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

Halit Dogan*, **Masab Ahmad***, **José A. Joao**†, **Omer Khan***
†Arm Research, Austin, TX, USA
*University of Connecticut, Storrs, CT, USA

Abstract—**The shared memory cache coherence paradigm is prevalent in modern multicores. However, as the number of cores increases, synchronization between threads limits performance scaling. Hardware-based core-to-core explicit messaging has been incorporated as an auxiliary communication capability to the shared memory cache coherence paradigm in the Tilera TILE-Gx72 multicore. We propose to utilize the auxiliary explicit messaging capability to build a *moving computation to data model* that accelerates synchronization using fine-grain serialization of critical code regions at dedicated cores. The proposed communication model exploits data locality and improves performance over both spin-lock and atomic instruction based synchronization methods for a set of parallelized graph analytic benchmarks executing on real world graphs. Experimental results show an average** 34% **better performance over spin-locks, and** 15% **over atomic instructions at 64 cores setup on TILE-Gx72.**

I. Introduction

Large-scale multicores are now pervasive, and the shared memory paradigm with hardware cache coherence is still the dominant inter-thread communication model. Even though distributed hardware cache coherence protocols are efficient for seamless data sharing between threads, the synchronization on shared data through coherence still remains a significant challenge for large multicores. Expensive overhead of coherency traffic leads to costly data sharing with increasing number of threads and cores that participate in synchronization.

Architectural improvements and novel communication models have been proposed to overcome the synchronization bottleneck. One of the most effective among many proposed solutions is to incorporate in-hardware explicit messaging into a shared memory machine [1], [2], [3], [4], [5], [6], [7], [8]. Commercial Tilera [9] architecture incorporates hardware-based explicit messaging support as an auxiliary core–to–core communication mechanism to the directory-based cache coherence protocol. It uses User Dynamic Networks (UDN) to enable movement of data from one core's register file to another core's register file without interfering with the cache coherence traffic and protocol. TSHMEM [10] explores barrier synchronization in the Tile-Gx36 and TILEPro64 machines using the UDNs. However, it does not explore synchronization model tradeoffs that involve fine–grain and coarse–grain synchronization for real application domains, such as graph analytics. Hence, the messaging network is not fully explored in terms of mitigating the synchronization bottleneck.

The moving compute to data model (MC2D) is a promising approach to accelerate synchronization in a shared memory machine that incorporates hardware explicit messages [8]. Building on previous simulation-based efforts, this paper implements the MC2D model to accelerate both fine and coarse-grain synchronization of shared data in the Tilera's 72–core TILE-Gx72 machine [11]. The MC2D model is compared to the traditional shared memory based lock and barrier implementations for synchronization. The spin-lock and atomic models are implemented using Tilera's support for hardware cache coherence based atomic instructions. Refactoring applications to use fine–grain critical sections enables additional concurrency compared to the coarse-grain synchronization. However, it adds extra lock acquisition overhead even when no other thread accesses the same shared data. Furthermore, it gets worse in the presence of contention due to instruction retries and cache line ping–pong between cores. Tilera also offers atomic instructions to mitigate overheads of the spin–locks when they are applicable in an algorithm's implementation. However, as atomic instructions do not completely eliminate cache line ping–pong, they also get costly as the core count increases. Additionally, only few simple operations are implemented with atomic semantics in an architecture, which limits their potential to implement arbitrary critical sections.

The MC2D model addresses the challenges of the traditional shared memory synchronization, and implements critical sections efficiently in a generalized way. It eliminates the need for atomic instructions, or locks to protect critical code sections. Rather, it offloads critical sections to a dedicated core (*service core*) via low latency explicit messages. The service core performs the requested critical section work atomically without interruption, thereby efficiently serializing shared work and preventing cache line ping–pong. Furthermore, serialization at a service core is alleviated by assigning multiple service cores. For the best performance, the right number of service cores is determined using a heuristic that trades off concurrency in algorithm's work (*worker cores*) with concurrency in the shared work (*service cores*).

The MC2D model is realized in the TILE-Gx72 machine for the graph analytics application domain since graph algorithms vary significantly in terms of their synchronization requirements. To the best of our knowledge, this is the first work that implements the MC2D model in a real multicore machine for graph problems. A characterization using various real world input graphs is conducted to identify key characteristics, such as contention on shared data and load balancing of threads to determine how MC2D compares to the traditional shared memory synchronization models. A novel shared work driven heuristic is proposed to determine the right number of worker and service cores for the MC2D model. Moreover, a core count scaling study is performed to highlight the superiority of the MC2D model as the impact of on-chip network latency dominates performance scaling in the traditional spin-lock and atomic synchronization models. Evaluation using six graph

2576-6996/18/$31.00 ©2018 IEEE
DOI 10.1109/ICCD.2018.00080

benchmarks and four real world brain, transportation, and social network graphs shows a performance improvement of 34% over spin-lock, and 15% over the atomic instruction based thread synchronization model.

II. Synchronization Models in Tilera TILE-Gx72

Tilera TILE-Gx72 processor utilizes a tiled multicore architecture approach. It contains 72 tiles, and the tiles are connected using an intelligent 2-D mesh network, called iMesh Interconnect. Each tile consists of a 64-bit VLIW core, 32 KB private level-1 data and instruction caches, and a 256 KB shared level-2 (L2) cache. Each VLIW core implements three separate pipelines. The first pipeline performs all arithmetic and logical, multiply and fused-multiply, and bit manipulation instructions. The second pipeline also executes all arithmetic and logical operations, as well as special purpose register reads and writes, and conditional branch instructions. The third pipeline is utilized for memory instructions, including the atomic memory operations. A directory is integrated into the L2 cache slices to support a directory–based cache coherence protocol. Tilera architecture also offers various configurations for data placement and caching schemes. By default, a cache line is homed at an L2 cache using a hardware hashing scheme, and also replicated in the L2 slice of the requesting core. Experiments with and without replicating cache lines in the local L2 slice of the requesting core vary performance by an average of 1% for the investigated workloads. Hence, the default L2 homing scheme is utilized in this paper. In addition, network routers are included in each tile to communicate with other tiles, I/Os and the on-chip memory controllers. A special network called User Dynamic Network (UDN) is used to enable tile–to–tile explicit messaging. Each tile contains four UDN queues for explicit messaging. These queues are implemented as small FIFO queues. Each queue is mapped to a special purpose register, which is used to send and receive data between execution units without any involvement of the cache coherence protocol and traffic. For example:

move udn0, r0 is a send operation in which data in *r*0 is moved to the special purpose register, *udn*0. Then, it is injected into the network, where it traverses to the destination tile.

move r0, udn0 is a receive operation in which the sent data is received and placed in the queue 0, and since the queues are mapped to special UDN registers, the data is read from the corresponding register (*udn*0 in this case) when it arrives in the specified queue. If the message does not make it to the queue when the "*move*" instruction is executed, this operation stalls the corresponding pipeline. In addition to "*move*" instruction, any ALU operation can send/receive data using the UDN network, registers and the queues.

The TILE-Gx72 provides Tilera Multicore Components (TMC) library [12] to initialize and make use of the UDNs. Hence, the low level instructions are not used for explicit communication. For this paper, the library calls provided by TMC library for tile–to–tile messaging are utilized. To be able to make use of the UDN networks, the threads are pinned to the cores based on their thread IDs in an ascending order. In the TILE-Gx72, the threads are spatially distributed among available cores.

A. Shared Memory Based Synchronization

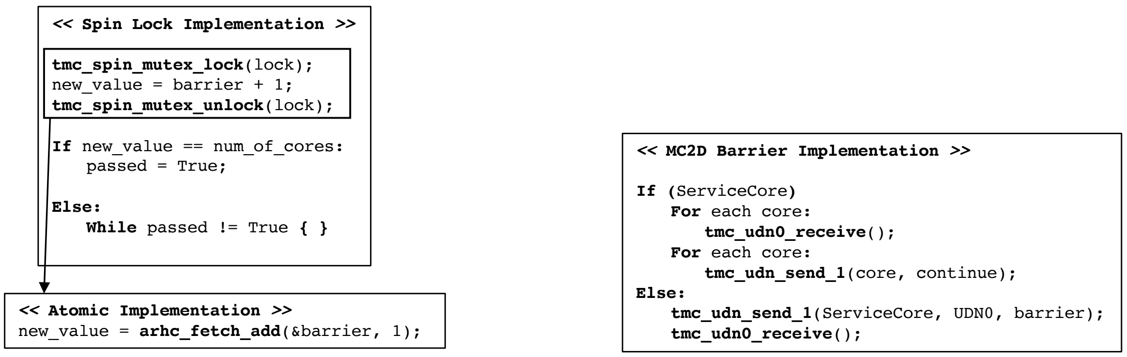
Tilera offers various atomic memory instructions for efficient thread synchronization on shared data. Tilera documentation does not provide implementation details of the atomic operations. Hence, it is not known if an atomic operation is implemented by locking a cache line in the private cache (*near atomic*), or as remote atomic operation at the home L2 cache for the cache line (*far atomic*). In either case, as an atomic memory operation utilizes the same pipeline as the load and store instructions, the other two VLIW pipelines can continue execution in parallel. Some of the atomic operations are as follows: *cmpexch*, *fetchadd*, *fetchaddgez*, *exch*, to name a few. Compare–and–exchange (*cmpexch*) is utilized to build the widely applicable spin-lock based synchronization model that protects an arbitrary critical code section. When applicable, the atomic instructions can be directly utilized to implement synchronization, which helps mitigate spin-lock's instruction retries and cache line ping–pong. However, as the core count increases, the atomic instructions also suffer from the cost of expensive data sharing between threads, as well as the on-chip network. Moreover, they are limited to specific operations and data sizes, thus limiting their applicability to a wide range of critical section implementations. In this paper, the atomic model implements lock–free data structures if the algorithm is suitable to employ the available set of instructions. Fortunately, all six evaluated graph benchmarks are supported with the atomic model in addition to the spin-lock model.

Tilera Multicore Components (TMC) library provides two types of spin-based synchronization. The first one is based on the kernel scheduler, while the other version deploys synchronization by utilizing the atomic instructions. As atomic instruction based synchronization does not interact with kernel scheduler, it is generally more efficient and thus utilized in this paper. Instead of yielding a core when the *mutex* is not available, spin–based primitive continues to perform tests until the mutex variable is available. As an optimization to reduce expensive retries, it utilizes an exponential backoff mechanism in which the thread stops trying to acquire the lock variable, and waits for some time to try again. If it fails, it increases the backoff cycles exponentially. The library also offers queue based locking. In this case, the threads are put into a waiting list until the mutex is available. When the mutex is available, the next thread is notified to acquire the lock. Experiments are conducted with both lock mechanisms, and it is observed that for the evaluated benchmarks there is no significant performance difference between the two implementations. Hence, the spin-lock with exponential backoff is adopted in this paper.

Similar to the atomic instruction model, the spin-based synchronization is also effective when the shared data is not contended. When there is no contention, concurrency is not limited by serialization on the shared data. On the other hand, with contended shared data, the serialization overheads increase due to the instruction retries and the lock variable ping-ponging. This leads to degraded performance scaling for the spin based synchronization.

B. Explicit Messaging Based Communication

The TILE-Gx72's explicit messaging using the UDN supports both blocking and non–blocking communication.



```
<< Spin Lock Implementation >>
tmc_spin_mutex_lock(lock);
new_value = barrier + 1;
tmc_spin_mutex_unlock(lock);

If new_value == num_of_cores:
    passed = True;

Else:
    While passed != True { }

<< Atomic Implementation >>
new_value = arhc_fetch_add(&barrier, 1);
```

(a) Shared memory barrier.

```
<< MC2D Barrier Implementation >>

If (ServiceCore)
    For each core:
        tmc_udn0_receive();
    For each core:
        tmc_udn_send_1(core, continue);
Else:
    tmc_udn_send_1(ServiceCore, UDN0, barrier);
    tmc_udn0_receive();
```

(b) MC2D barrier.

Fig. 1: Barrier implementation using spin–lock, atomic fetch–and–add, and MC2D models in TILE-Gx72

Blocking communication is realized using a send operation followed by a receive operation at the sender core. The receiver core pairs the sender's operations with receive and send, respectively. The pipeline of the sender core is blocked until it receives an explicit reply from the destination core. Blocking communication is useful to enforce strong data consistency, or when the sender requires an explicit reply back from the destination before proceeding. An implementation must take into account the limited UDN queue capacity (118 words per tile), otherwise a blocking communication may result in an application level deadlock [13]. This paper utilizes blocking communication to implement barrier synchronization between cores. At most 71 cores send a one-word message to a single core in the barrier implementation, which has sufficient capacity to hold 71 words of data in its UDN queue.

Non-blocking communication is realized using a send operation at the sender core paired with a receive operation at the destination. The core that executes the send operation is allowed to continue execution without blocking the pipeline. This allows the sender core to utilize all three VLIW pipelines after sending a message, and thus overlap non-blocking communication with other useful work. This paper utilizes non-blocking communication to enable efficient execution of concurrent critical section tasks. The usage of blocking and non-blocking communication is discussed in detail in the following section.

C. Moving Compute to Data (MC2D) Model

In the MC2D model, instead of bringing data where the computation resides, the shared data is pinned at certain cores and the computation is moved to them using explicit messages. The low latency of in-hardware explicit messaging in the TILE-Gx72 allows the MC2D model to exploit locality on shared data and accelerate synchronization. The need for atomic memory instructions or spin-lock primitives is replaced with serialization of critical section execution at a dedicated core, termed as *service core*. When realizing the MC2D model, the disjoint shared data operating on critical sections is distributed among a set of *service cores* to further exploit concurrency. Note that the shared data is pinned to a service core such that its updates are performed atomically at that core without interruption. The remaining cores, termed as *worker cores* perform the actual algorithmic work, and send critical section invocation requests to the service cores using the UDNs. Upon receiving a UDN message, the service core executes the requested critical code section. At the end of execution, a service core either waits for another UDN request, or sends an explicit reply message to the worker core to update it with the completion of its request.

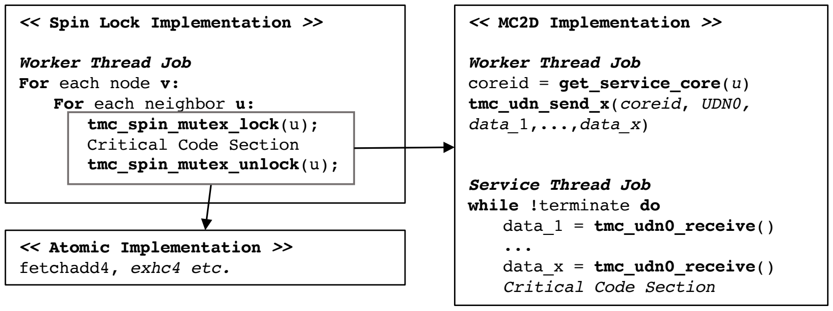
Tilera's TMC library exposes the UDN network to the programmer via an API. Following subsections discuss the implementation of coarse and fine–grain synchronization using the MC2D model.

1) Coarse–grain Synchronization: Figure 1 illustrates the implementation of barrier synchronization using various capabilities of TILE-Gx72. As seen in the pseudocode in Figure 1a, the spin–lock based barrier is implemented by locking and incrementing the barrier variable, and spinning until all the cores perform their shared data updates. When many/all cores participate in synchronization, both lock and barrier variables are contended, and ping-pong between cores results in expensive thread synchronization. By removing the lock and performing the increment with an atomic *fetchadd* instruction, the barrier performance can be improved by mitigating cache line ping–ponging. However, as the core count increases, the contention on the shared barrier variable hurts performance even when atomic *fetchadd* instruction is utilized.

The MC2D barrier removes the shared barrier variable as shown in Figure 1b. It makes use of the blocking communication capability of the UDN network. When the barrier synchronization is needed, one of the cores among worker cores is assigned as the service core. All other cores send a barrier message to the specified service core, then execute a receive operation to wait for a reply from the *service core*. When the service core receives messages from all the participating cores, it broadcasts a proceed message to the participants.[1] This eliminates the spinning and the barrier variable sharing, leading to more efficient synchronization. One can also utilize a separate core as service core to manage the barriers. However, this paper employs one of the worker cores as the service core to handle the barrier synchronization.

Three evaluated graph algorithms, PAGERANK, CONNECTED COMPONENTS and COMMUNITY DETECTION involve only bar-

[1]If the core handling the barrier messages participates in barrier synchronization, and receives other barrier messages before reaching the barrier, then such messages wait in the UDN queue until the core itself reaches the barrier.



```
<< Spin Lock Implementation >>

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

```
<< Atomic Implementation >>
fetchadd4, exhc4 etc.
```

```
<< MC2D Implementation >>

Worker Thread Job
coreid = get_service_core(u)
tmc_udn_send_x(coreid, UDN0,
data_1,...,data_x)


Service Thread Job
while !terminate do
    data_1 = tmc_udn0_receive()
    ...
    data_x = tmc_udn0_receive()
    Critical Code Section
```

Fig. 2: Implementation of fine–grain synchronization using spin-lock, atomic, and MC2D models in TILE-Gx72

rier synchronization in their implementations. These algorithms consist of multiple phases of graph traversal, and at the end of each phase, threads are synchronized to make sure the updates to shared data are visible to all the threads.

2) Fine–grain Synchronization: Figure 2 shows synchronization of shared data for a primitive graph benchmark, where the nodes are statically divided among threads, and each thread visits the neighbors of its nodes and performs an atomic update for each neighbor. The pseudocode on the upper-left shows the spin–lock implementation, where the atomic update is done by acquiring the corresponding lock for the node. If an atomic instruction for the critical section is available, the locks are removed and a single atomic instruction performs the critical section (bottom left). For MC2D, the lock is removed and the critical section work is moved to dedicated service cores by using explicit messaging functions provided by TMC library (pseudo code on the right in Figure 2). The worker cores send critical section requests along with the necessary data indices to the corresponding service cores. The desired service core is obtained using a lookup function, *get_service_core*(), as seen in the pseudocode. This function is used to distribute the disjoint shared data among the service cores. The number of data word(s) that needs to be sent is algorithm dependent. If multiple words are sent, they are placed into the destination queue in the order they are sent. Programmer must make sure that send and receive ordering is maintained to keep correct functionality, as seen in the pseudocode. A service core then receives the required number of words, and executes the critical section. *Note that the work efficiency of the original algorithm remains unchanged when porting to the MC2D model. Only the critical section work is moved to a separate core. If there is a test to filter out redundant lock acquisitions, the test stays as is in all the versions.*

The advantage of MC2D over traditional shared memory synchronization comes from improved data locality. It eliminates both retries and ping–pong of shared data by getting rid of the locks and pinning the shared data at service cores. Due to the non–blocking nature of the explicit *send* instruction, it also helps overlapping the communication stalls with other computations, allowing each worker core to continue execution after sending a request to the corresponding service core. In a way the worker and service core tasks are pipelined using non–blocking communication. While a worker core prepares to send another message, the service core processes the previous message(s). Hence, efficient task level parallelism is enabled. The MC2D model can also be implemented using synchronous messages in which a worker core waits until its request is processed and an explicit reply message is received from the service core. This approach may be useful when there is strict consistency requirement in the algorithm's synchronization. However, it prevents hiding the communication latency, and may lead to performance degradation when the benefits of data locality cannot overcome the communication overheads. The evaluated graph algorithms do not require strong consistency, hence the MC2D model utilizes non–blocking communication in this paper.

Three graph algorithms with fine-grain synchronization are evaluated, i.e., SINGLE SOURCE SHORTEST PATH, TRIANGLE COUNTING and BREADTH FIRST SEARCH. Their parallel implementations use similar approach to the primitive algorithm described in Figure 2. However, they are described below to provide relevant details.

Triangle Counting (TC) divides nodes among threads, and each thread keeps track of the number of connections per node. As nodes may share the same neighbor, the counters that track the connections are protected with spin–locks. TC does not include any test to prevent redundant locking, which infer a higher amount of contention on shared data. As the critical section of TC is just an increment operation, it is also implemented using the fetch–and–add atomic instruction directly. Finally, for the MC2D model implementation, each worker core sends the neighbor ID to the corresponding service core, and moves forward with subsequent work. Consequently, the service core performs the requested shared data update.

Breadth First Search (BFS) visits the nodes iteratively by opening new pareto fronts in each iteration over the graph. Each thread goes through its part of the graph, and the implementation contains a test to prevent redundant synchronizations. It tries to guarantee that each node is visited only once in the whole program execution. The visiting part is protected with fine–grain locks to ensure that no other thread visits the same node. BFS is also implemented using an atomic compare–and–swap instruction to eliminate the lock overhead. For the MC2D model, each worker core sends the neighbor ID, and the corresponding service core asynchronously operates on it upon receiving the request.

Single Source Shortest Path (SSSP) divides the nodes among threads, and each node updates its distance to its connected edge. The algorithm iterates over the graph until the distance array converges. Each distance array relaxation is protected with a fine–grain lock. A test is applied right before lock acquisition to prevent unnecessary locking if the node is already converged. The critical section of SSSP requires a fetch, decrement, compare and update, which is not available as a single atomic instruction in Tilera. Hence, the update part of the critical section is implemented using a single atomic swap instruction to emulate the atomic model, even though this change alters the algorithm. On the other hand, the MC2D model can be used to implement any arbitrary atomic operation. Therefore, the relaxation work (critical section) is shipped to the service core, where update to the requested shared data is performed. Each worker core sends node, neighbor ID, and the edge weight to the corresponding service core. The MC2D version of SSSP also performs the test before asynchronously sending each critical section request.

3) Determining Right Service Core Count: The MC2D model exploits concurrency in shared work using more than one service core. Unfortunately, this takes away concurrency from the worker cores for all other algorithm work. If the distribution of the worker and service cores is not done properly, the parallel implementation can suffer performance loss due to load imbalance between threads. Hence, it is important to find the right number of cores for a near-optimal spatial allocation of *service* and *worker cores*. This cannot be done statically since the distribution of shared work is highly dependent on the graph algorithm, as well as the input graph.

We propose to deploy a profiling based heuristic that utilizes the percentage of shared work to determine the right ratio of worker and service cores. In this approach, the shared memory version of the workload is profiled to obtain the percentage time spent in the critical section with respect to the total completion time. As the MC2D model ships the work in the critical section to dedicated cores, the time spent in the critical section is a good indicator to determine the appropriate number of service cores. It is anticipated to show linear correlation with the service core count.

III. Methodology

The TILE-Gx72 multicore processor executes at $1GHz$ and is equipped with $16GB$ of DDR3 main memory. It runs a linux version that is modified for the Tilera architecture. A modified version of GCC4.4.7 that supports Tilera specific features is utilized for the compilation of the benchmarks. As discussed in Section II, three thread synchronization models (Spin, Atomic, and MC2D) are evaluated in this paper.

A. Performance Metrics

Up to 64 cores in the system are utilized for performance evaluation. While running experiments, no other program interferes with the active application. Following are the evaluation metrics used in the paper.

- **Completion Time:** Completion time is measured by running each benchmark to completion, and only the parallel region is measured in each application. Memory allocations, initialization of data, and thread spawning overheads are not taken into account. Every run is repeated ten times and the average number is reported.
- **Load Imbalance:** Load imbalance is determined by calculating the variability in the instruction counts of the cores. The number of instructions for each core is determined using the hardware event counters in Tilera. The variability across instruction counts of the cores is calculated using the following formula:

$$Variability = \frac{Max(Instructions) - Min(Instructions)}{Max(Instructions)}$$

- **Shared Work:** The percentage time spent in the critical section is determined by measuring the time between lock–acquire and lock–release in the Spin model. A specific counter per thread keeps track of this time, and determines the total time spent in the shared work for each thread. Then, the average shared work across all cores is calculated using the per thread data. Finally, the amount of work done in the critical section compared to the total completion time is determined as a percentage number, and reported as the *shared work*. This metric is used to determine the number of service cores for the MC2D model.

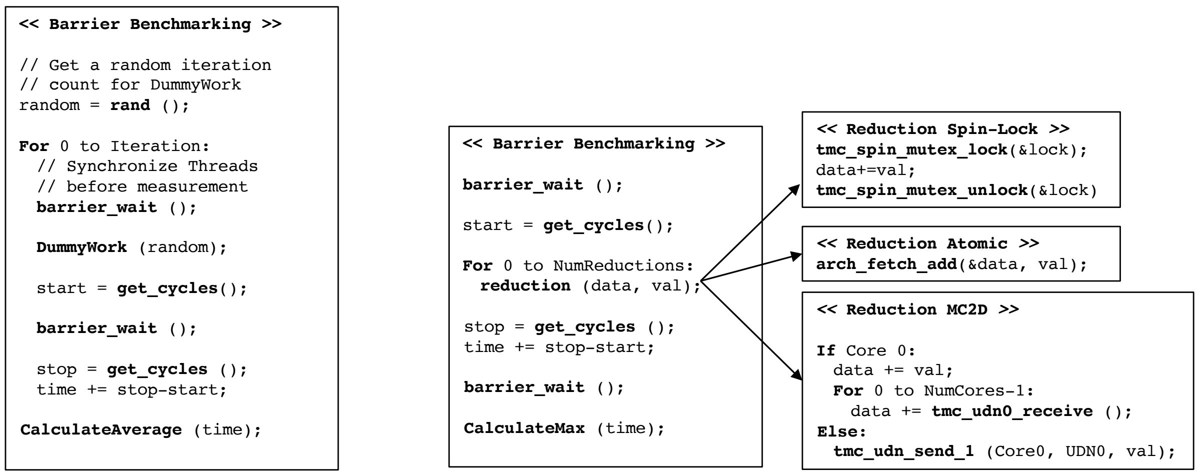
TABLE I: Input graphs and their respective statistics.

Inputs	Nodes	Edges	Degree
Mouse Brain [14]	562	0.57M	1027
CA Road Network [15]	1.9M	5.5M	2.8
Facebook [16]	2.9M	41.9M	14.3
LiveJournal [16]	4.8M	85.7M	17.6

B. Benchmarks and Inputs

Six graph benchmarks from the CRONO [17] suite are adopted for this work, namely SSSP, TC, BFS, PAGERANK, CC, and COMM. These benchmarks are ported to the TILE-Gx72 using the Spin, Atomic and MC2D synchronization models. For all models, pthreads library is used to spawn threads, and each thread is pinned to a physical core based on the thread ID. For evaluation, four real world graphs are chosen to explore input diversity, as summarized in Table I.

In order to build intuition for the performance advantages of the MC2D model over the Spin and Atomic models, two microbenchmarks are evaluated. In the first one, barrier synchronization is evaluated under load balanced versus imbalanced execution of threads. The pseudo code for the barrier microbenchmark is shown in Figure 3a. The barrier is executed 10000 times, and average barrier time is calculated at the end. Before each barrier measurement, all the threads are synchronized with another barrier. Two sets of measurements are conducted. In the first one, the threads arrive at the barrier at the same time, hence it is contended. The *DummyWork()* function seen in the pseudo code is removed for this experiment. For the second experiment, each thread performs some dummy work before entering the barrier synchronization. In the dummy work, each thread executes a random number of instructions to observe different arrival times. All three barrier implementations discussed in Section II-C1 are evaluated. The number of threads is varied from 2 to 64. The MC2D model utilizes Core 0 as the service core. Therefore, the barrier variables for the Spin and Atomic models are also mapped to Core 0's L2 slice to minimize network variability across synchronization models.



```
<< 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);
```

(a) Barrier benchmarking.

```
<< 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);
```

(b) Reduction benchmarking.

Fig. 3: Pseudo code for two microbenchmarks.

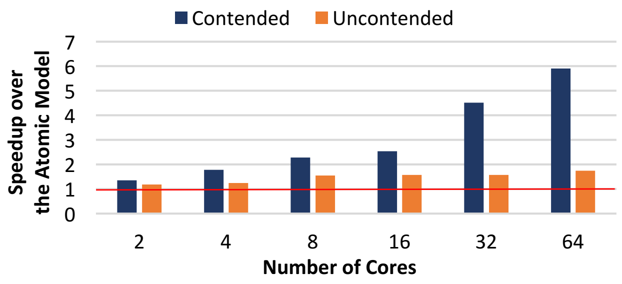


Fig. 4: Speedup of barrier microbenchmark for the MC2D model over the Atomic model at different core counts.

The second evaluated microbenchmark is a reduction in which multiple threads perform a summation function over a shared data. The pseudo code is shown in Figure 3b. All the participating threads are first synchronized with a barrier, and then start performing the reduction routine. This leads to contended synchronization between threads. Moreover, the number of reductions per thread are varied from 1 to 1000, at power of 10 increments. The multiple per-thread reductions further increase contention on the shared data. The reduction is implemented with a global lock for Spin, and a fetch–and–add atomic instruction for the Atomic model. In the MC2D model, all cores send their local data to the service core 0, which serially performs the summation. Similar to the barrier microbenchmark, the shared data is mapped to Core 0's L2 slice for the Spin and Atomic models. The number of threads are varied from 2 to 64 for evaluation.

IV. Microbenchmark Evaluation

Figure 4 shows the speedup of the barrier microbenchmark for the MC2D model over the Atomic model at different core counts. These experiments are also conducted for the Spin model, however the Spin barrier performs consistently worse than the Atomic barrier due to its lock acquisition overheads. Hence, the results for the Spin model are not shown in the figure. When the barrier is contended, i.e., threads reach the synchronization point at similar times, the performance of the atomic instruction based barrier degrades drastically as the number of participating cores increases. At 64 cores, the MC2D barrier performs $\sim 6\times$ faster than the Atomic model implementation. Although the fetch–and–add atomic instruction to update the barrier variable improves performance, the Atomic model still relies on spinning until all cores perform their shared data updates. This leads to ping-ponging of the barrier variable between participating cores, which gets worse as the distance and contention on the network-on-chip increases. On the other hand, the MC2D model based barrier utilizes a local variable at Core 0, and eliminates the bouncing between cores. Hence, it provides superior performance under contention.

The barrier performance is also measured when it is not contended. In this case, each thread performs different amount of work before arriving at its barrier update. Hence, the Atomic model is able to hide much of the cache line ping-pong latencies, and improves performance significantly compared to the contended barrier. However, the MC2D model barrier still outperforms the Atomic model barrier for the un-contended case. Under the Atomic model, the participating threads utilize the backoff mechanism in their waiting loop for the atomic updates to the barrier variable, and thus incur some cache line ping-pongs. On the contrary, the MC2D model does not require a backoff mechanism. The participating cores block their execution while the service core 0 explicitly manages the barrier variable updates, and finally sends a reply to inform the worker cores to continue their execution.

Figure 5 shows the speedup of the MC2D model over the Atomic model for the reduction microbenchmark. Similar to the barrier experiment, the Spin results are not shown since they are considerably worse than both Atomic and MC2D models. As seen in the figure, when a single reduction is performed, the Atomic model consistently provides better performance than the MC2D model at all core counts. The atomic fetch-and-add instruction seems to completely eschew cache line ping–pong similar to the MC2D model. However, each core issues a single

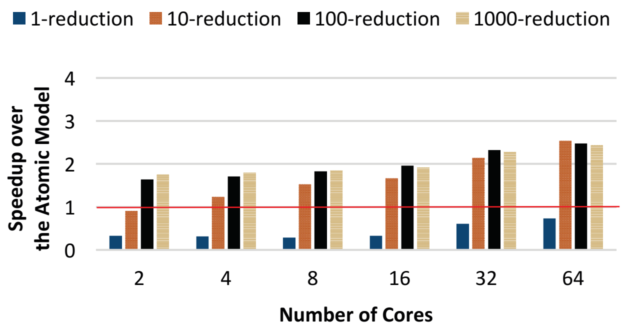


Fig. 5: Speedup of MC2D over the Atomic model for various number of per-thread reductions at different core counts.

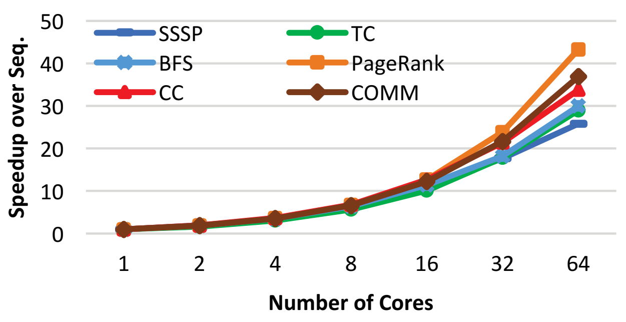


Fig. 6: Average per-benchmark performance scaling results using the Spin model.

atomic instruction, and exploits concurrency to achieve good performance. The MC2D model, on the other hand incurs additional instructions to send the reduction message from each worker core to the service core 0, which receives each message, and serially updates the shared variable. However, as the number of per-thread reductions is increased, the MC2D model is able to overlap the explicit messaging latency by using the non-blocking send messages. Consequently, it yields higher performance than the Atomic model, which waits for the completion of each atomic update before executing the next per-thread atomic update. It is illustrated in the figure that the MC2D model consistently improves over the Atomic model as both number of reductions and core counts are increased. The next section discusses the evaluation of graph benchmarks which provide ample opportunities for the MC2D model to pipeline threads and hide the latency of synchronization.

V. Evaluation of Graph Benchmarks

A. *Performance Scaling of Graph Benchmarks*

A performance scaling study is conducted to illustrate that the baseline spin–lock based shared memory implementations scale to 64 cores on the TILE-Gx72 platform. Each benchmark is executed by varying the core count from 1 to 64. The average speedup for each core count is plotted relative to the sequential execution of the benchmark. The sequential implementation spawns a single thread of execution that exploits all on-chip shared cache and memory controller resources. Figure 6 shows the performance scaling results for the six evaluated benchmarks. As seen, all benchmarks improve performance up to 64 cores. The benchmarks with coarse–grain communication (such as PageRank) scale better than the ones with fine–grain communication (such as TC). This is expected as contended shared data in several graph benchmarks lead to the synchronization bottleneck. Overall, the Spin model achieves 27× to 45× performance improvement at 64 cores over sequential.

B. *Performance of MC2D and Atomic over Spin*

Figure 7 shows the normalized completion time results of the Spin, Atomic and MC2D models at 64 cores setup. Atomic and MC2D both follow the same trends over Spin. There is almost no performance difference between all three synchronization models for the benchmarks with coarse–grain synchronization (PAGERANK, CC, and COM) . Even though both MC2D and Atomic based barriers are more efficient than Spin, these benchmarks do not show any significant performance change since each core has a considerable amount of work between barriers. The only exception is CC using the relatively small mouse-brain graph, which does not incur much computations between barrier synchronization of threads. In this case, an efficient barrier with the MC2D model provides better performance.

Unlike the workloads with coarse–grain synchronization, the ones with fine–grain synchronization (SSSP, TC, and BFS) show some variability across different benchmark–input combinations. Here, contention is an important metric to indicate the cases where performance can be improved using better synchronization primitives. As seen in Figure 7, while MC2D reduces completion time for TC and SSSP, the completion time does not significantly change for BFS. This is due to the fact that SSSP and TC involve more contended locks than BFS. Figure 8 illustrates the contention of each workload with respect to their performance over Spin. *Contention* is the average number of lock–acquisitions per node in a graph, determined using per-node counters in the critical section. As observed, when the contention increases, the performance obtained from MC2D and Atomic also escalates. Since BFS algorithm guarantees that each lock variable is acquired only once in the whole program execution, the locks are not contended, and the shared work done by each thread is very small compared to the private work. As a result, there is not much to improve with a more efficient synchronization model.

On the other hand, the Spin implementation of TC requires locking of each edge without any condition, as explained in Section II-C2. Therefore, contention is higher as illustrated in Figure 8. Consequently, the MC2D model significantly improves performance for all input graphs. This performance achievement mainly comes from removing the lock acquisition overheads by pinning shared data at dedicated service cores, and using low latency non–blocking explicit messages. TC is an ideal showcase of MC2D as the shared data is neither read nor written by any other core. It totally eliminates sharing of the shared data with any other thread, and basically makes it private data to the service cores. As a result, it provides an average of 76% performance benefit over Spin.

SSSP is a benchmark where the number of lock acquisitions per node (contention) is greater than BFS, but less than TC. It has a test before getting into critical section to make sure no

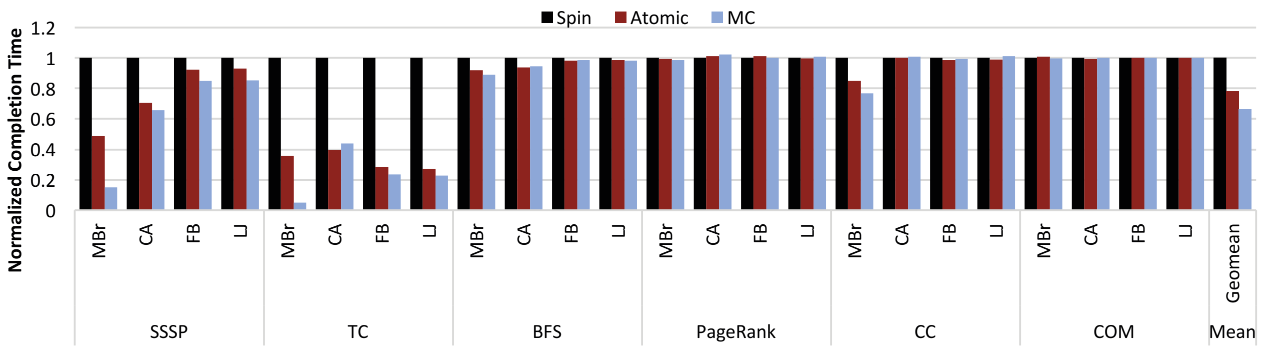


Fig. 7: Completion time results under Spin, Atomic and MC2D models. All results are normalized to Spin.

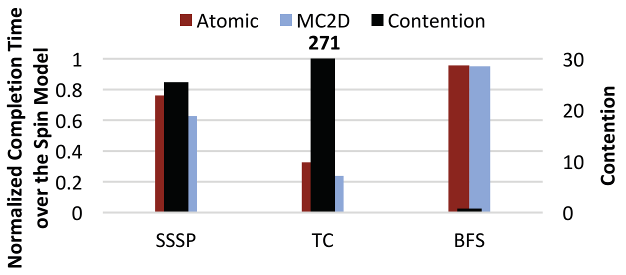


Fig. 8: Contention vs. performance of MC2D and Atomic models.

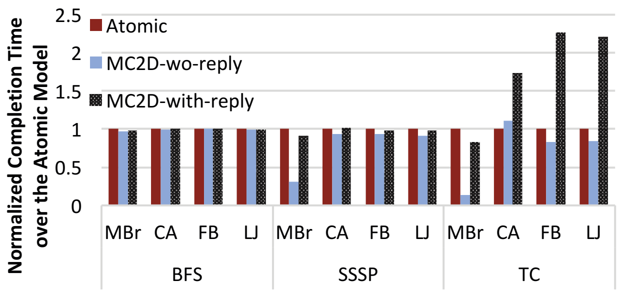


Fig. 9: Normalized performance of MC2D models with and without reply messages against the Atomic model.

redundant lock acquisition is performed. However, due to its iterative nature, each lock is acquired multiple times in the program execution. Therefore, similar to TC, removing these locks with MC2D, and shipping critical sections to service cores with non–blocking messages help improve performance. On average, MC2D yields 34% efficient program execution compared to the Spin model.

C. *Performance of MC2D over Atomic*

In this section, the MC2D model is evaluated against Atomic, which is a more efficient implementation of synchronization as compared to Spin. Since benchmarks with coarse–grain synchronization do not show much performance differences, they are not discussed further. Benchmarks with fine–grain synchronization are implemented using the atomic instructions available in the TILE-Gx72, as explained in Section II-C2. Figure 7 shows that on average the MC2D model accomplishes 15% better performance as compared to the Atomic model.

As discussed earlier, the contention in BFS is negligibly small, hence MC2D does not offer any additional performance. Almost all of the performance benefits stem from TC and SSSP. Both algorithms under MC2D show similar behavior against Atomic, except TC executing with the California road network graph, where Atomic slightly outperforms MC2D. The main benefits come from overlapping the communication stalls with other computation. The MC2D model utilizes asynchronous messaging for the critical section requests. Each worker core sends its request to the corresponding service core and continues to do other useful work, including subsequent requests for the critical section executions. This implicitly pipelines the critical section executions and reduces the overheads of synchronization. To verify the performance advantage of the non–blocking communication in the MC2D model, a study is performed where each worker core waits for an explicit reply message from the corresponding service core to ensure the critical section work completed before proceeding. Figure 9 illustrates the performance comparison for the default MC2D model without reply, and the MC2D model with reply. When the MC2D model waits for the reply, the performance gets worse than the Atomic model when contention is high in the benchmark. This illustrates that the fine–grain synchronization stalls benefit significantly when they are overlapped with other useful work in the worker cores.

For both TC and SSSP, MC2D yields higher speedup over Atomic with the mouse brain graph as compared to other graphs. The mouse brain is a dense graph in which almost all nodes are connected to each other, thus more sharing occurs between cores, and MC2D exploits performance since it eliminates sharing by pinning the shared data at service cores.

D. *MC2D Model and Cache Coherence*

Even though the MC2D model accelerates synchronization on the shared data, it relies on the hardware cache coherence protocol for efficient movement of cache lines between cores. This is specifically important for parallel implementation of work efficient algorithms. For example, SSSP contains a test to ensure that redundant critical section executions are not performed. To implement this test, a worker core reads some shared data that is pinned on the service core. This data sharing adds coherence traffic, however it prevents unnecessary work.

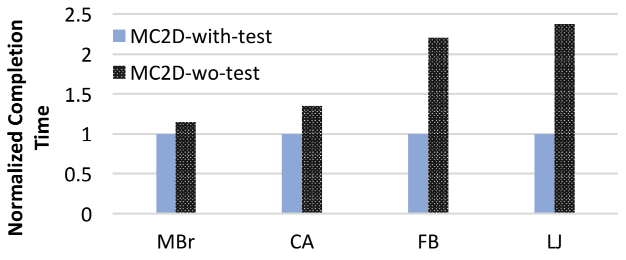


Fig. 10: Performance of SSSP under MC2D model with and without test before sending critical section invocations.

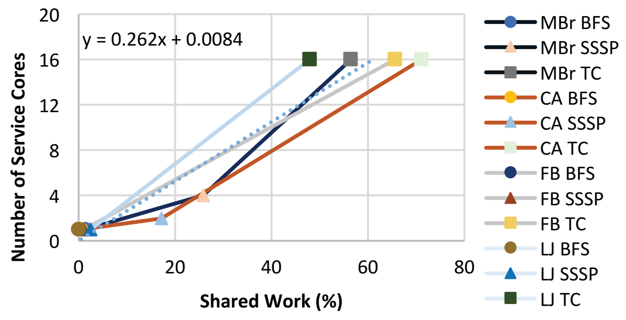


Fig. 11: Correlation of service core count with shared work.

Figure 10 shows the implementation of SSSP with and without the test under the MC2D model. As observed, the performance of MC2D without the test decreases significantly since it incurs overheads of redundant critical section invocations. The savings from eliminating coherence traffic cannot compensate for the overheads of redundant critical section invocations, and hence the performance of MC2D without the test decreases significantly as compared to the MC2D with test. Moreover, as the size of the input graph increases, the performance penalty of not using the test also goes up. This is observed for LiveJournal and Facebook graphs that filter significant critical section requests when the test is utilized.

E. Heuristic to Determine Service Core Count

As discussed in Section II-C3, tuning the number of worker and service cores plays a significant role for the MC2D model to deliver near-optimal performance. As MC2D ships the critical section executions to the dedicated service cores, it is expected that the time spent in the critical sections shows correlation with the optimal number of service cores. Figure 11 shows strong correlation between the profiled shared work (see Section III-A) and the ideal service core count for each benchmark–input combination. As seen, BFS has a very small amount of shared work (less than 1%) for all four input graphs, which results in only one service core allocation. On the other hand, TC involves notable shared work (grater than 50%), which results in a higher number of service cores (16 cores). SSSP's shared work varies depending on the input graph as the convergence of the algorithm depends on the graph itself. Therefore, it requires 1–4 service cores. The correlation between shared work and service core count is captured with a simple linear model as demonstrated in the figure. It serves as a heuristic to determine the service core count for a given shared work. The heuristic is employed to find the number of service cores for all benchmark-input combinations, and the result is compared to a service core count determined using an exhaustive search. The performance difference between using the heuristic and the ideal number of service cores is observed to be within 3%.

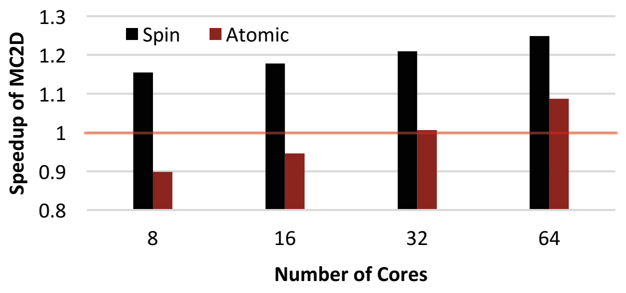


Fig. 12: Average performance scaling results of MC2D compared to Spin and Atomic models.

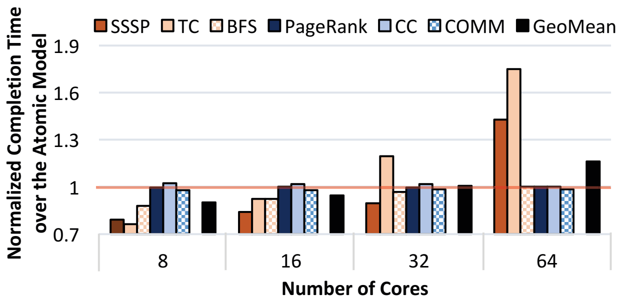


Fig. 13: Average per-benchmark completion time of MC2D over the Atomic model.

F. Implications of Cores Scaling

The MC2D model is expected to improve synchronization bottleneck in the on-chip network as the core count increases. Therefore, a core scaling study for all three synchronization models is conducted to investigate the impact of the core count on performance. All benchmark–input combinations are executed to completion using 8, 16, 32, and 64 cores. Figure 12 shows the average speedup of MC2D over the Spin and Atomic models as the core count is increased. The speedup of MC2D over both models gets higher with the increase in core count. While MC2D outperforms Spin even at 8 cores, MC2D underperforms Atomic when using less than 32 cores. Figure 13 shows the per-benchmark average normalized completion time of MC2D over Atomic. The performance degradation mainly stems from benchmarks with fine–grain synchronization. As SSSP, TC, and BFS require *service core(s)* for critical section work, it is hard to load balance the worker and service cores as the total core count goes down. Figure 14 demonstrates the load imbalance for both MC2D and Atomic models at different core counts. The Atomic model observes less than 20% variability in instruction count, whereas the MC2D model incurs a much higher variability. This stems mainly from the fact that service cores execute much fewer instructions than the worker cores, specially in SSSP and BFS. SSSP generally requires one or two service cores, while BFS only needs a

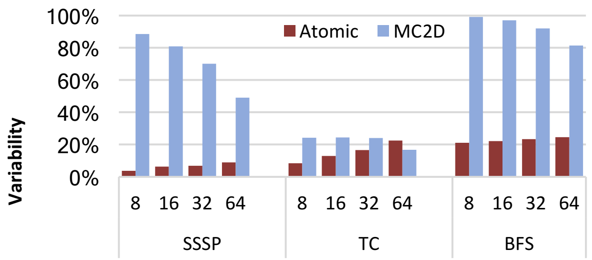


Fig. 14: Load imbalance of Atomic and MC2D models at various core counts; normalized to respective 8-core results.

single service core. However, these cores execute much fewer instructions than the worker cores. Sparing a few cores out of 64 does not hurt performance, even if there is load imbalance between worker and service cores. However, at lower total core counts, this imbalance shows up as performance degradation. Consequently, the performance declines as the core count goes down for the benchmarks with fine–grain synchronization.

VI. Related Work

Accelerating synchronization using explicit communication is first studied in the context of chip multiprocessors by the Alewife and ActiveMsg projects [1], [2]. Here, message passing is integrated into multiprocessors with shared memory architecture to help alleviate the cost of inter-processor communication.

More recently, in the context of single chip multicores, Active Messages (AM) [7] proposes to utilize hardware message passing on top of shared memory to mitigate the bottleneck of serialization on shared data. It requires two contexts per core in which one of them is utilized as interrupt based message handler. Moreover, HAQu [5] and CAF [6] both propose accelerated hardware queues to improve fine–grain synchronization in shared memory multicore systems. While HAQu adds new instructions to accelerate fast queuing in the program's address space, CAF introduces a new hardware structure that is attached to the on-chip network. The idea of using separate cores to handle critical section code is explored in ACS [18]. ACS is similar to the MC2D model, however it does not remove locks and only uses blocking explicit messages. More recently, Dogan et al., [8] explores the MC2D model to accelerate both fine and coarse-grain and synchronization by exploiting data locality and the non–blocking aspect of explicit messaging in the hardware. All the above mentioned works explore the benefits from explicit messaging over traditional shared memory synchronization using *simulation models* of multicores, and do not explore graph analytics for real world input graphs. This paper is the first to our knowledge to explore the novel moving compute to data model on a real multicore machine, Tilera TILE-Gx72, executing real graph inputs from the brain, transportation and social networks. The TSHMEM [10] work investigates barrier synchronization using core–to–core messaging of Tilera machines using micro-benchmarks. However, it does not explore synchronization primitives for the novel moving compute to data model, nor for real workloads, such as graph analytics.

VII. Conclusion

This paper evaluates a novel moving compute to data (MC2D) model to accelerate synchronization in graph processing on the commercial Tilera TILE-Gx72 multicore machine. The MC2D model is compared against traditional shared memory synchronization models based on atomic instructions. The key idea is to accelerate synchronization on shared data by shipping critical section executions to dedicated cores using the machine's auxiliary core–to–core messaging network. By pinning shared data to dedicated cores, the MC2D model improves data locality. In addition, it overlaps communication with computation by utilizing non–blocking messages. Evaluation shows that the MC2D model improves performance of graph benchmarks executing on real world graphs by an average of 34% over Spin, and 15% over the Atomic model.

Acknowledgment

This work was supported in part by Semiconductor Research Corporation (SRC). This research was also partially supported by the National Science Foundation under Grant No. CNS-1718481. The authors wish to thank Christopher Hughes of Intel and Brian Kahne of NXP for their continued support and feedback.

References

[1] J. Kubiatowicz and A. Agarwal, "The Anatomy of a Message in the Alewife Multiprocessor," in *International Conf. on Supercomputing*, 1993.
[2] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *International Symp. on Computer Architecture*, 1992.
[3] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible Architectural Support for Fine-grain Scheduling," in *Architectural Support for Programming Languages and Operating Systems*, 2010.
[4] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to Software: Raw Machines," in *IEEE Computer*, 1997.
[5] D. Tiwari, J. Tuck, S. Y, and S. Lee, "HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor," *International Symposium on High Performance Computer Architecture*, 2011.
[6] Y. Wang, R. Wang, A. Herdrich, J. Tsai, and Y. Solihin, "CAF: Core to Core Communication Acceleration Framework," in *International Conference on Parallel Architectures and Compilation*, 2016.
[7] R. C. Harting and W. J. Dally, "On-Chip Active Messages for Speed, Scalability, and Efficiency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 2, pp. 507–515, Feb 2015.
[8] H. Dogan, F. Hijaz, M. Ahmad, B. Kahne, P. Wilson, and O. Khan, "Accelerating Graph and Machine Learning Workloads Using a Shared Memory Multicore Architecture with Auxiliary Support for In-hardware Explicit Messaging," in *IEEE International Parallel and Distributed Processing Symposium*, May 2017.
[9] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. Brown III, and A. Agarwal, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro, 2007.*
[10] B. C. Lam, A. D. George, and H. Lam, "TSHMEM: Shared-Memory Parallel Computing on Tilera Many-Core Processors," in *Int. Symp. on Parallel Distributed Processing Workshops and Phd Forum*, 2013.
[11] "TILE-Gx72 Processor ," http://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx72.pdf, 2015-16.
[12] Tilera-Corp., "UG527-Application Libraries Reference Manual," 2014.
[13] Tilera, "UG505-Programming The TILE-GX Processor," 2014.
[14] R. A. Rossi and N. K. Ahmed, "The Network Data Repository with Interactive Graph Analytics and Visualization," in *AAAI Conf. on Artificial Intelligence*, 2015. [Online]. Available: http://networkrepository.com
[15] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters," *Internet Mathematics*, 2009.
[16] J. Leskovec and R. Sosivc, "SNAP: A General-Purpose Network Analysis and Graph-Mining Library," *Trans. on Intelligent Sys. and Tech.*, 2016.
[17] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores," in *IEEE Int. Symp. on Workload Characterization*, Oct 2015.
[18] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating Critical Section Execution with Asymmetric Multi-core Architectures," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.