Scalable Software Transactional Memory for Chapel High-Productivity Language

Srinivas Sridharan and Peter Kogge, U. Notre Dame

Brad Chamberlain, Cray Inc

Jeffrey Vetter, Future Technologies Group, ORNL

Talk Overview

- Motivation:
  - Most STM designs target shared memory systems
  - Need for concurrency control on large-scale systems
    - Emerging applications do not fit the MPI model
    - Distributed memory is globally addressable (e.g. PGAS model)
- GTM: Global Transactional Memory
  - Targets large-scale distributed memory systems
    - *STM metadata overhead << Network Latency*
  - Asynchronous STM abstractions: Parallelism inside TXs
  - Multi-core & Multi-node Environment
  - On-going work on Chapel Language STM exploration

Chapel Language

- Chapel is a parallel language developed by Cray Inc, part of the DARPA's HPCS program
  - Primary goal is to enhance programmer productivity
    - Improve programmability, without sacrificing performance
- TM concepts satisfy Chapel's productivity goals
  - **`Atomic`** keyword included in language specification
    - Identify transactional code segments
  - Semantics distinct from implementation mechanism
    - Based on target platform: HTM, STM, or HyTM
  - Chapel's Multiresolution language philosophy
    - High-level counterpart to low-level **`Sync`** variables

`Atomic` keyword in Chapel

- Number of open questions under investigation:
  - Strong vs Weak Isolation ?
  - Memory Consistency Model ?
  - I/O in atomic blocks ?
  - **`Sync`** variables in atomic blocks ?
  - Support STM semantics across multiple Locales ?
    - *Locale* is an architectural unit of locality
    - Threads within a locale have uniform access to local memory
    - Memory within other locales accessible at a price
    - E.g.: A multicore processor node in a cluster system

Distributed STM: Rationale

- Need for concurrency control across nodes
- STM can provide productivity benefits
  - Programmability advantages over locks
  - Lock-based approaches don't scale (serialization issues)
    - No global hardware cache coherence
  - *STM metadata overhead << Network Latency*
    - In multicores: locks preferred over STM for performance reasons
    - Comparable performance between locks and STM if communication requirements are the same
- Key: Tolerate remote communication latency

Example: Bank Transaction

Chapel Source Code

```
var balance: [1..num] int;
atomic {
  balance[i] -= amount;
  balance[j] += amount;
}
```

Compiler

Multicore STM Library

```
TX_BEGIN;
t1 = TX_LOAD(&balance[i]);
t1 = t1 - amount;
TX_STORE(&balance[i], t1);
t2 = TX_LOAD(&balance[j]);
t2 = t2 + amount;
TX_STORE(&balance[j], t2);
TX_COMMIT;
```

`Atomic` statement block is mapped to a sequence of STM library calls.

TX_BEGIN:	Start Transaction
TX_LOAD:	Transactional Read
TX_STORE:	Transactional Write
TX_COMMIT:	Commit Transaction

balance[1..num]
Shared Memory

Example: Bank Transaction

Chapel Source Code

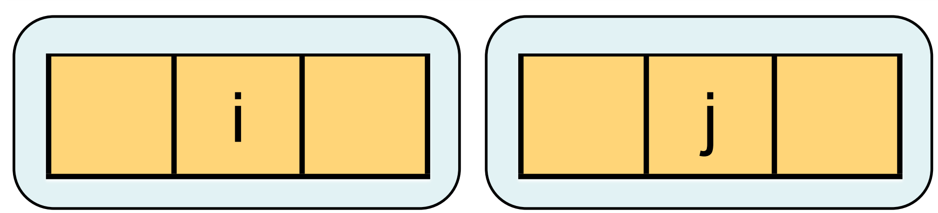
```
var balance: [1..num] int;
atomic {
  balance[i] -= amount;
  balance[j] += amount;
}
```

Distributed STM Library

```
TX_BEGIN;
t1 = TX_LOAD(node1, &balance[i]);
t1 = t1 - amount;
TX_STORE(node1, &balance[i], t1);
t2 = TX_LOAD(node2, &balance[j]);
t2 = t2 + amount;
TX_STORE(node2, &balance[j], t2);
TX_COMMIT;
```

PGAS models allow direct access to remote memory.
Global Memory Access = <Node-id, Local address>

`TX_LOAD:` Remote Transactional Read
`TX_STORE:` Remote Transactional Write



node1 node2

Distributed Memory (PGAS)

GTM: Global Transactional Memory

Motivation

Asynchronous STM

Abstractions

GTM Design

Scalability Results

Related Work

Future Directions

Asynchronous STM Abstraction

- STMs enforce a blocking STM abstraction
  - Return from STM call only after request fully satisfied
  - Performance ramifications:
    - Multicores: STM metadata management overheads
    - Distributed memory: Remote communication overheads
- Asynchronous abstraction helps resolve issue
  - Differentiate between when request is *issued* from when request is *expected to complete*
    - Simultaneous STM requests in-flight
  - Overlap remote latency with local computation and/or other independent communication
  - Reduce single-node STM overheads (future work)

Example: Bank Transaction

Synchronous (Blocking) STM Abstraction

```
TX_BEGIN;
t1 = TX_LOAD(node1, &balance[i]);
t2 = TX_LOAD(node2, &balance[j]);
t1 = t1 - amt;
t2 = t2 + amt;
TX_STORE(node1, &balance[i], t1);
TX_STORE(node2, &balance[j], t2);
TX_COMMIT;
```

`TX_LOAD` and `TX_STORE`:
Issue Transactional Read/Write request and wait for results to arrive. Remote communication latency affects performance.

Asynchronous (Non-blocking) STM Abstraction

```
TX_BEGIN;
TX_L_NB(t1, node1, &balance[i]);
TX_L_NB(t2, node2, &balance[j]);
TX_WAIT(t1);
t1 = t1 - amt;
TX_S_NB(node1, &balance[i], t1);
TX_WAIT(t2);
t2 = t2 + amt; ...
```

`TX_L_NB` and `TX_S_NB`:
Issue Transactional Read/Write request and return immediately.
`TX_WAIT`:
Wait for request to complete

GTM: Global Transactional Memory

Motivation

Asynchronous STM Abstractions

GTM Interface

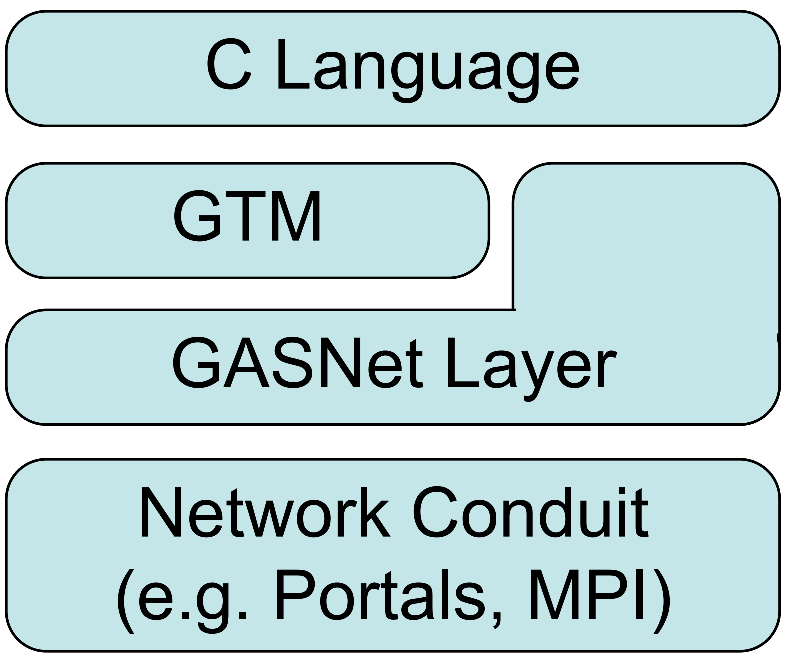
Scalability Results

Related Work

Future Directions

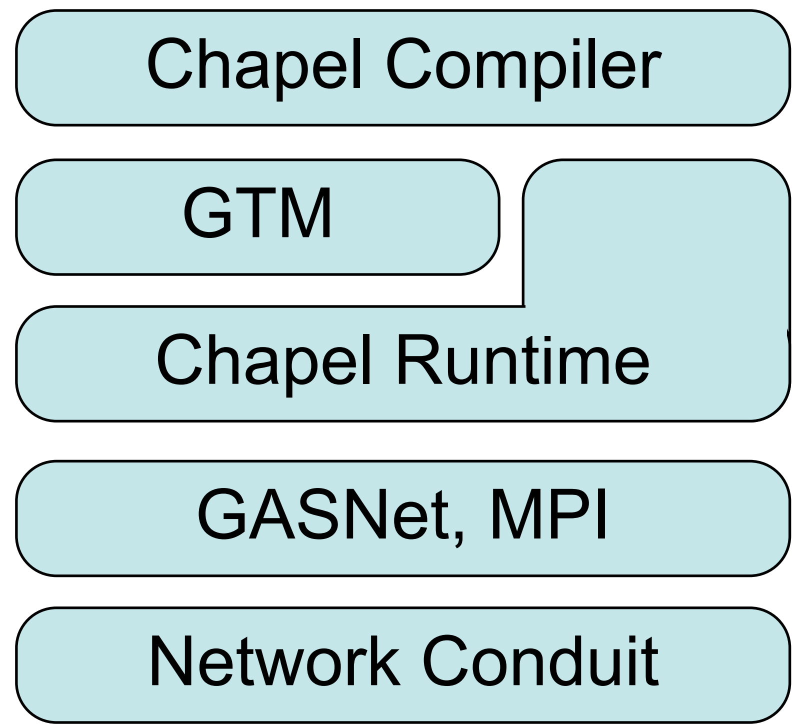
GTM Framework

Standalone Framework



Rest of this talk

Chapel-GTM Framework



On-going Work

GTM Execution Environment

- Fixed number of SPMD tasks created at startup
- SPMD tasks/nodes can be multithreaded
  - Exploit hardware thread-level parallelism
- Partitioned Global Address Space (PGAS) model
  - Transactional access of entire global address space
- Compatible with Chapel's runtime environment

GTM Interface Functionality

- Initialize and Clean-up STM runtime
- Start and Commit transactions
- Blocking and Non-blocking Variations
  - Transactional load/store:
    - Transfer data between global memory and private storage
  - Transactional malloc/free:
    - Dynamically manage local/remote transactional storage
  - Transactional Remote Procedure Call (RPC):
    - Execute user-level procedures on the target node
    - For exploiting locality (stay tuned...)
- Manage pending non-blocking requests

GTM Descriptors

- Transaction Descriptor or TDesc (tx):
  - Handle for identifying a transaction
  - Tracks private metadata describing the transaction
- Handle Descriptor or HDesc (op):
  - Handle for identifying a non-blocking request
  - NULL for synchronous/blocking requests
- Node Descriptor or NDesc:
  - Target node on whose context request must execute
  - Each operation has target source and node
    - If source and target are same, then operation is local else operation will generate remote communication

Managing Transactional State

GTM Procedure	Description
`tx = gtm_tx_create()`	Returns a new TDesc `tx`
`gtm_tx_destroy(tx)`	Destroys the transaction `tx`
`gtm_tx_begin(tx)`	Begin executing transaction `tx`
`gtm_tx_commit(tx)`	Attempt to commit transaction `tx`
`gtm_tx_abort(tx)`	Abort transaction `tx`
`op = gtm_op_create()`	Returns a new handle descriptor op
`gtm_op_destroy(op)`	Destory the handle op

- Transactions must be started and committed by same node
- All calls are local and blocking
  - Commit/Abort may implicitly generate messages

GTM Call Semantics

Call Semantics	HDesc (op)	NDesc (tgt)
Local Blocking	NULL	Source
Local Non-Blocking	Valid HDesc	Source
Remote Blocking	NULL	Remote
Remote Non-Blocking	Valid HDesc	Remote

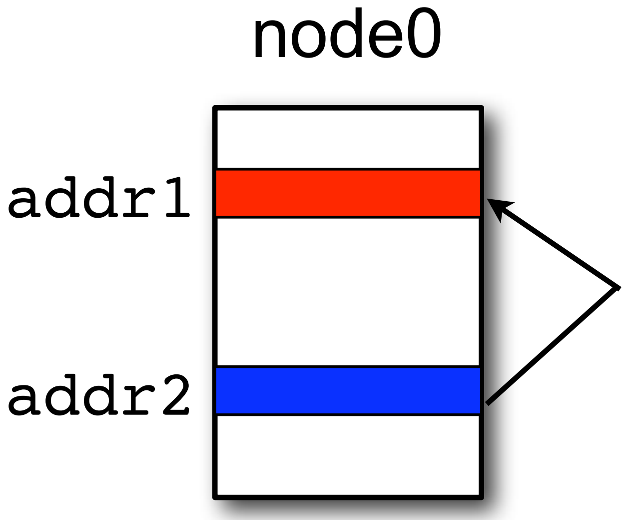
- HDesc and NDesc determine call semantics
  - HDesc: Blocking or Non-Blocking
    - Valid HDesc: No active request, Non-NULL
  - NDesc: Local or Remote operation

Transactional Load Interface

```
gtm_tx_load(tx, op, tgt, destAddr, srcAddr, size)
On node0:
    gtm_tx_begin(tx);
    gtm_tx_load(tx, NULL, node0, addr1, addr2, 4);
    ...
    gtm_tx_commit(tx);
```

Blocking Call

Local Operation



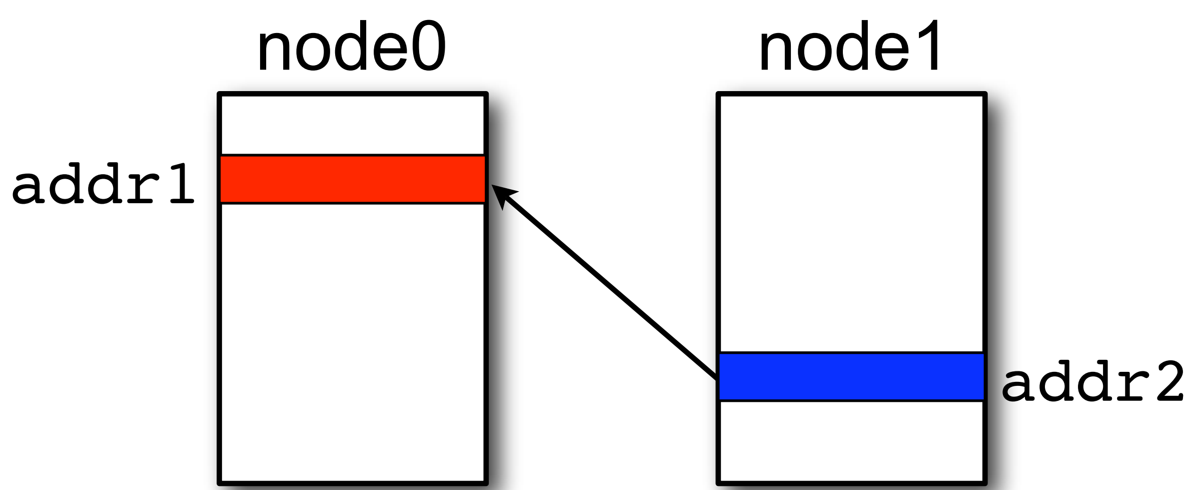
Transactional Load Interface

```
gtm_tx_load(tx, op, tgt, destAddr, srcAddr, size)
```

```
On node0:
   gtm_tx_begin(tx);
   gtm_tx_load(tx, NULL, node1, addr1, addr2, 4);
   ...
   gtm_tx_commit(tx);
```

Blocking Call

Remote Operation



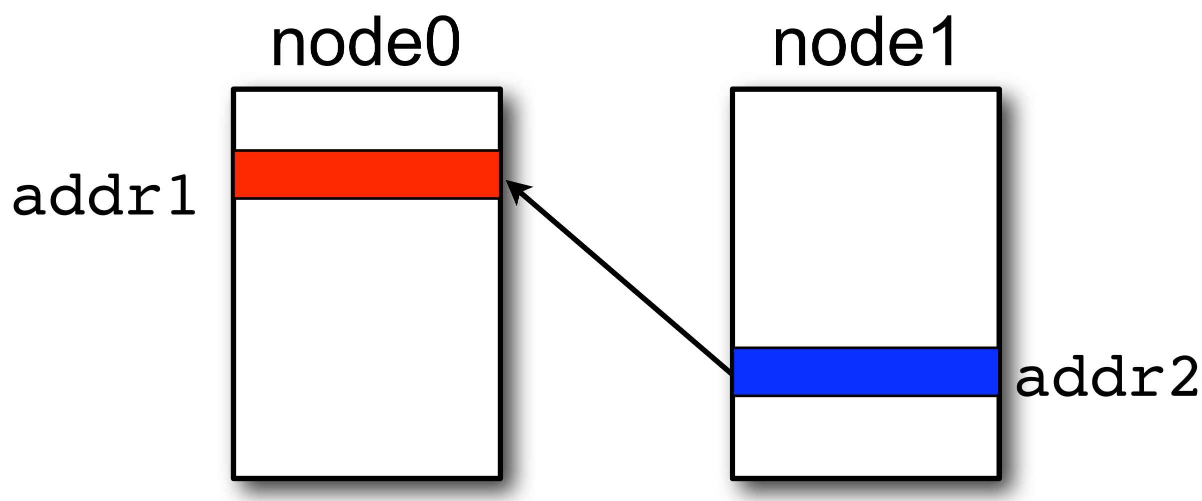
Transactional Load Interface

```
gtm_tx_load(tx, op, tgt, destAddr, srcAddr, size)
```

```
On node0:
  HDesc *op1 = gtm_op_create();
  gtm_tx_begin();
  gtm_tx_load(tx, op1, node1, addr1, addr2, 4);
```

Non-Blocking Call

Remote Operation



Managing Non-Blocking Requests

gtm_op_wait(tx,op)	Wait for request on op to complete. If op fails then abort tx.

```
On node0:
  gtm_tx_load(tx, op1, node1, addr1, addr2, 4);
  <computation or independent communication>
  gtm_op_wait(tx, op1);
```

gtm_op_test(tx,op)	Return status of request on op.

```
On node0:
  gtm_tx_load(tx, op1, node1, addr1, addr2, 4);
  ...
  gtm_op_test(tx, op1);
```

Transactional Data Management

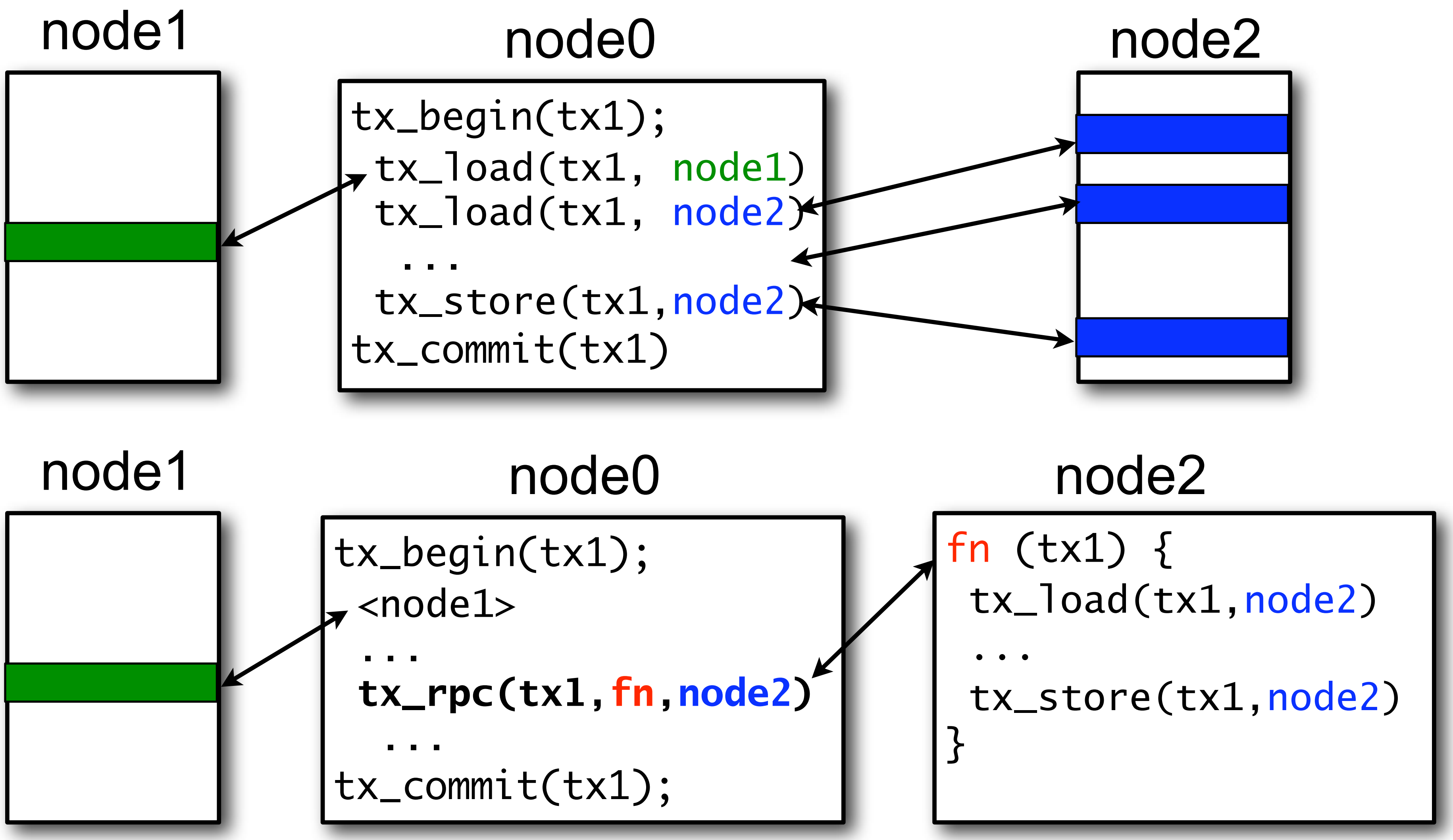
gtm_tx_store(tx, op, tgt, srcAddr, size, destAddr)
Transactional store of size bytes from *destAddr* on *tgt* to *srcAddr* on callee.

gtm_tx_malloc(tx, op, tgt, size, addr)
Transactional allocation of *size* bytes starting at *addr* on *tgt.*

gtm_tx_free(tx, op, tgt, size, addr)
Transactional free of *size* bytes starting at *addr* on *tgt.*

- Same call semantics as `gtm_tx_load`
  - Must be called inside transactional boundaries
  - Use the same calls for managing non-blocking requests

Transactioal RPC Mechanism



Transactional RPC Interface

```
gtm_tx_fn(tx, op, tgt, fName,
          iBuf, iSize, oBuf, oSize)
```

Execute *fName* on *tgt* node.
Local or Remote variations.
Blocking or Non-Blocking variations.
Input arguments: *iBuf* (size *iSize*)
Output results: *oBuf* (size *oSize*)

- `gtm_op_test` and `gtm_op_wait` for managing pending requests.
- Can be called from outside transactional boundary
  - Execute independent transactions on remote nodes

STM Algorithmic Choices

Feature	Description	Algorithmic Choice
Nesting semantics	nesting transactional blocks	Flat
Granularity	size of transactional data	Word
Conflict Detection	when conflicts are detected	Early
Write synchronization	how writes are handled	Deferred
Read synchronization	how reads are handled	Read-Versioning
Conflict tolerance	semantics for read	Validation
Forward Progress	completion guarantees	None

GTM: Global Transactional Memory

Motivation

Asynchronous STM Abstractions

GTM Interface

Scalability Results

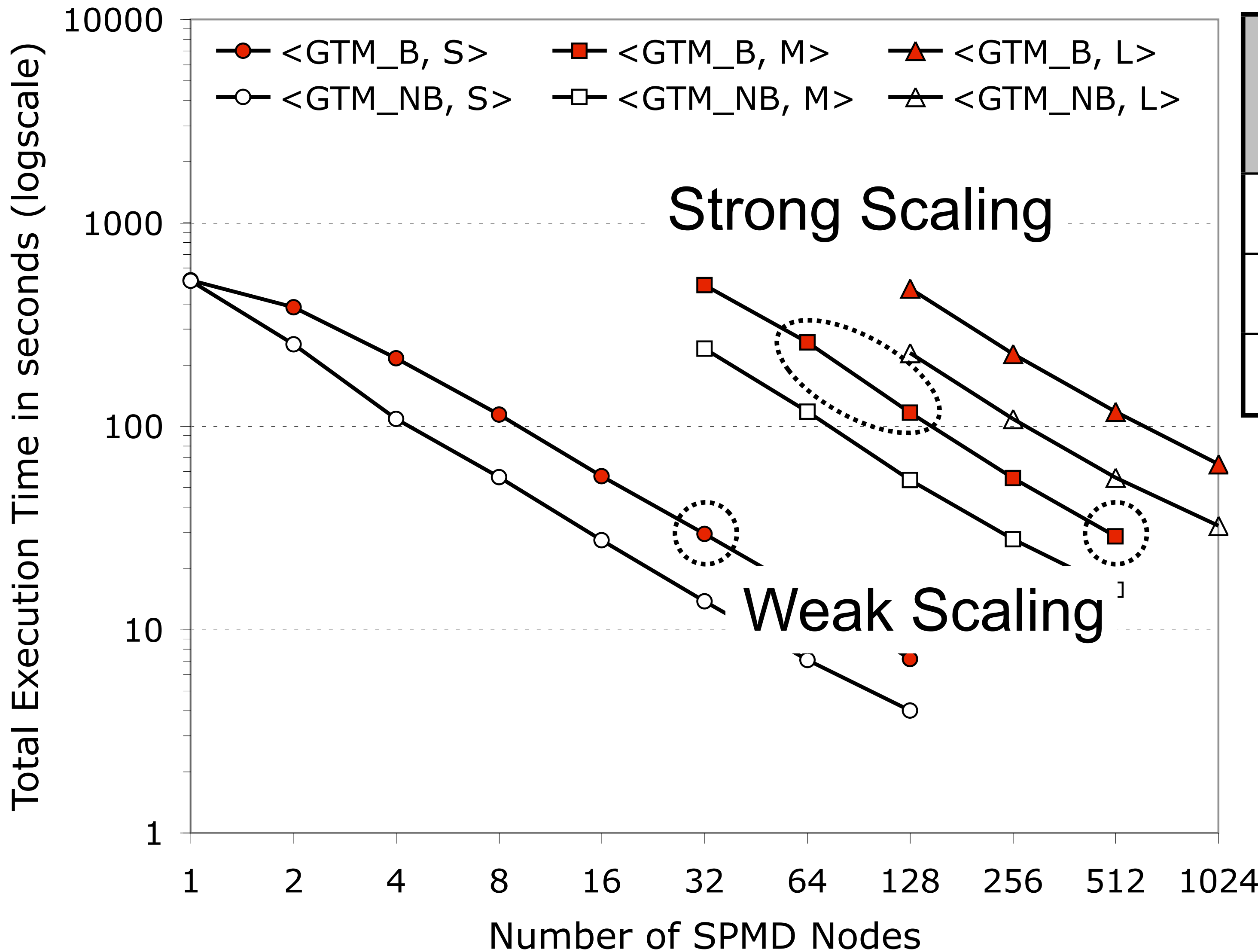
Related Work

Future Directions

Experimental Methodology

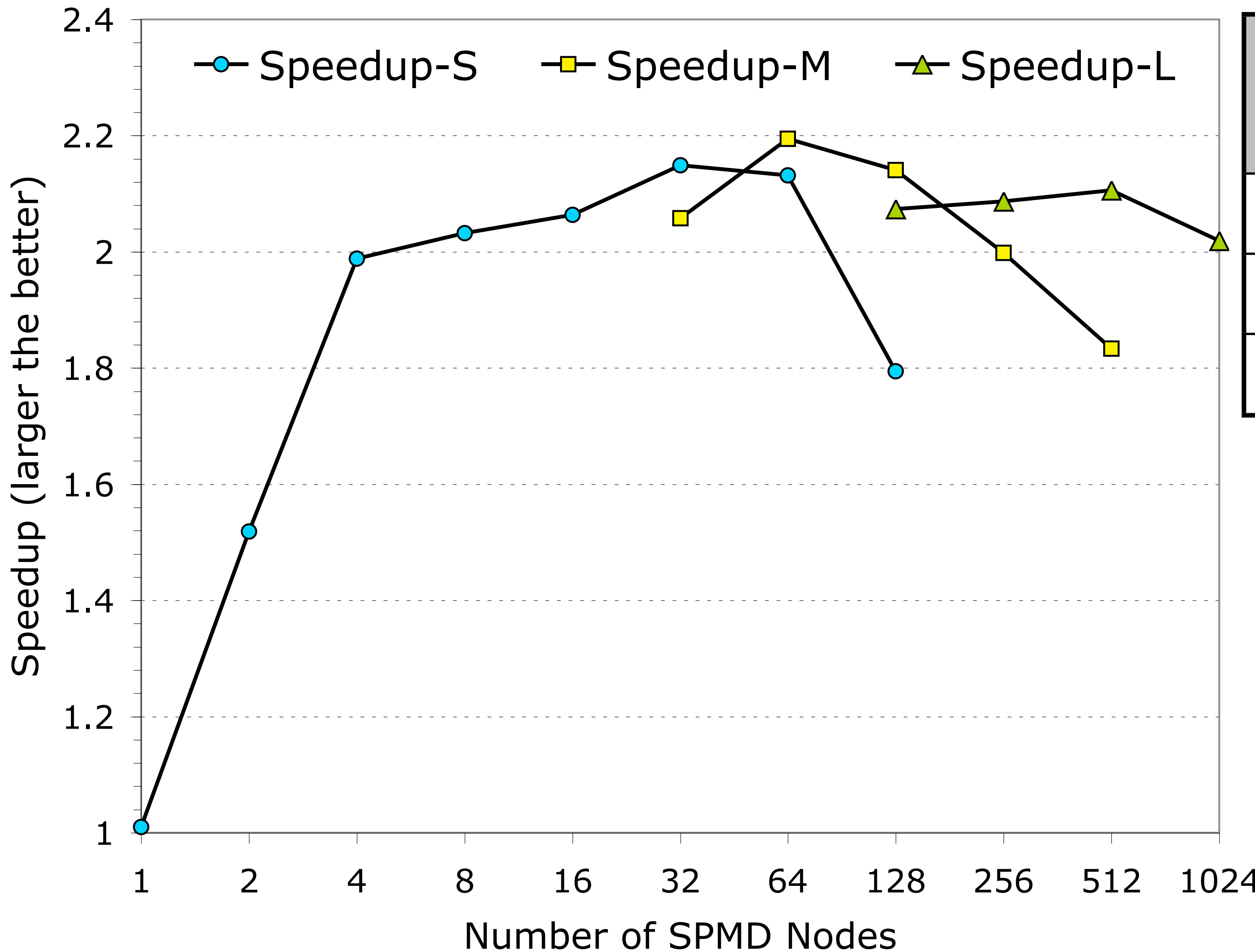
- ORNL NCCS Jaguar Cray XT4
  - 2.1 GHz Quad-core Opteron, 8GB memory
- Red-Black Tree Benchmark
  - Each node maintains balanced RB Tree
  - One thread per SPMD task
    - Insert, Delete, and Update operations across all nodes
- Additional results not presented
  - Priority Queue and Bank Transaction Benchmark
  - Effects of multithreading, Serialization issues in GASNet

Execution Time: Red-Black Tree



Total TX Commits	
S	2^{24}
M	2^{28}
L	2^{30}

Speedup: Red-Black Tree



Total TX Commits	
S	2^{24}
M	2^{28}
L	2^{30}

GTM: Global Transactional Memory

Motivation

Asynchronous STM Abstractions

GTM Interface

Scalability Results

Related Work

Future Directions

Related Work: Cluster-STM

Feature	GTM	Cluster-STM
Parallelism Environment	SPMD-Threads	Strict SPMD
Asynchronous Abstraction	Yes	No
Transactional Memory Region	Global Address Space	Limited to fixed segment
Transaction Identifier	TDesc	SPMD Id
STM Algorithms	One	Four

- Cluster-STM: Chapel's past collaboration with UIUC
  - PPoPP '08: Bocchino, Adve, and Chamberlain
- First to provide RPC with STM semantics

GTM: Global Transactional Memory

Motivation

Asynchronous STM Abstractions

GTM Interface

Scalability Results

Related Work

Future Directions

Future Directions

- Chapel Runtime
  - Under progress: Chapel-GTM runtime exploration
  - Use asynchronous abstraction to reduce scalar STM metadata management overheads
- Chapel Compiler
  - Implement **`Atomic`** keyword
    - Compiler optimizations
- Develop benchmarks to benefit from Chapel-GTM
  - Under progress: Bader MST, SAT solver, NAS UA
  - Suggestions and possible collaborations...

More information:

`chapel.cs.washington.edu`

Carpe TM!

Thank You.