chip
- TM will appear in 32MB level 2 cache
- 18 cores
    - 16 for running actual computations
    - 1 for the operating system
    - 1 to improve chip reliability

IBM Blue Gene/Q



STM vs HTM

- Software is more flexible than hardware and permits the implementation of a wider variety of more sophisticated algorithms.
- Software is easier to modify and evolve than hardware.
- STMs can integrate more easily with existing systems and language features, such as garbage collection.
- STMs have fewer intrinsic limitations imposed by fixed-size hardware structures, such as caches.

HTM vs STM

- HTM systems can typically execute applications with lower overheads than STM systems.
- Less reliant than STM systems on compiler optimizations to achieve performance.
- HTM systems can have better power and energy profiles.
- Treat *all memory accesses within a transaction as implicitly transactional, accommodating*
- HTM systems can provide strong isolation without requiring changes to non-transactional memory accesses.
- HTM systems are well suited for systems languages such as C/C++ that operate without dynamic compilation, garbage collection, and so on.

Conclusions

- TM appears in recent years as strong candidate for parallel programming
- There are many different implementations and this complicates portability a bit.
- TM is still work-in-progress, details vary between languages and between processors so care is needed when implementing an algorithm in a particular setting.