# HyflowCPP: A Distributed Transactional Memory Framework for C++

**Sudhanshu Mishra, Alexandru Turcu, Roberto Palmieri, Binoy Ravindran**

Virginia Tech

USA

*The 12th IEEE International Symposium on Network Computing and Applications (IEEE NCA13)*

# Lock-based concurrency control has serious drawbacks

- Coarse grained locking
  - Simple
  - But no concurrency

```
public boolean add(int item) {
  Node pred, curr;
  lock.lock();
  try {
    pred = head;
    curr = pred.next;
    while (curr.val < item) {
      pred = curr;
      curr = curr.next;
    }
    if (item == curr.val) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = curr;
      pred.next = node;
      return true;
    }
  } finally {
    lock.unlock();
  }
}
```

# Fine-grained locking is better, but…

- Excellent performance
- Poor programmability
- Lock problems don’t go away!
  - Deadlocks, livelocks, lock-convoying, priority inversion,….
- Most significant difficulty – composition

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

# Lock-free synchronization overcomes some of these difficulties, but…

“lock-free retry loop”

```
public boolean add(int item) {
 while (true) {
   Node pred = null, curr = null, succ = null;
   boolean[] marked = {false};  boolean snip;
   retry: while (true) {
     pred = head; curr = pred.next.getReference();
     while (true) {
       succ = curr.next.get(marked);
       while (marked[0]) {
         snip = pred.next.compareAndSet(curr, succ, false, false);
         if (!snip) continue retry;
          curr = succ; succ = curr.next.get(marked);
       }
       if (curr.val < item)
            pred = curr; curr = succ;
     }
   }
   if (curr.val == item) {  return false;
   } else {
     Node node = new Node(item);
     node.next = new AtomicMarkableReference(curr, false);
     if (pred.next.compareAndSet(curr, node, false, false)) {return true;}
   }
 }
}
```

# Transactional memory

- Like database transactions
- ACI properties (no D)
- Easier to program
- Composable

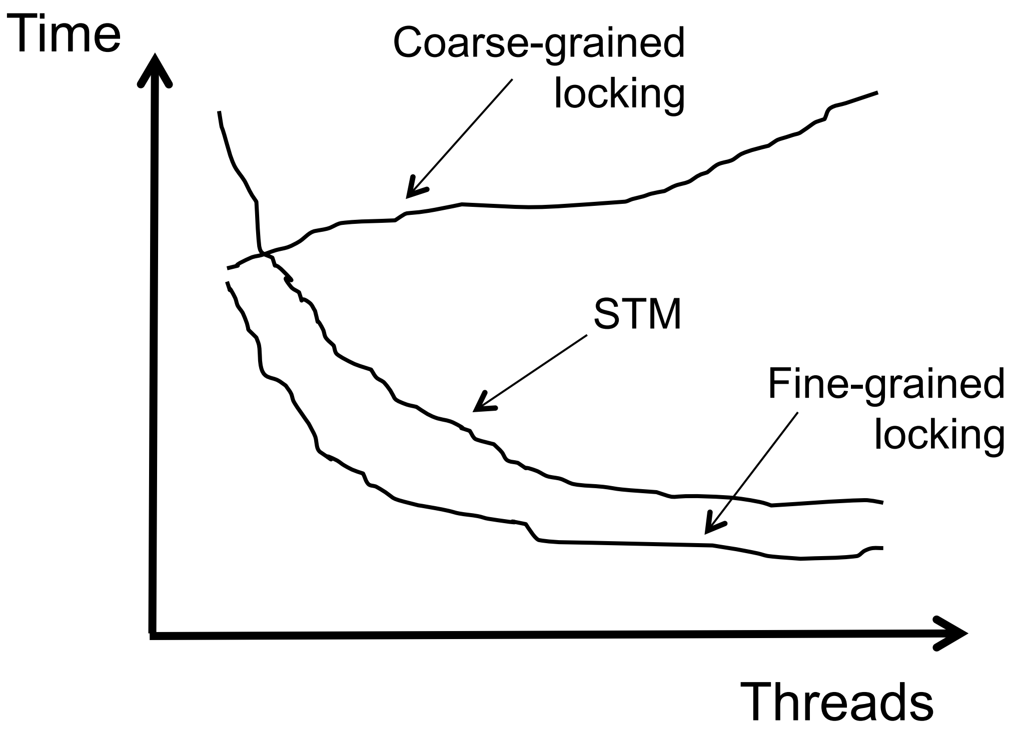
- First HTM, then STM, later HyTM

```
public boolean add(int item) {
  Node pred, curr;
  atomic {
   pred = head;
   curr = pred.next;
   while (curr.val < item) {
     pred = curr;
     curr = curr.next;
   }
   if (item == curr.val) {
     return false;
   } else {
     Node node = new Node(item);
     node.next = curr;
     pred.next = node;
     return true;
   }
  }
}
```

M. Herlihy and J. B. Moss (1993). Transactional memory: Architectural support for lock-free data structures. *ISCA*. pp. 289–300.

N. Shavit and D. Touitou (1995). Software Transactional Memory. *PODC*. pp. 204—213.

# Optimistic execution yields performance gains at the simplicity of coarse-grain, but no silver bullet




- High data dependencies
- Irrevocable operations
- Interaction between transactions and non-transactions
- Conditional waiting
- ……

E.g., C/C++ Intel Run-Time System STM (B. Saha et. al. (2006). McRT-STM: A High Performance Software Transactional Memory. *ACM PPoPP*)

# Three key mechanisms needed to create atomicity illusion

## Versioning

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

Where to store new `x` until commit?

- *Eager*: store new `x` in memory; old in *undo log*
- *Lazy*: store new `x` in *write buffer*

## Conflict detection

T0

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

T1

```
atomic{
    x = x / 25;
}
```

How to detect conflicts between `T0` and `T1`?

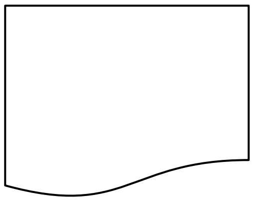
- Record memory locations read in *read set*
- Record memory locations wrote in *write set*
- Conflict if one's read or write set intersects the other's write set

# Distributed TM (or DTM)

- Extends TM to distributed systems
  - Nodes interconnected using message passing links
- Execution and network models
  - Execution models
    - Data flow DTM (DISC 05)
      - Transactions are immobile
      - Objects migrate to invoking transactions
    - Control flow DTM (USENIX 12)
      - Objects are immobile
      - Transactions move from node to node
  - Herlihy's metric-space network model (DISC 05)
    - Communication delay between every pair of nodes
    - Delay depends upon node-to-node distance

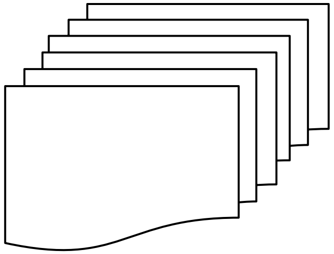
# Replication models in (dataflow) DTM

- No replication: non-fault-tolerant



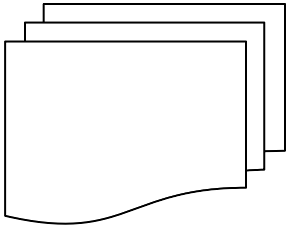
Only one copy for each object

- Full replication: fault-tolerant, but non-scalable



All objects replicated on all nodes

- Partial replication: fault-tolerant and scalable



Each object replicated only at a subset of nodes

# Paper’s motivations

- C++ preferred for high performance products
- No JVM Overhead
- Manual Memory Management
  - No automatic garbage collector
- Low-level optimization for network management
- No current complete DTM support in C++

# HyflowCPP advantages

- First ever DTM support for C++
- Pluggable support for different algorithm and policies
- Performance oriented design
- Support of various features:
  - Strong Atomicity
  - Nesting supports
    - Open Nesting
    - Close Nesting
  - Checkpointing

# Why HyflowCPP?

- First ever DTM support for C++
- Pluggable support for different algorithm and policies
- Performance oriented design
- Support of various features:
  - Strong Atomicity
  - Nesting supports
    - Open Nesting
    - Close Nesting
  - Checkpointing

Integration with transactional systems in C++ without additional layer

Ready for complex distributed concurrency controls and replication models

Support for avoiding false-conflicts in distributed transactional data structures

Partial abort mechanisms already provided by the framework and ready for programmers

# Programming Interface

- Atomic section based support using Macros

HyflowCPP atomic construct

```
HYFLOW_ATOMIC_START{
// Example of simple compare
// and swap operation
    value = Read(Address);
    if( value==myValue )
        Write(Address, myValue)
}HYFLOW_ATOMIC_END;
```

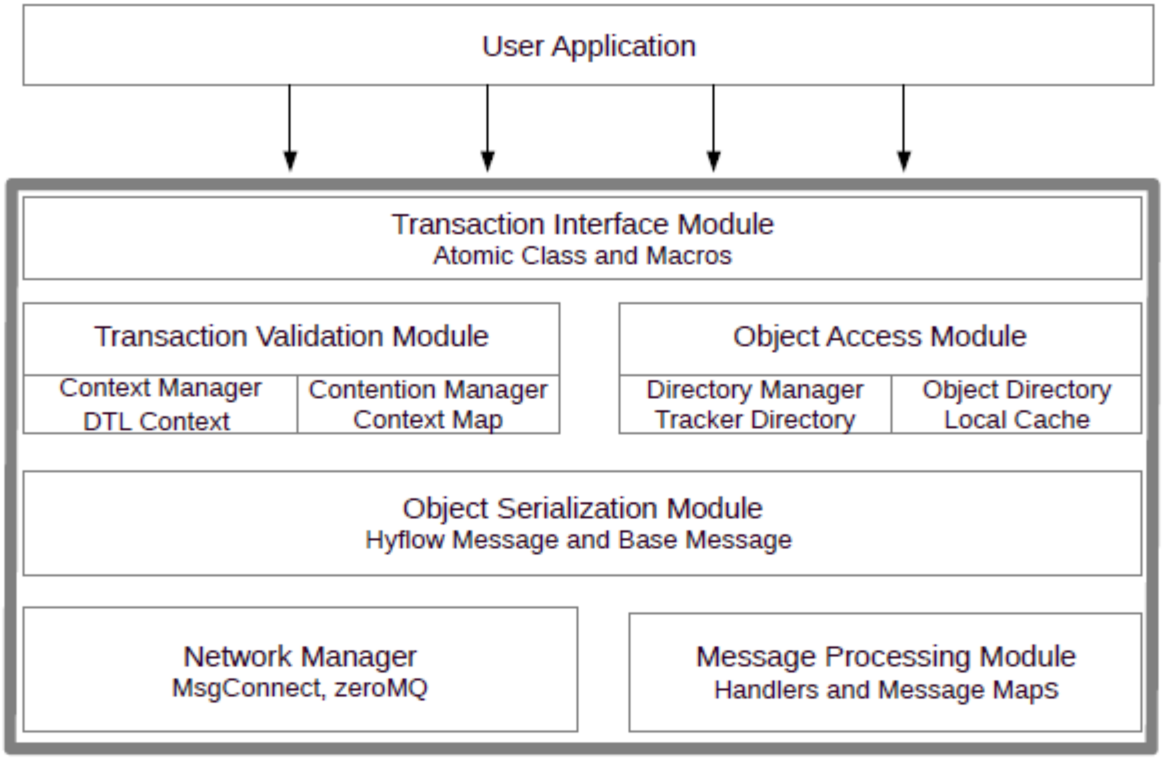
Standard STM atomic construct

```
atomic{
// Example of simple compare
// and swap operation
    value = Read(Address);
    if( value==myValue )
        Write(Address, myValue)
}
```

- HYFLOW_PUBLISH(obj),
- HYFLOW_DELETE(obj)
- HYFLOW_FETCH(objId, isRead)
- HYFLOW_CHECKPOINT_HERE( ) and more….

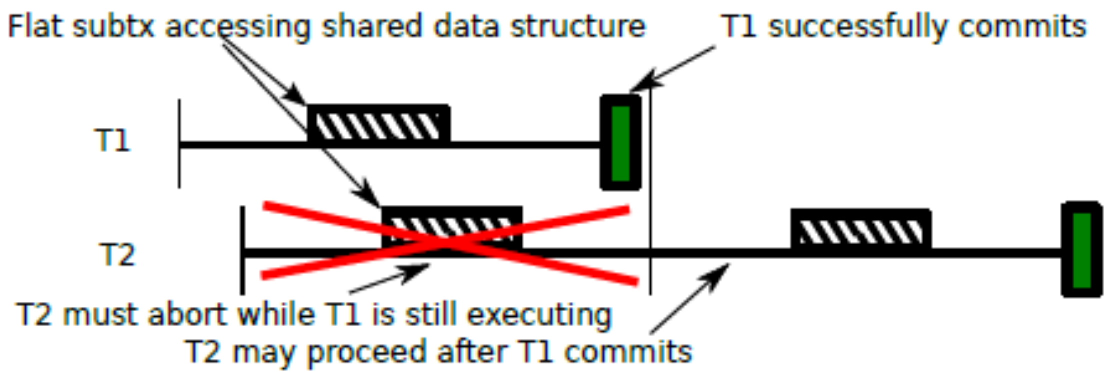
# System Architecture

- API level design
- ~12K LoC
- Modular Architecture
- Pluggable support for different component.
- Dependencies:
  - Boost Thread
  - Boost Serialization
  - ZeroMQ
  - Pthread
  - Intel TBB



# Flat Nesting

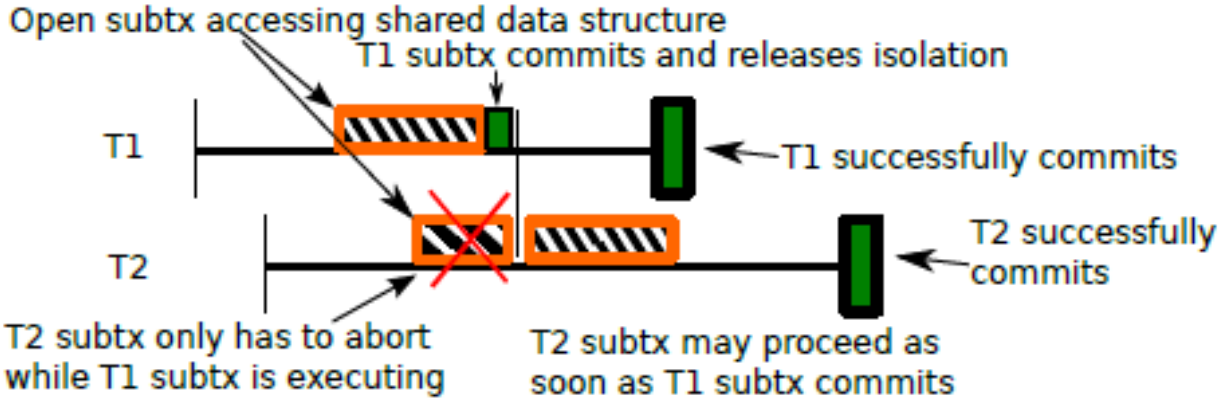
- Composable Transaction.
- No real nesting support.




- Thread Context factory to provide thread specific context.
- Merge inner transaction to parent transaction at commit time.

# Open Nesting

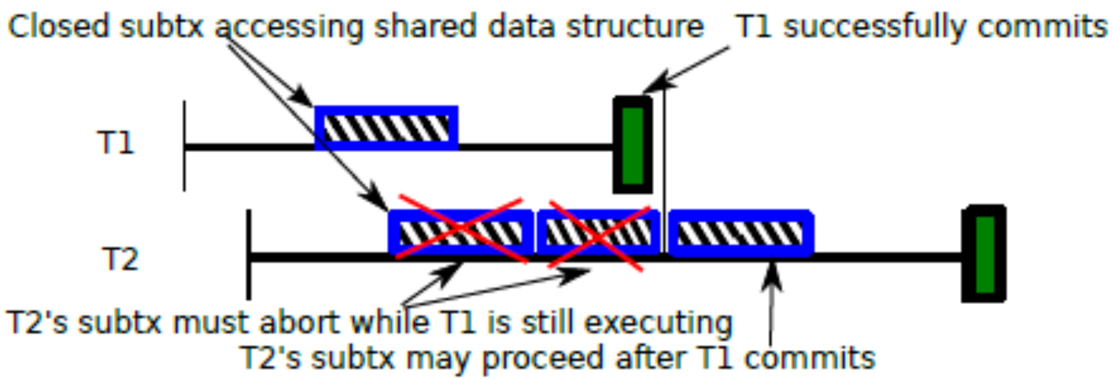
- Abstract lock overhead, lock issues like livelock.




- Acquire the abstract lock in inner transaction.
- Outer most transaction release the abstract locks.
- Performance improvement by removing false conflict.

# Closed Nesting

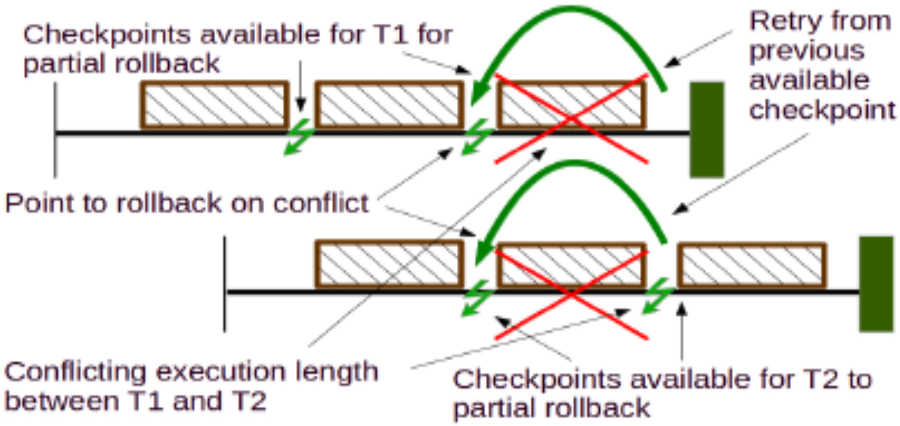
- Performance improvement by retrying inner transaction
- Partial rollback limited to current inner transaction executing




- Inner-transaction commit operation merges the inner-transaction’s read-/write-set to parent transaction
- Parent-transaction globally commits when all the inner transactions are successfully executed

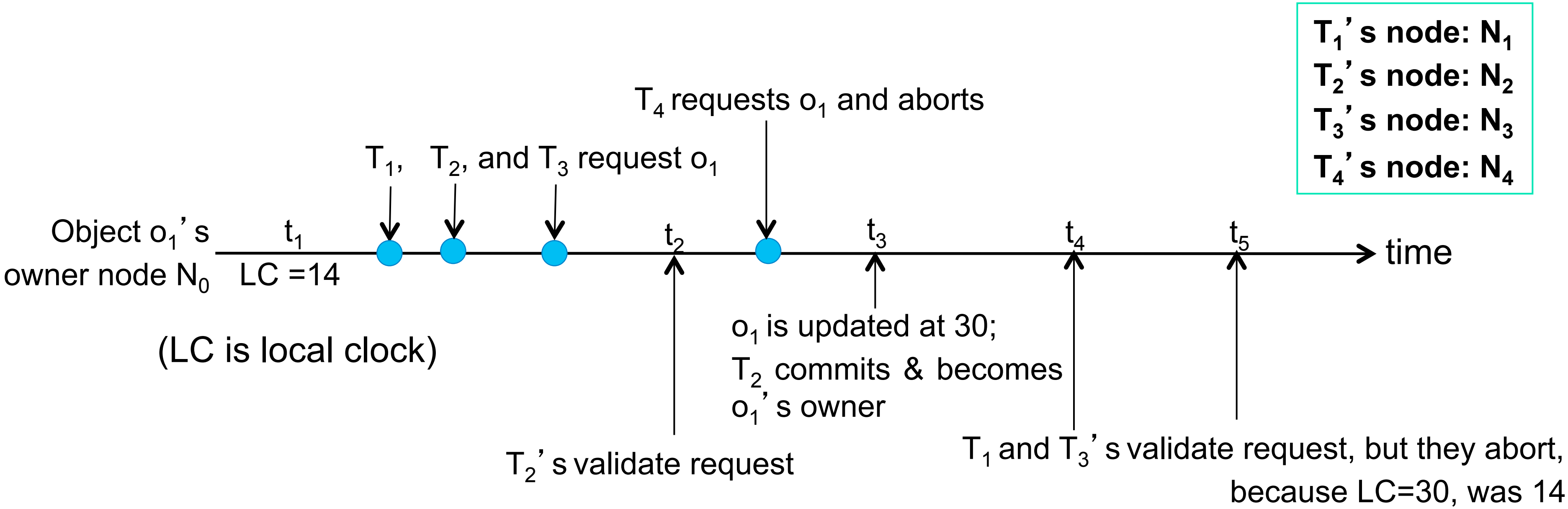
# Checkpoitning (no-nesting model)

- Performance improvement by partial rollback.
- Non negligible memory overhead.




- Transaction creates checkpoints locally
- Checkpoints saved along with the object accessed
- Conflict during execution phase, can restart from appropriate checkpoint

# Atomicity, consistency, and isolation in data-flow DTM



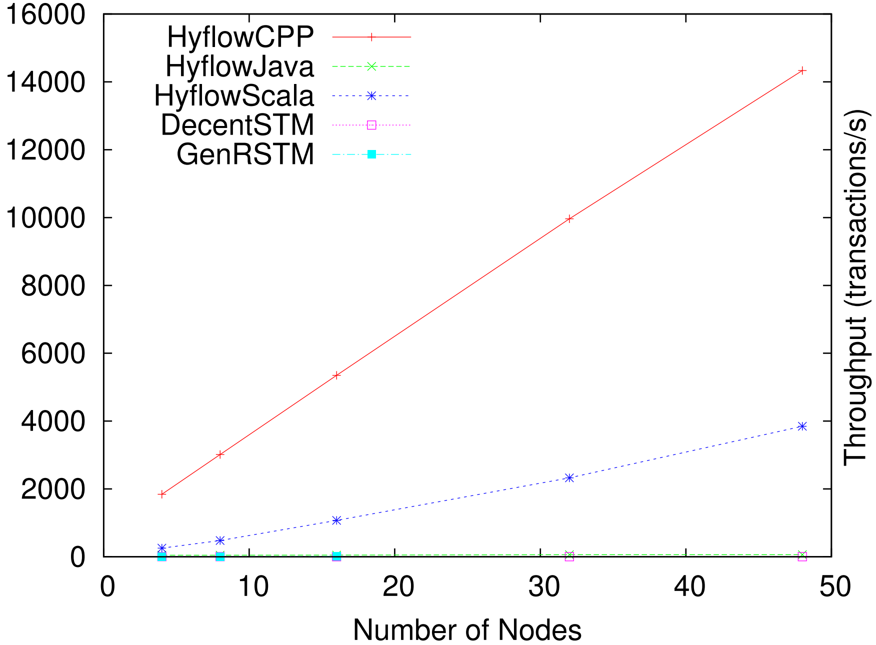

- Transactional Forwarding Algorithm (TFA)
  - Early validation of remote objects
  - Atomicity for object operations in the presence of asynchronous clocks

# Experiments

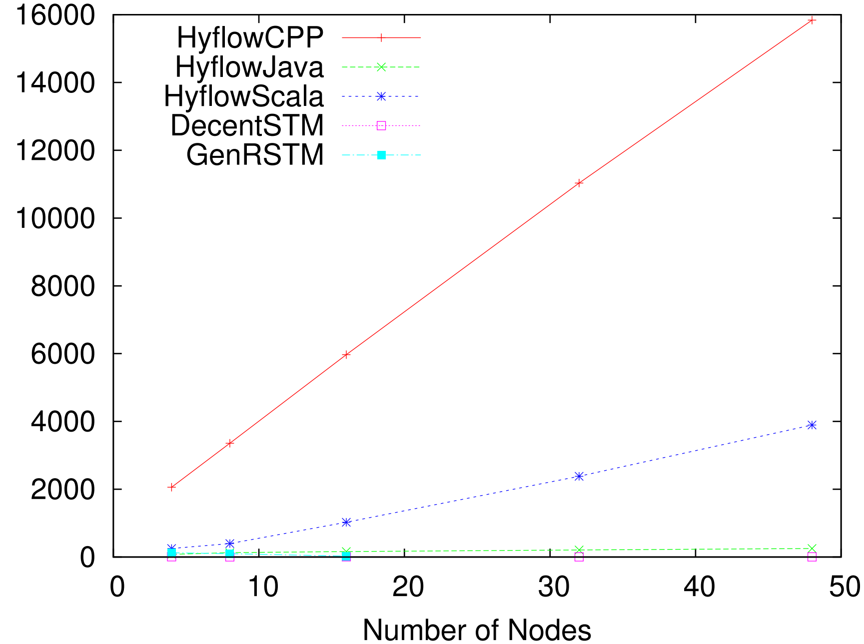
- Test-bed:
  - Cluster of 48 nodes interconnected by a Gigabit connection
  - Each node equipped with 2 application threads running
  - Ubuntu Linux 10.04 server OS
- Competitors (JVM-based DTM frameworks):
  - GenRSTM, DecentSTM, HyflowJava & HyflowScala.
- Benchmarks:
  - Micro Benchmarks:
    - Bank, Linked-List, Skip-list, Binary Search Tree, Hash-table
  - Macro Benchmarks:
    - Loan, Vacation, TPCC

# Flat Nesting

- Bank benchmark




Bank 20% read workload

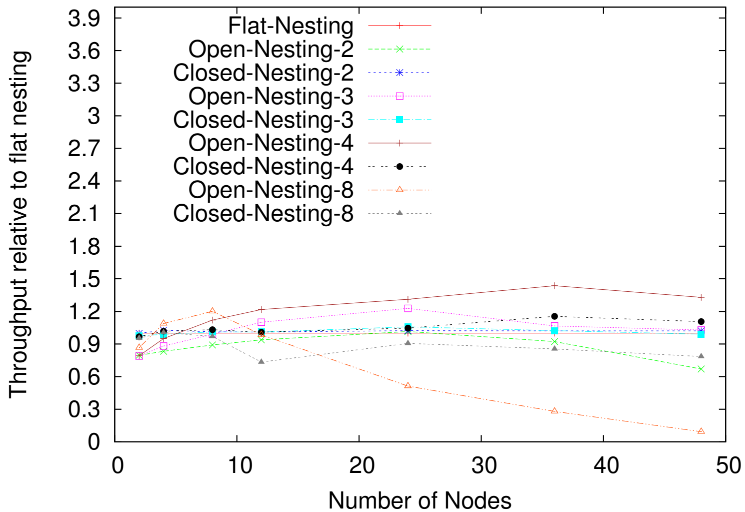



Bank 80% read workload

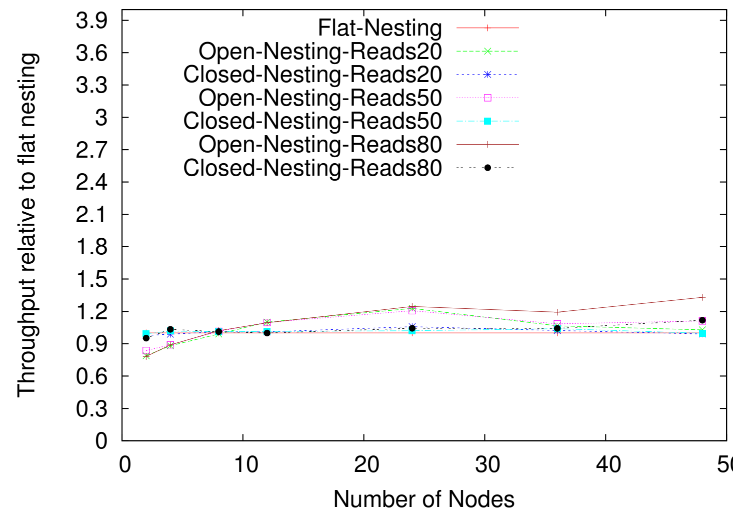
- Best competitor is HyflowScala.
- HyflowCPP speed-up is around 4x

# Open Nesting

- Hash Table benchmark
  - {2,3,4,8} inner-transactions
  - 20% read workload



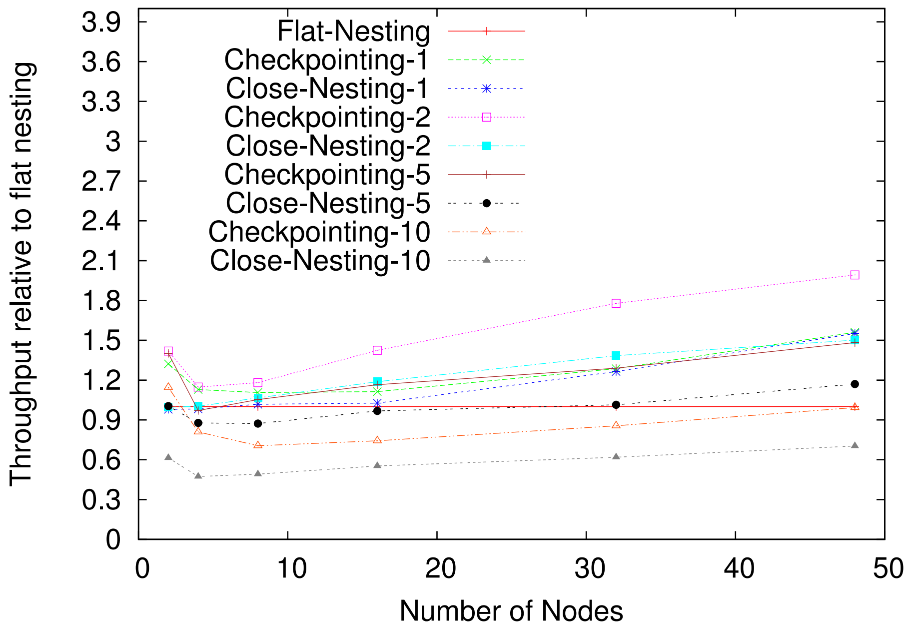

- Hash Table benchmark
  - {20,50,80}% of read workload
  - 3 inner-transactions




- Relative throughput to flat nesting
- Maximum speed-up 1.5x due to overhead of compensating actions in case of abort

# Closed Nesting and Checkpointing

- Bank benchmark
  - {1,2,5,10} granularity of Checkpointing and Closed-Nesting
  - Relative throughput to flat nesting




- HyflowCPP speed-up is around 2x
- Checkpointing is better than Closed-Nesting

# Thank you for the attention & Questions

HyflowCPP is available as open-source software at:

http://www.hyflow.org/hyflow/wiki/HyflowCPP