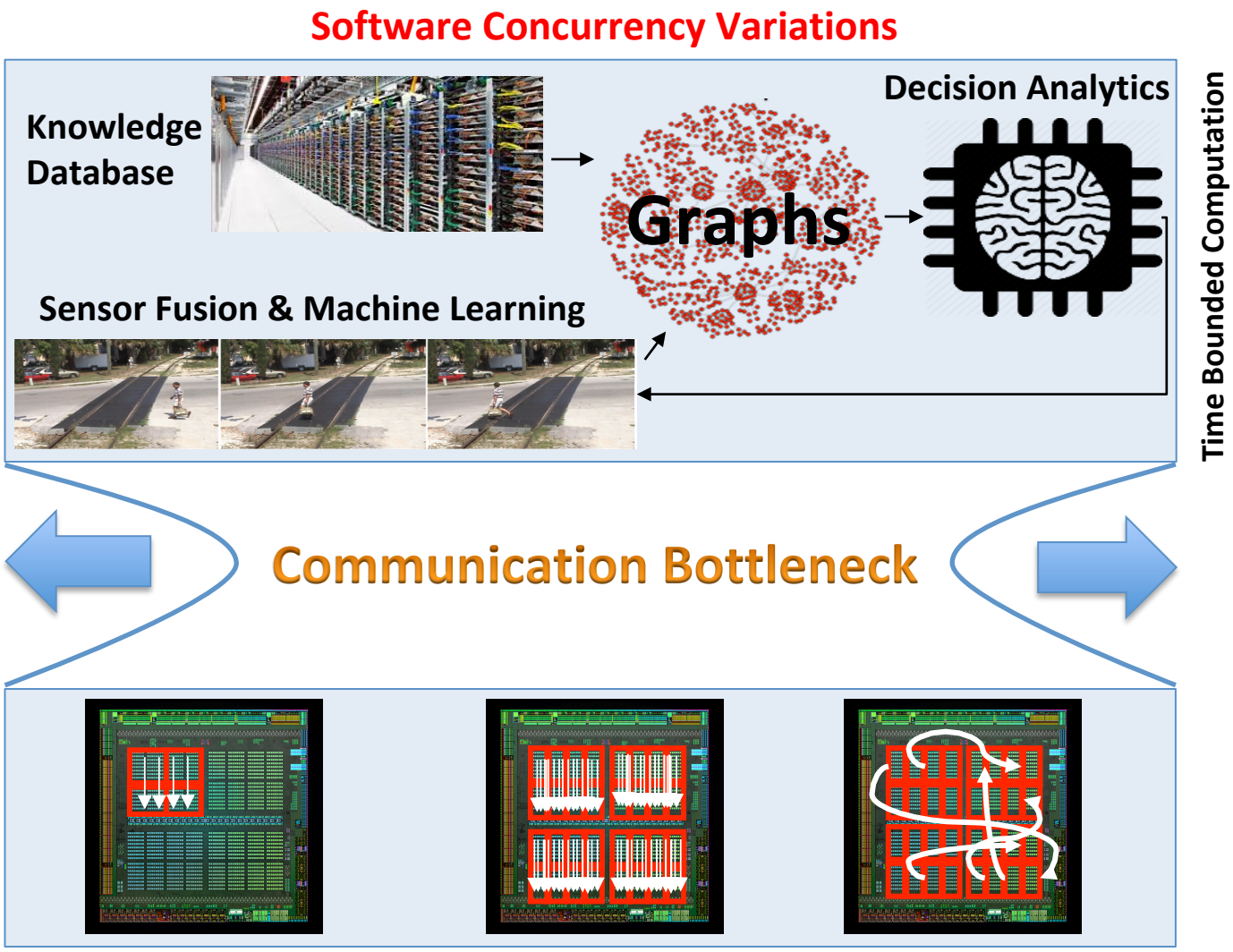
UCONN COMPUTER ARCHITECTURE GROUP

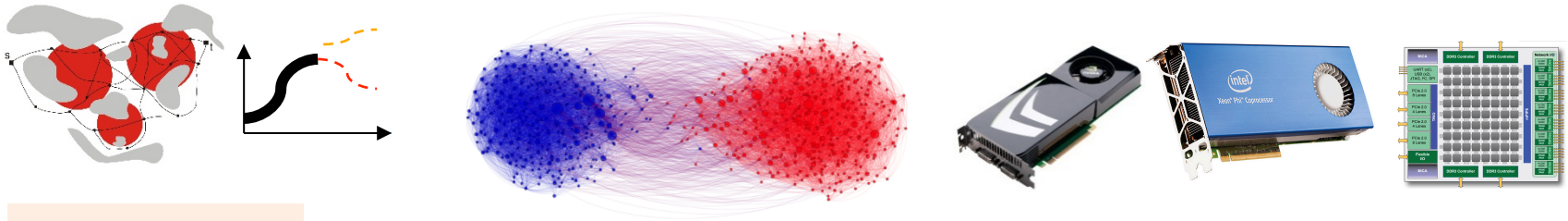
Accelerating Synchronization in Graph Analytics using Moving Compute to Data Model on Tilera TILE-Gx72

Halit Dogan[+], Masab Ahmad[+], Jose Joao*, Omer Khan[+]
[+] University of Connecticut, Storrs CT USA
* Arm Research, Austin, TX USA

The Concurrency Wall



Concurrent Processing of Graph Problems



Problems

Graphs

Algorithms

Complexity vs. Accuracy

Parallel Machines

Changing Input Graphs lead to variations

Various near-optimal algorithms

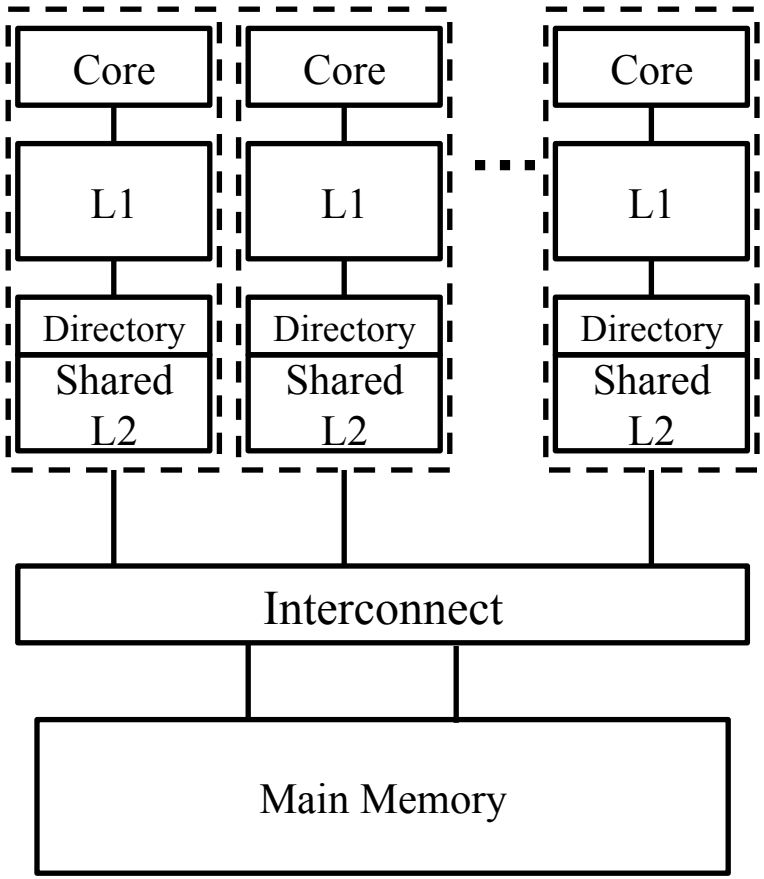
Performance bottlenecked by Communication

- Real apps, such as unmanned vehicles or intelligence queries have a specific execution model
  - Graphs are spatio-temporally streamed into the machine
  - Real-time constraints are implied to solve problems
  - Performance often relies on efficient scaling of communication

This paper explores a novel communication model on a 72-core Tilera Multicore

Background: Shared Memory Paradigm

Tiled Multicore Architecture

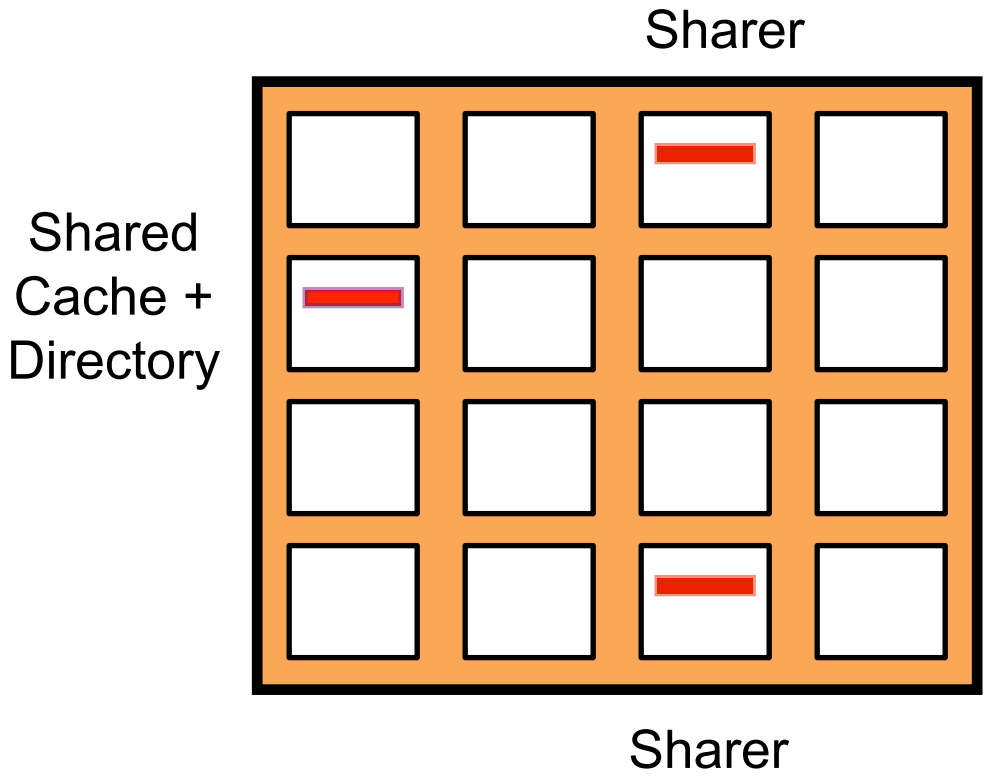


- Provides a unified view of memory to all cores
- Data access to shared memory via hardware level coherence protocol (directory based)
- Synchronization using hardware level atomic operations, and APIs that are exposed to the programmer via primitives, such as locks and barriers

The Advantage of Shared Memory
Directory-based Cache Coherence

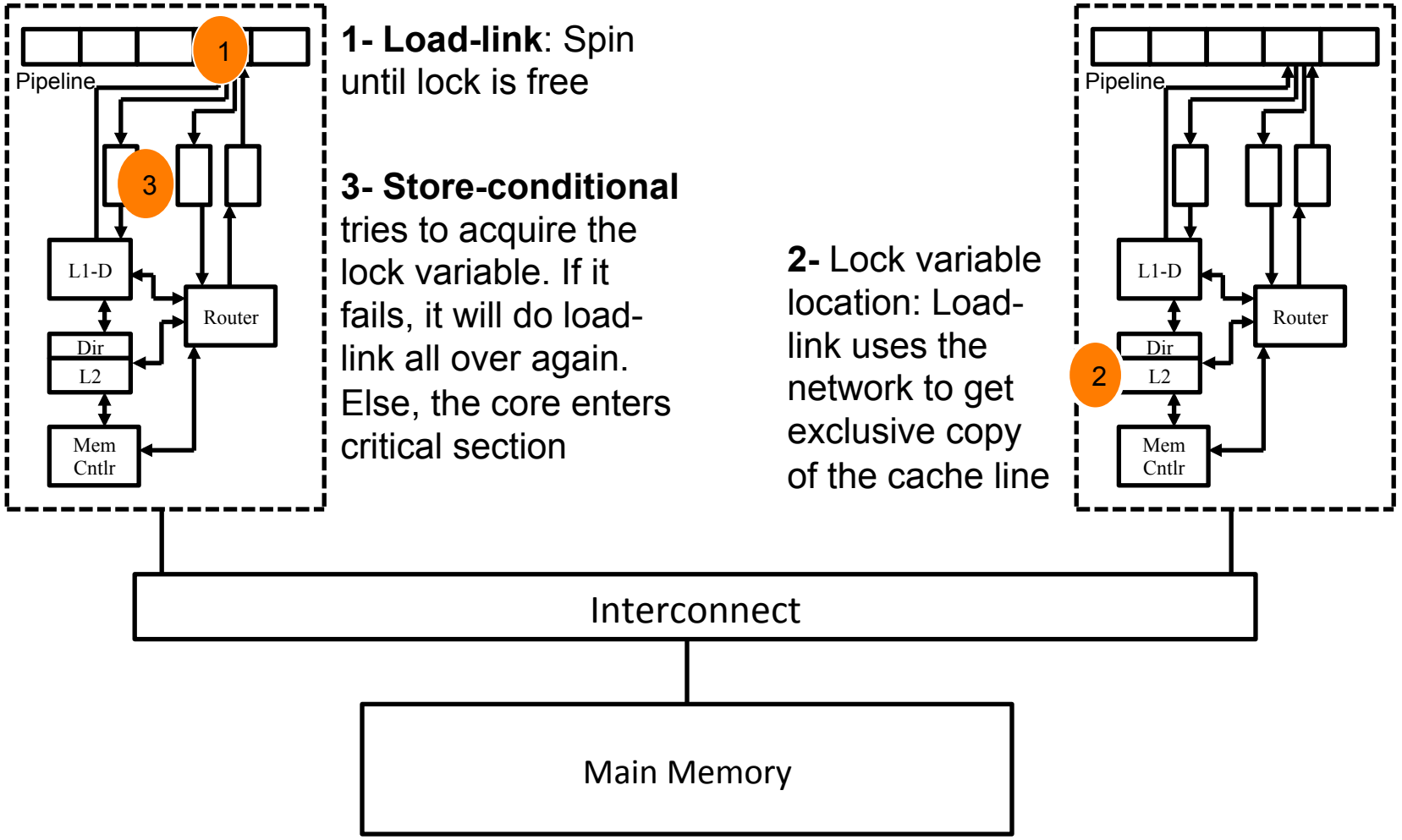
- **Private caches**: 1 or 2 levels
- **Shared cache**: Last-level

- Concurrent reads lead to replication in private caches
  - Directory maintains coherence for replicated lines

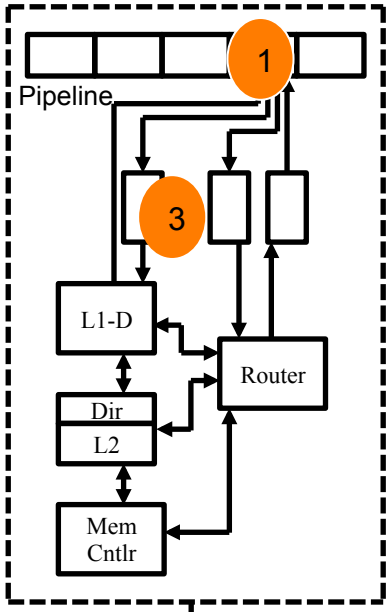


☺ **Exploits spatio-temporal locality for private and shared read-mostly data (cache line level replication)**

The Challenge with Shared Memory
Synchronization at the Hardware Level



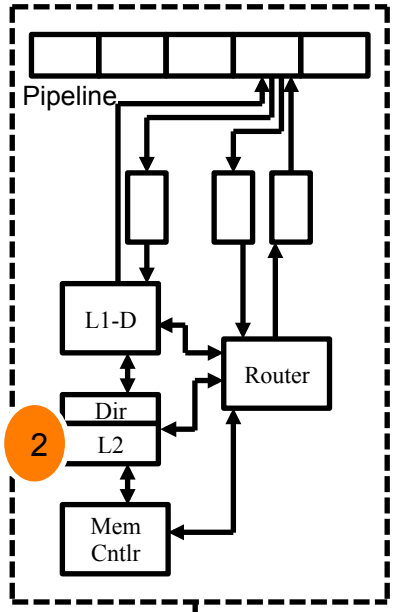
The Challenge with Shared Memory
Synchronization at the Hardware Level



1- Load-link: Spin until lock is free

3- Store-conditional tries to acquire the lock variable. If it fails, it will do load-link all over again. Else, the core enters critical section

2- Lock variable location: Load-link uses the network to get exclusive copy of the cache line



Generalization of Shared Memory Shortcomings:

- ☹ Inefficient for data with low spatio-temporal locality
- ☹ Increased on-chip communication and time spent waiting for expensive and repetitive events

Retries for contended shared data lead to wasteful invalidations, synchronous write-backs, cache line ping-pong

Efficient Communication Model

- Retain shared memory paradigm for data access, but utilize explicit communication as an *auxiliary method* for thread synchronization
  - Explicit communication instructions in the ISA for direct core-to-core communication (akin to ATOMICS)
    - Hardware-level **send** and **receive** instructions
- Which **Communication Model**?
  - Spin-based synchronization through atomic instructions based APIs, such as locks and barriers
  - Atomic instructions (far or near) that cater for software needs
  - Move computation to data (MC2D) by pining critical section work at a single location (core), and utilize send/receive messaging instructions to invoke execution (instruction(s) --- kernel(s))
    - A generalized way to do atomics!
    - ISA support for send and receive explicit messaging instructions
    - For details, see MC2D model in our IPDPS 2017 paper…

Moving Computation to Data Model
Method in TILE-Gx72

- Spatially distribute threads among *worker cores* and *service cores*. Do away with locks and offload critical section(s) to the service cores
- Explicit send message invokes critical section execution at the service core, which serializes critical section execution requests
  - Each critical section must be performed in bulk at a single service core without violating atomicity (if needed)
  - Data structures being operated in a critical section *may* be distributed among multiple service cores to exploit concurrency in moving computation model
  - Send *with* and *without* acknowledgment enables programmable way to manage consistency requirements versus performance tradeoffs

```
<< Spin Lock Implementation >>

Worker Thread Job
For each node v:
    For each neighbor u:
        spin_mutex_lock(u);
        Critical Code Section
        spin_mutex_unlock(u);
```

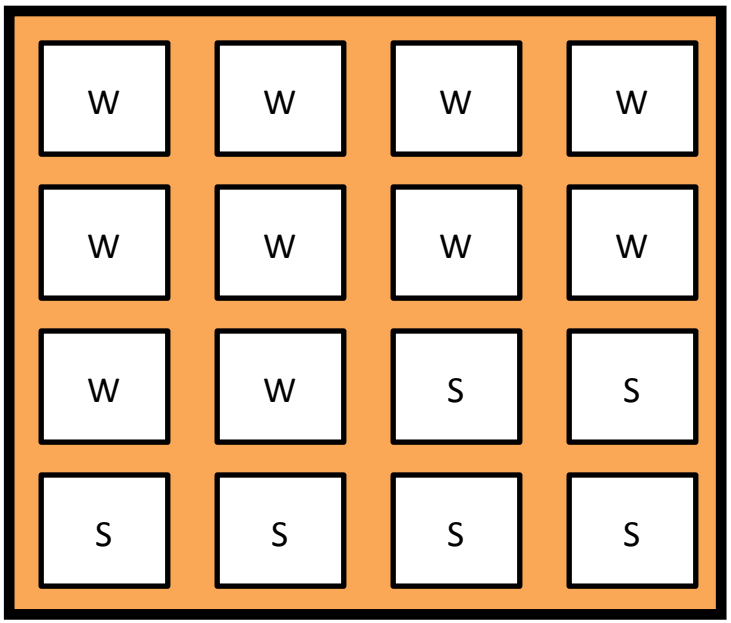
```
<< MC2D Implementation >>

Worker Thread Job
coreid = get_service_core(u)
send_x(coreid,data_1,...,data_x)


Service Thread Job
while !terminate do
    data_1 = receive()
    ...
    data_x = receive()
    Critical Code Section
```

Moving Computation to Data Model
Spatial Distribution of Service and Worker Cores

- A single service core may become bottleneck, hence multiple cores can be assigned as service cores as long as shared data can be distributed among these cores

- Best performing mapping varies for different applications

Worker and service cores are **spatially** distributed on available cores



W: Worker Core
S : Service Core

Blocking / Non-Blocking Communication

- **Non-blocking Communication**
  - Sender core continues execution as soon as the **send** instruction commits in the core pipeline
  - **receive** is blocking and consumes a send message at the destination core

- **Blocking Communication**
  - **send** instruction follows a blocking **receive** instruction that is an explicit reply from destination core
  - Destination core implements a **receive** instruction followed by a **send** instruction to explicitly send a reply message to the sender core

Moving Computation to Data Model
Pros and Cons

- ☺ Removes instruction retries and cache line ping-pongs for contended shared data → overlaps Communication and Computation; improves data locality!
  - As *core counts* increase, the network bottleneck gets worse and MC2D model is expected to deliver better communication scalability
- ☺ Generalizes arbitrary atomic operations and critical sections with only send/receive instructions
- ☹ Need to load-balance worker and service cores
  - We will look at a *heuristic to load balance*, as well as a *temporal architecture* that assumes worker and service threads are multi-threaded on each core

Parallel Shortest Path Benchmark:
Traditional Locks and Barriers

```
Initialize distance_array(D)
(v; u) is a vertex, neighbor pair

<< Parallel Relax Function >>
for (each neighbor, u of v) do
    if D[v] + D[v; u] < D[u] then
        Lock (u)
        if D[v] + D[v; u] < D[u] then
            D[u]  D[v] + D[v; u]
        UnLock (u)
```

Implementation checks if the vertex needs to be relaxed before invoking fine-grain synchronization event

Threads synchronously update the distances of the connected vertices in a given range. The range is updated synchronously across all threads (not shown here).

Parallel Shortest Path Benchmark:
Moving Compute to Data (MC2D)

Initialize distance_array(D)
(v; u) is a vertex, neighbor pair

<< Worker Thread: Parallel Relax Function >>

for (each neighbor, u, of v) do
    if D[v] + D[v; u] < D[u] then
        send(D[v]; u)

<< Service Thread: Parallel Relax Function >>

recv(D[v]; u) – D[v] is arg1; u is arg2
u = Windex[arg2]
D_tmp = arg1
if D_tmp + D[v; u] < D[u] then
    D[u] = D_tmp + D[v; u]

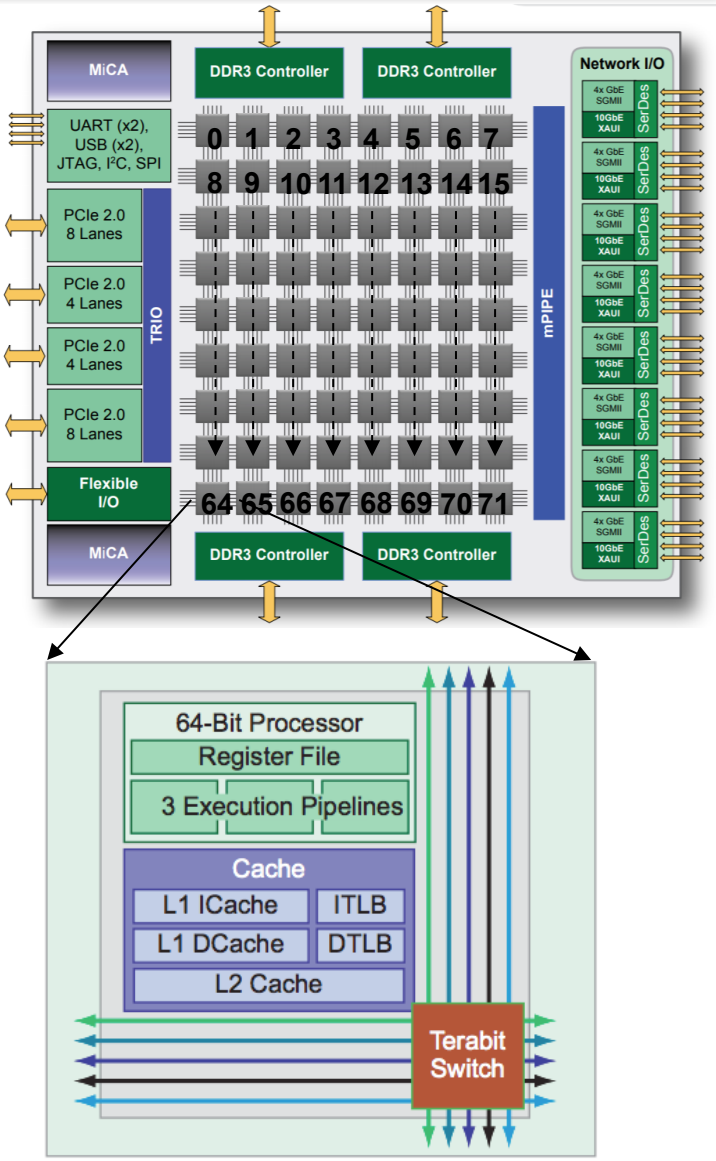
Test on D-array is retained for work efficient implementation

Locks are removed. Instead *send* message communicates the vertex to be relaxed to the service thread

Each service thread manages a set of vertices, and atomically update distance for the requested vertex

Evaluation Methodology – TILE-Gx72

- 72 VLIW (3-way) cores; ~23 MB caches (2-level cache hierarchy per tile)
- Load/Store: directory-based cache coherence for data movement
- Atomic instructions used to build synchronization primitives
  - CAS, fetch-and-add etc.
- In-hardware send/receive using NoC's User Defined Network (UDN) → Enables MC2D model



Evaluation Benchmarks and Inputs

- Graph Workloads

Fine-grain Communication	Coarse-grain Communication
Single Source Shortest Path (SSSP)	Page Rank
Triangle Counting (TC)	Connected Components (CC)
Breadth First Search (BFS)	Community Detection (COMM)

- Input Graphs

Input Graph	# Vertices	# Edges	Degree
Mouse brain	562	0.5M	1027
CA-Road Network	1.9M	5.5M	2.8
Facebook	2.9M	42M	14.3
LiveJournal	4.8M	85.7M	17.6

Key Objectives for Tilera Tile-Gx72 Study

1. Evaluate performance scaling for various communication models: Spin, Atomics, and Moving Compute to Data (MC2D)

2. Micro-benchmarks to show intuition for how MC2D is an effective communication model

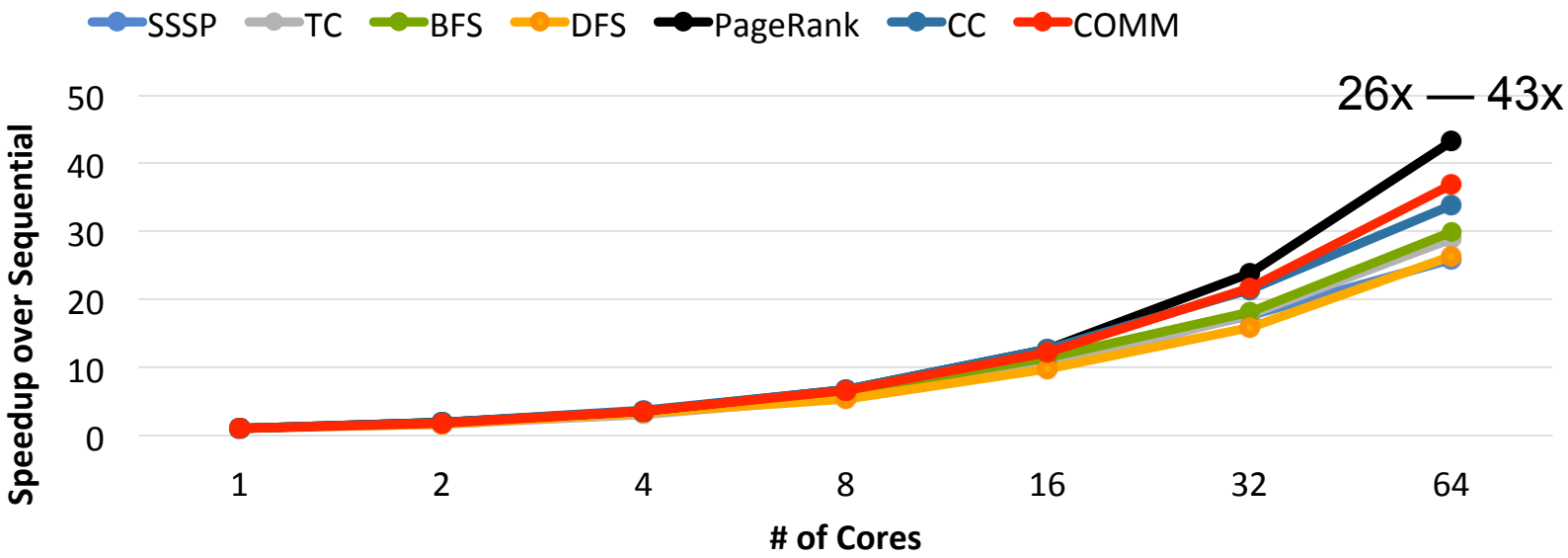
3. Demonstrate Moving Compute's capability to overlap communication with computation

4. Demonstrate hardware cache coherence is still needed for efficient data movement at fine granularity

5. Demonstrate why cache coherence based inter-core queues is not as efficient as in-hardware messaging

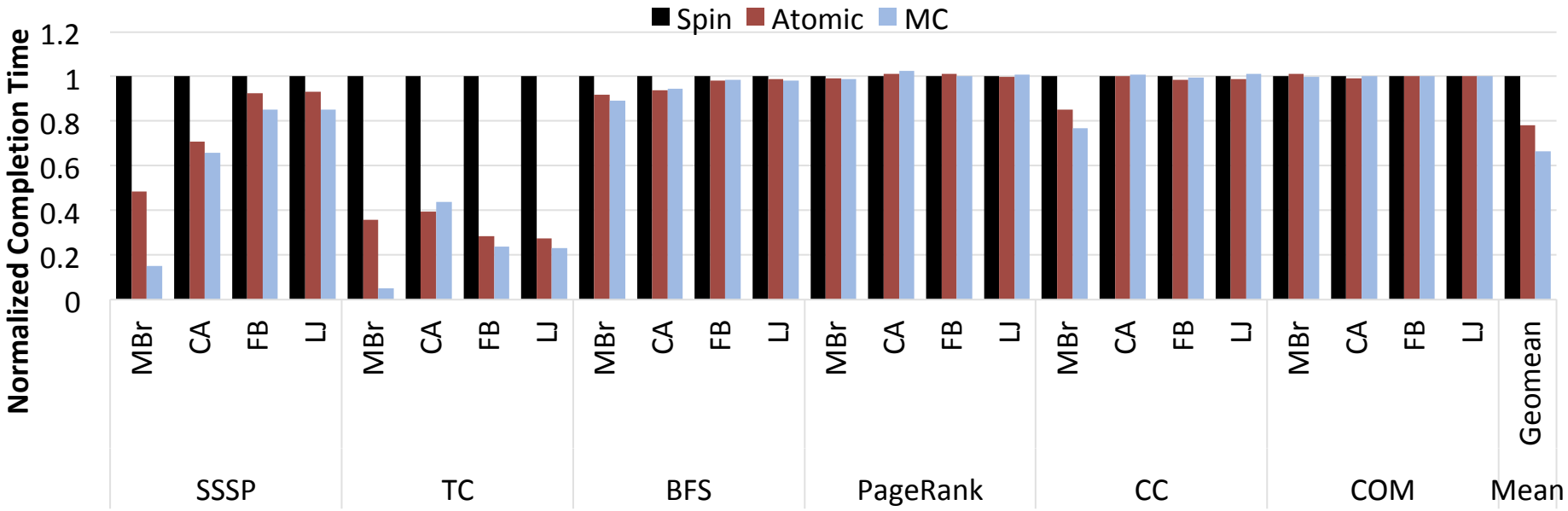
6. Demonstrate superiority of Moving Compute model as core counts increase

Does Spin Scale for Graph Benchmarks?



- Spin based locks implemented using "*compare-and-swap" atomic* instruction with an exponential back-off mechanism to mitigate instruction retries
- Tilera also offers queue based locks in which threads are placed into a queue until the lock variable is available
  - Trades off retries with additional latency to push locks to requesters
  - Our experiments do not show notable performance difference between two implementations for graph benchmarks

Spin vs. Atomic vs. Moving Compute at 64 cores



- Atomic model consistently outperforms the spin-model since all cores perform atomics concurrently (far atomic), hence better parallelism than Spin
  - Fine grain synchronization (SSSP, TC, BFS) leads to larger performance variations
- MC2D improves over Atomic model:
  - It eliminates expensive data ping-pongs and improve data locality
  - Exploit concurrency using the non-blocking communication that overlaps communication with computations

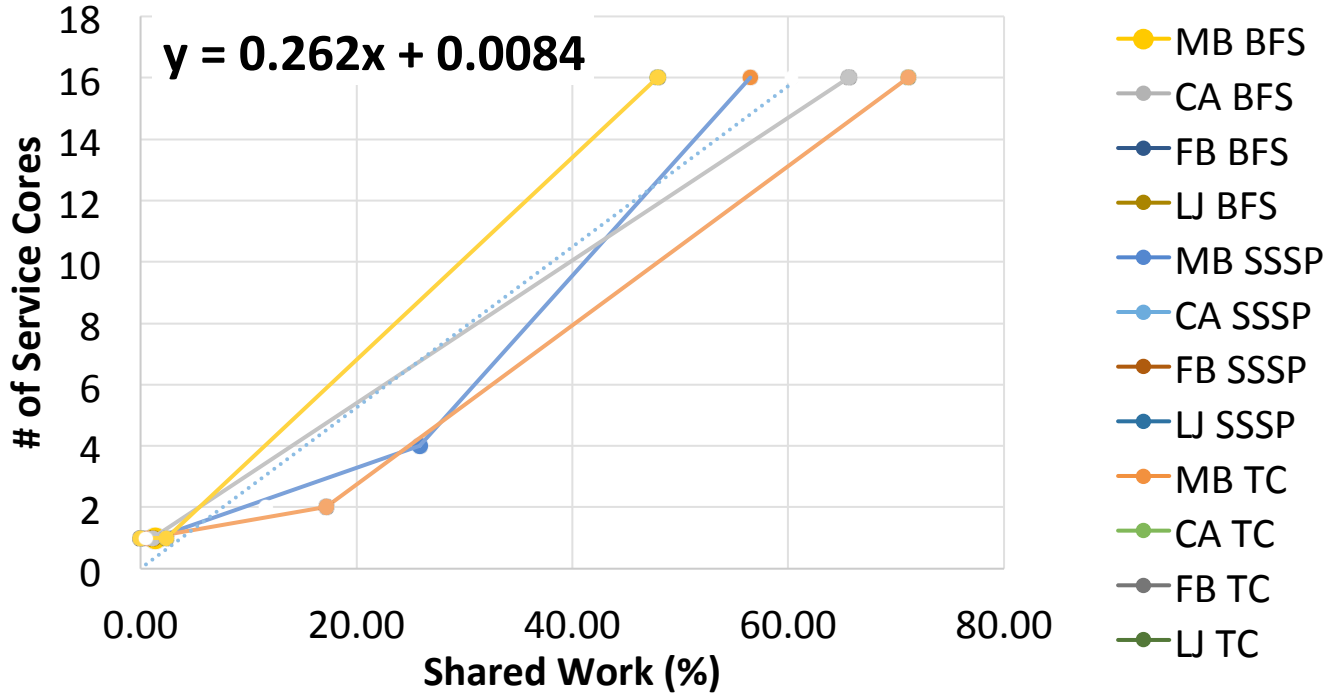
MC2D model accomplishes 15% better performance as compared to the Atomic Model

Determining the Right Service Core Count

- Either an exhaustive sweep study needs to be done
  - Time consuming as the core count increases

- Or use a heuristic model: MC2D model moves shared work to service core(s), hence the amount of shared work done in a algorithm can determine the right service core count
  - **Shared Work:** Track the time spent in critical section(s) with respect to total completion time
  - *Intuition*: If the shared work percentage is high, the benchmark requires more service cores

Shared Work driven heuristic



- Best performing service core count correlates with the shared work
- Based on the amount of shared work, service core count can be determined using a simple linear model

Key Objectives for Tilera Tile-Gx72 Study

1. Evaluate performance scaling for various communication models: Spin, Atomics, and Moving Compute to Data (MC2D)

2. **Micro-benchmarks to show intuition for how MC2D is an effective communication model**

3. Demonstrate Moving Compute's capability to overlap communication with computation

4. Demonstrate hardware cache coherence is still needed for efficient data movement at fine granularity

5. Demonstrate why cache coherence based inter-core queues is not as efficient as in-hardware messaging

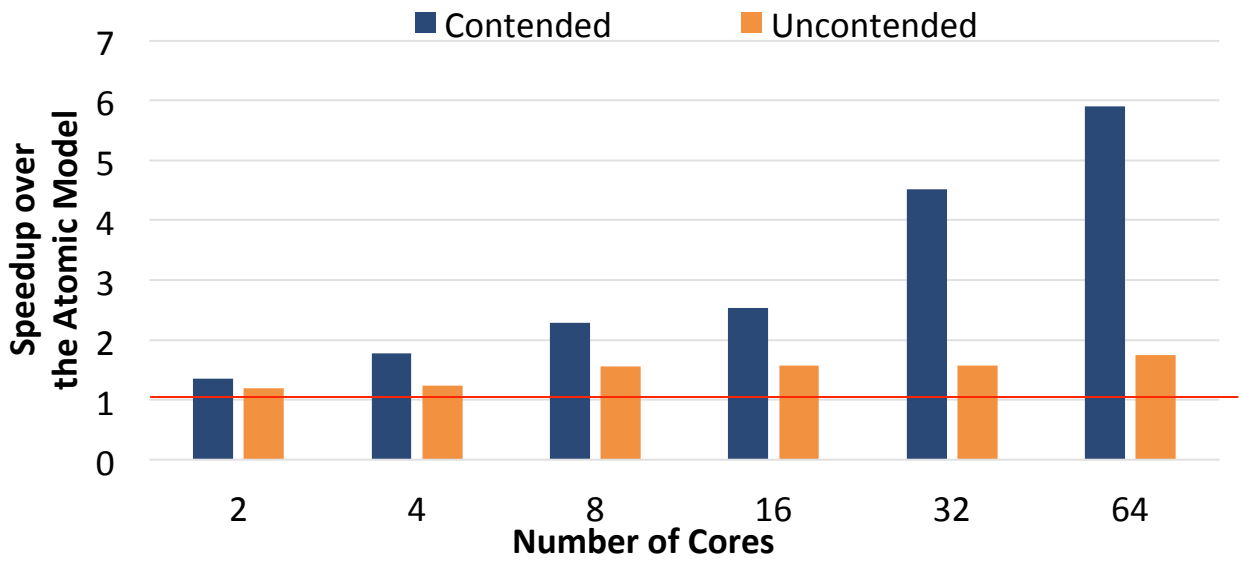
6. Demonstrate superiority of Moving Compute model as core counts increase

Barrier Micro-bencmark

- Spin, Atomic and MC2D barriers are evaluated for two cases
  - Contended barrier
  - Uncontended barrier
- **Contended:**
  - Threads reach the barrier at similar time stamps
  - DummyWork () function is removed in this case
- **Uncontended:**
  - Threads reach the barrier at different times
  - Each thread performs some **dummy work** for random number of iterations before hitting the barrier

```
<< Barrier Benchmarking >>

// Get a random iteration
// count for DummyWork
random = rand ();

For 0 to Iteration:
  // Synchronize Threads
  // before measurement
  barrier_wait ();

  DummyWork (random);

  start = get_cycles();

  barrier_wait ();

  stop = get_cycles ();
  time += stop-start;

CalculateAverage (time);
```

Results – Barrier



- **Contended barrier:** MC2D is ~6x faster than Atomic based barrier at 64 cores as a result of ping-ponging of the shared barrier variable

- **Uncontended barrier:** Even though threads reach the barrier at different times as a result of back-off mechanism, they still ping-pong the barrier variable which is eliminated with MC2D

Reduction Micro-benchmark

```
<< Barrier Benchmarking >>

barrier_wait ();

start = get_cycles();

For 0 to NumReductions:
  reduction (data, val);

stop = get_cycles ();
time += stop-start;

barrier_wait ();

CalculateMax (time);
```

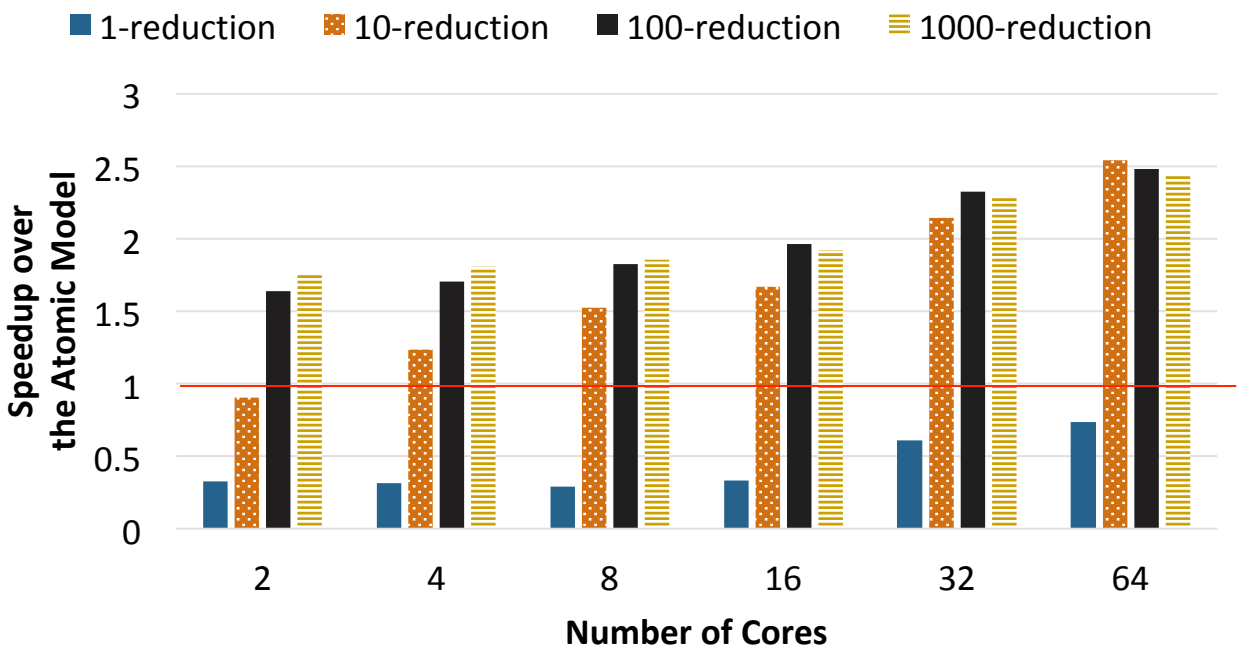
```
<< Reduction Spin-Lock >>
tmc_spin_mutex_lock(&lock);
data+=val;
tmc_spin_mutex_unlock(&lock)
```

```
<< Reduction Atomic >>
arch_fetch_add(&data, val);
```

```
<< Reduction MC2D >>

If Core 0:
  data += val;
  For 0 to NumCores-1:
    data += tmc_udn0_receive ();
Else:
  tmc_udn_send_1 (Core0, UDN0, val);
```

- Non-blocking reduction using Spin, Atomic and MC2D models are evaluated
  - Number of threads are varied from 2 to 64
  - Number of reductions per thread is varied from 1 to 1000 at power of 10 increments

Results – Reduction



- At single reduction, Atomic model outperforms the MC2D at all core counts because MC2D cannot hide the communication overheads for a single reduction per thread, and Atomic model exploits parallelism across all cores
- Increasing the number of reductions per thread helps to hide much of the communication latency, hence MC2D starts to provide better performance

Key Objectives for Tilera Tile-Gx72 Study

1. Evaluate performance scaling for various communication models: Spin, Atomics, and Moving Compute to Data (MC2D)

2. Micro-benchmarks to show intuition for how MC2D is an effective communication model

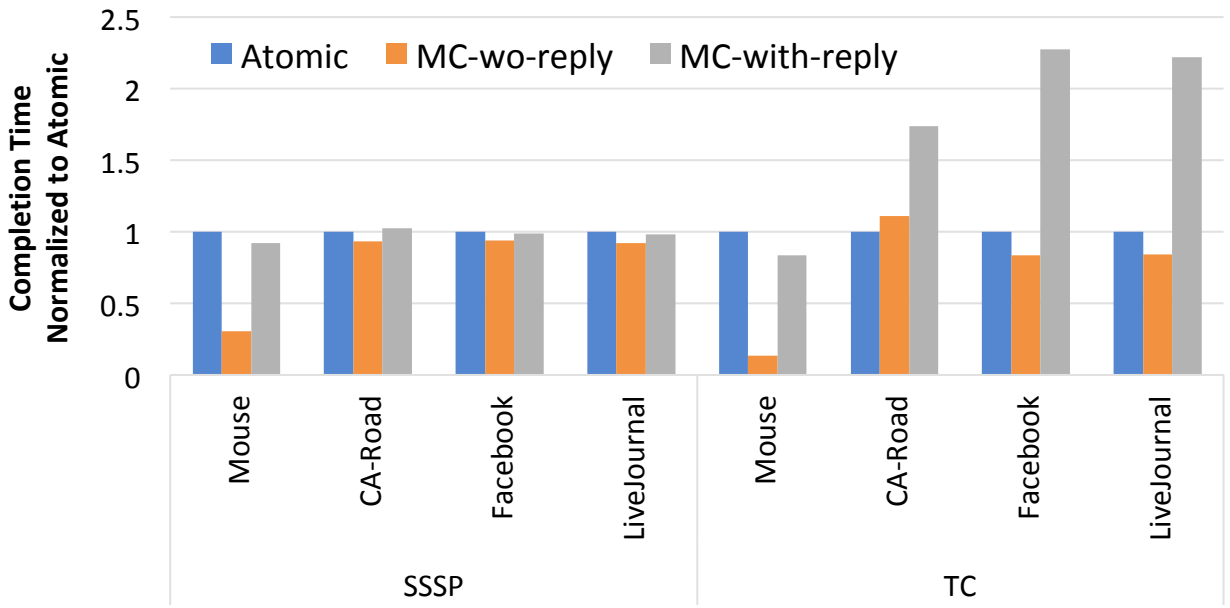
3. **Demonstrate Moving Compute's capability to overlap communication with computation**

4. Demonstrate hardware cache coherence is still needed for efficient data movement at fine granularity

5. Demonstrate why cache coherence based inter-core queues is not as efficient as in-hardware messaging

6. Demonstrate superiority of Moving Compute model as core counts increase

Exploiting Parallelism with MC2D Model



- Send without explicit acknowledgment (non-blocking communication) enables an efficient way to hide communication stalls by letting worker cores proceed with other work
  - MC2D overlaps communication with computation at fine granularity, while ensuring consistency needs when required

Key Objectives for Tilera Tile-Gx72 Study

1. Evaluate performance scaling for various communication models: Spin, Atomics, and Moving Compute to Data (MC2D)

2. Micro-benchmarks to show intuition for how MC2D is an effective communication model

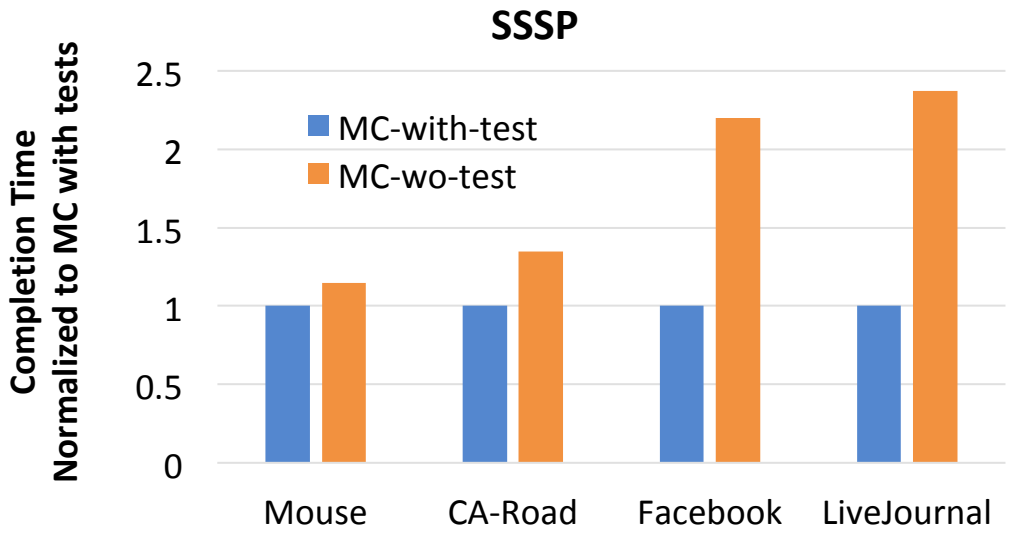
3. Demonstrate Moving Compute's capability to overlap communication with computation

4. **Demonstrate hardware cache coherence is still needed for efficient data movement at fine granularity**

5. Demonstrate why cache coherence based inter-core queues is not as efficient as in-hardware messaging

6. Demonstrate superiority of Moving Compute model as core counts increase

MC2D Model and Cache Coherence



- Several graph algorithms perform checks on shared data to determine the need for fine-grain synchronization. These tests at the hardware level rely on efficient hardware cache coherence support to make the implementation work efficient
- Without checks, the benchmark eschews coherence misses but pays additional work penalty, as seen by the impact on performance

Key Objectives for Tilera Tile-Gx72 Study

1. Evaluate performance scaling for various communication models: Spin, Atomics, and Moving Compute to Data (MC2D)

2. Micro-benchmarks to show intuition for how MC2D is an effective communication model

3. Demonstrate Moving Compute's capability to overlap communication with computation

4. Demonstrate hardware cache coherence is still needed for efficient data movement at fine granularity

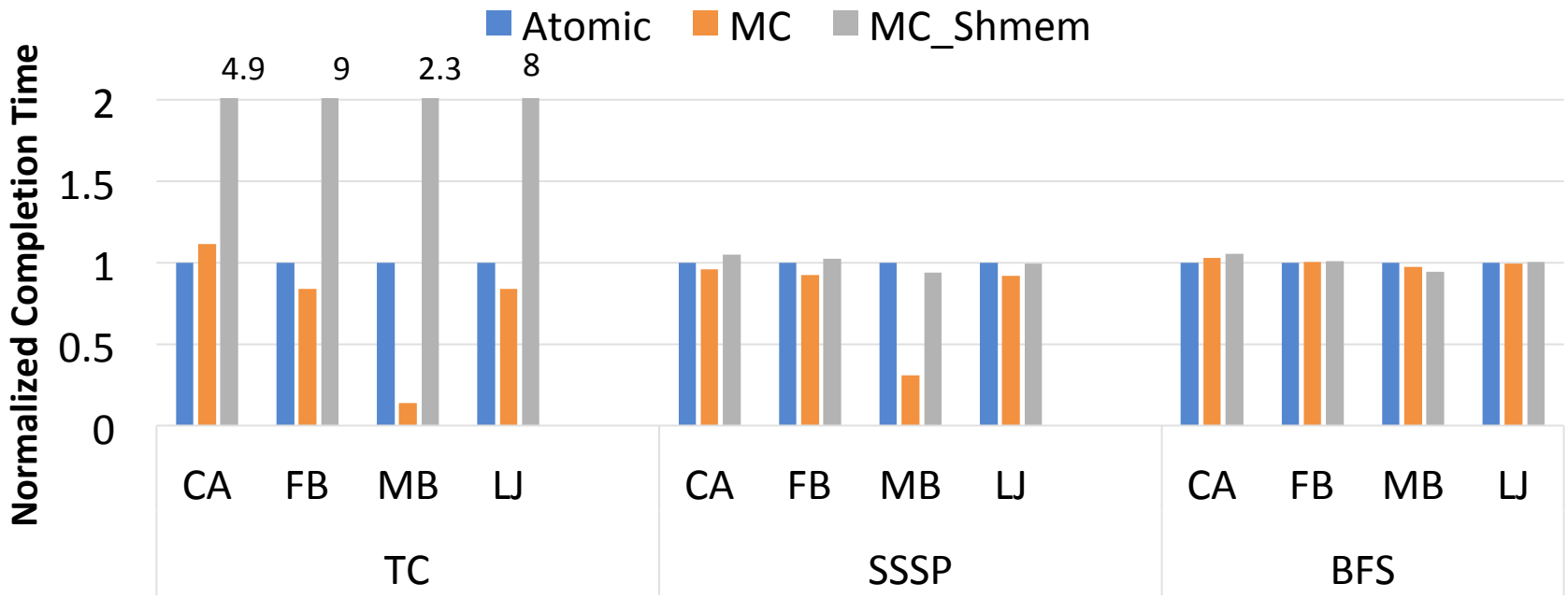
5. **Demonstrate why cache coherence based inter-core queues is not as efficient as in-hardware messaging**

6. Demonstrate superiority of Moving Compute model as core counts increase

MC2D using Shared Memory (MC_Shmem)

- A shared buffer per service core is utilized to send/receive data between worker and service cores
  - Each element of a shared buffer consists of data and the respective flag
  - A worker puts the data into next available slot in the shared buffer, then sets the flag for that entry
  - The service core spins over the flag of that slot until it is set, then reads the data
- The index of the shared buffer is protected with an atomic instruction to make sure that multiple worker cores do not write to the same entry
- For iterative algorithms, the index to shared buffers are reset to initial starting point to make sure that the same buffer is utilized in the next iteration

MC_Shmem versus MC2D In-Hardware Messages



- MC_shmem shared buffer to send/receive messages ping-pongs between service and worker cores, leading to performance loss compared to both Atomic and MC2D (using in-hardware messaging)
  - When contention on shared data is low, performance impact is less prominent since less messages are involved to perform synchronization

Key Objectives for Tilera Tile-Gx72 Study

1. Evaluate performance scaling for various communication models: Spin, Atomics, and Moving Compute to Data (MC2D)

2. Micro-benchmarks to show intuition for how MC2D is an effective communication model

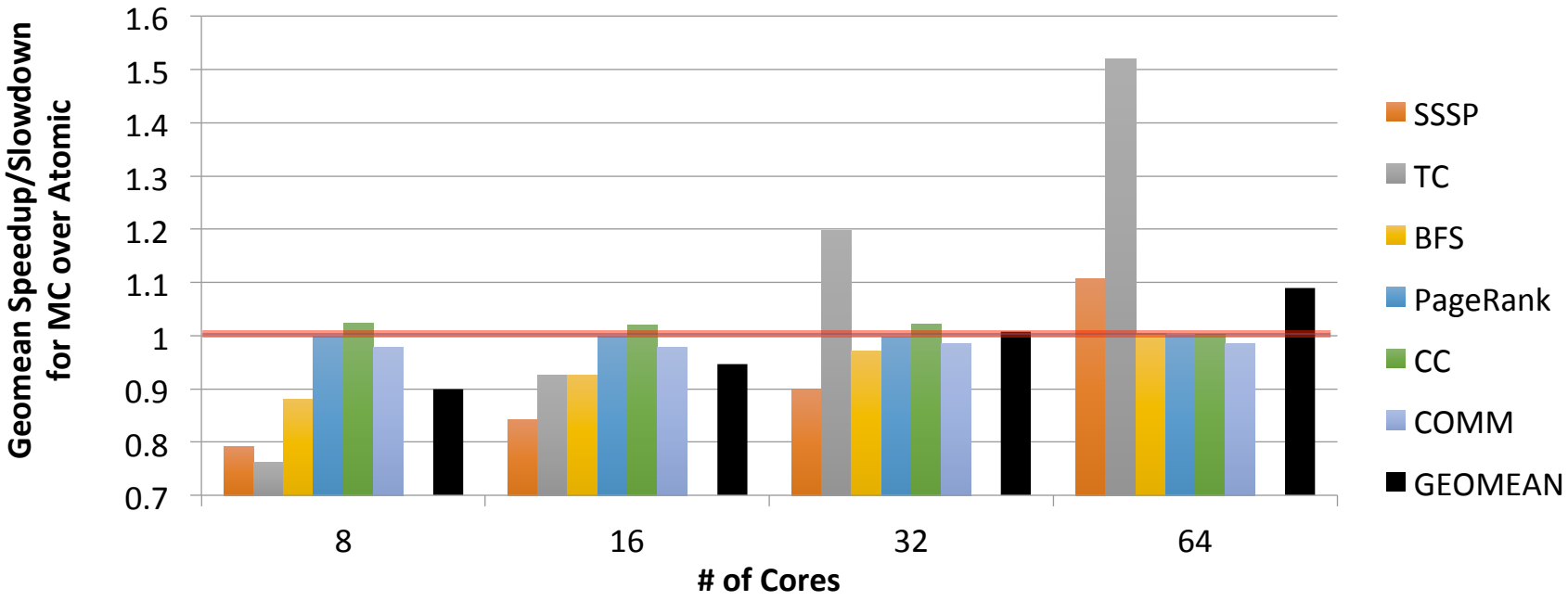
3. Demonstrate Moving Compute's capability to overlap communication with computation

4. Demonstrate hardware cache coherence is still needed for efficient data movement at fine granularity

5. Demonstrate why cache coherence based inter-core queues is not as efficient as in-hardware messaging

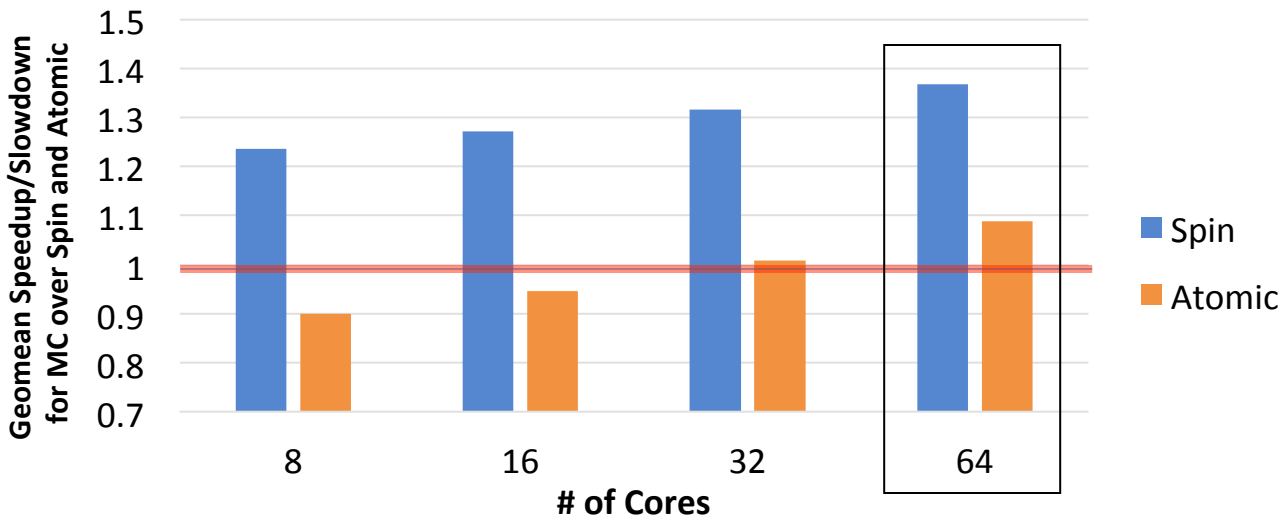
6. **Demonstrate superiority of MC2D model as core counts increase**

Atomic vs. Moving Compute Core Scaling Trends



- MC2D model *underperforms* Atomic model at lower core counts, specifically for benchmarks with fine-grain synchronization
  - Moving compute observes higher load imbalance between worker and service cores at low core counts. Atomics on the other hand utilize all available cores to fully exploit parallelism in a load balanced manner
- Moving Compute starts outperforming Atomic at 64 cores

Moving Compute Core Scaling Trend



- MC2D model scales better than both Spin and Atomic as upper limits of core counts are approached in Tile-Gx72
  - Larger network increases data access latencies, hence cache line ping-pongs are more expensive in Spin as compared to Atomic and Moving Compute models
- MC2D model (1) generalizes atomics, and (2) scales to higher core counts and delivers superior performance at 64 cores!