Accelerating Synchronization Using Moving Compute to Data Model at 1,000-core Multicore Scale

HALIT DOGAN and MASAB AHMAD, University of Connecticut
BRIAN KAHNE, NXP Semiconductors
OMER KHAN, University of Connecticut

Thread synchronization using shared memory hardware cache coherence paradigm is prevalent in multicore processors. However, as the number of cores increase on a chip, cache line ping-pong prevents performance scaling for algorithms that deploy fine-grain synchronization. This article proposes an in-hardware moving computation to data model (MC) that pins shared data at dedicated cores. The critical code sections are serialized and executed at these cores in a spatial setting to enable data locality optimizations. In-hardware messages enable non-blocking and blocking communication between cores, without involving the cache coherence protocol. The in-hardware MC model is implemented on *Tilera Tile-Gx72* multicore platform to evaluate 8- to 64-core count scale. A simulated RISC-V multicore environment is built to further evaluate the performance scaling advantages of the MC model at 1,024-cores scale. The evaluation using graph and machine-learning benchmarks illustrates that atomic instructions based synchronization scales up to 512 cores, and the MC model at the same core count outperforms by 27% in completion time and 39% in dynamic energy consumption.

CCS Concepts: • **Computer systems organization → Multicore architectures**;

Additional Key Words and Phrases: Multicore, synchronization, locality

ACM Reference format:
Halit Dogan, Masab Ahmad, Brian Kahne, and Omer Khan. 2019. Accelerating Synchronization Using Moving Compute to Data Model at 1,000-core Multicore Scale. *ACM Trans. Archit. Code Optim.* 16, 1, Article 4 (February 2019), 27 pages.
https://doi.org/10.1145/3300208

1 INTRODUCTION

With the proliferation of shared memory processors with hundreds of cores on chip, thread synchronization has emerged as a significant challenge for performance scaling. Conventionally, thread synchronization is realized using standalone atomic instructions, or using synchronization primitives such as spin-based locks. At small core counts, spin-based synchronization primitives

This research was partially supported by the National Science Foundation under Grant No. CNS-1718481. This work was also supported in part by Semiconductor Research Corporation (SRC). The authors wish to thank Christopher Hughes of Intel and José Joao of Arm Research for their continued support and feedback.
Authors' addresses: H. Dogan, M. Ahmad, and O. Khan, Department of Electrical and Computer Engineering, University of Connecticut, 371 Fairfield Way, U-4157 Storrs, Connecticut 06269 USA; B. Kahne, NXP Semiconductors, 6501 W William Cannon Dr, Austin, TX 78735; emails: {halit.dogan, masab.ahmad}@uconn.edu, brian.kahne@nxp.com, khan@uconn.edu.

1544-3566/2019/02-ART4
https://doi.org/10.1145/3300208

are efficient. However, the overheads of such primitives exponentially increase as the core counts go up. This primarily happens due to expensive cache line ping-pong between cores due to increased on-chip network latency. It also incurs instruction retry overhead that results in higher dynamic energy consumption. Performance scaling can be improved using atomic instructions (when applicable), since they eliminate the overheads of lock acquisition, such as instruction retries and lock variable ping-pong. However, the shared data still ping-pong between cores, and the expensive coherence traffic leads to performance scaling challenges at higher core counts.

The key idea is to keep the shared memory cache coherence and accelerate thread synchronization using a novel moving compute to data model (MC). In the MC model, shared data are logically pinned to a dedicated thread, called *service thread*. The *worker threads* execute application code and invoke requests to update shared data at the *service thread*. By utilizing the MC model, any type of synchronization can be realized without ping-ponging of the shared data, as they are pinned to a dedicated thread. For example, the critical sections can be offloaded to the service thread to accelerate thread synchronization. The MC model can be implemented by utilizing hardware cache coherence in which a software based shared buffer is deployed to communicate messages between worker and service threads. RCL (Lozi et al. 2012) proposes a similar approach to improve performance of POSIX locks using remote core locking. Unfortunately, the shared buffer ping-pongs between the worker and service threads, leading to synchronization challenges at higher core counts. Therefore, in this article, the communication between worker and service threads is carried out using auxiliary send and receive instructions implemented at the hardware-level using a low-latency point-to-point messaging network. Note that all on-chip and off-chip data accesses are still managed using shared memory load and store instructions using the hardware cache coherence protocol.

In-hardware messaging has been investigated by researchers to overcome the limitations of hardware cache coherence. For example, it is first explored by Alewife and ActiveMsg (Kubiatowicz and Agarwal 1993; von Eicken et al. 1992) in the context of shared memory multiprocessors. More recently, RAW (Waingold et al. 1997), ADM (Sanchez et al. 2010), Active Messages (Harting and Dally 2014), HAQu (Tiwari et al. 2011), CAF (Wang et al. 2016), and ACS (Suleman et al. 2009) explore explicit messaging in the context of multicore processors. Commercial Tilera (Wentzlaff et al. 2007) builds processors that support both shared memory and in-hardware messaging between cores. Intel has also announced plans for Queue Management Device (QMD) for more efficient inter-core communication (Moore 2016). Although these works incorporate explicit messaging in one form or another and try to improve communication between cores, they naively exploit its capabilities in a hardware-software combined context, and do not focus on improving thread synchronization in the context of large scale (1,000 cores) multicore processors. This article's focus is to accelerate shared data synchronization using the MC model for futuristic 1,000-core scale multicores.

The MC model pins shared data at the *service thread*, and thus enhances shared data locality. Moreover, by utilizing hardware based explicit messaging to enable non-blocking communication, the *worker threads* overlap the execution of critical code sections with other useful work. In addition, as compared to the lock based critical sections, it gets rid of the lock related overheads, such as instruction retries and the mutex variable ping-pong. Utilizing a single *service thread* may become a performance bottleneck due to serialization of multiple requests. Therefore, to exploit concurrency across disjoint critical sections, multiple service cores are assigned as *service threads* and the shared data are divided among them. The remaining worker threads exploit concurrency in the underlying algorithmic code, and direct requests to the corresponding service threads. This article proposes to utilize distribution of the *worker* and *service threads* among dedicated cores in a spatial setting.

The key challenge of the spatial MC model is the need to load balance the work between cores executing the worker and service threads to obtain near-optimal performance scaling. One idea is to temporally map a *worker* and a *service thread* in each core, similar to Active Messages (Harting and Dally 2014). This achieves load balance, but at the cost of doubling the number of threads relative to the core count. More threads now participate in synchronization, and thus potentially increase the overall communication stalls on chip. The temporal MC model is expected to outperform the spatial model at lower core counts, but as the number of cores approach hundreds to a thousand cores on chip, the spatial MC model is expected to better utilize the on-chip network resources. However, load balancing the worker and service thread counts can be challenging as it is workload dependent. Therefore, this article explores a heuristic to determine load balanced mapping of the worker and service threads. The heuristic relies on the percentage of shared work in an application to decide the number of service cores. If selected properly, the service cores match the concurrency needs of the shared work, while the worker cores optimally exploit concurrency in the remaining algorithmic work for a given workload. The contributions of this article are as follows:

(1) A spatial moving compute to data (MC) model is proposed utilizing low-latency hardware explicit messages to accelerate synchronization. The MC model mitigates cache line ping-pong, improves data locality, and hides communication latency with non-blocking messaging. These key aspects deliver high performance scaling for both fine and coarse grain synchronization in parallelized workloads from the machine-learning and graph processing domains.
(2) The MC model is prototyped on *Tilera Tile-Gx72* multicore machine. Up to 64 cores are explored, and at 64 cores, the MC model demonstrates a 10% performance advantage over atomic instruction based synchronization.
(3) To explore 1,000-core scale multicore, a RISC-V based multicore simulation environment is utilized to characterize the spatial and temporal MC model implementations, and compare performance and energy consumption over both spin-lock and atomic instruction based synchronization models. In addition, the spatial MC model is evaluated against a software shared buffer based moving compute to data model. The spatial MC model is shown to enable superior performance scaling up to 1,024 cores.
(4) A heuristic based on profiling the percentage of shared work is introduced for efficient spatial distribution of worker and service cores in the MC model. The proposed heuristic automates thread distribution for a 5% performance loss compared to a manually tuned configuration.

2 THREAD SYNCHRONIZATION MODELS

Shared memory hardware cache coherence provides ease of programming, flexibility on sharing data between threads, and seamless data movement. However, thread synchronization becomes a significant performance bottleneck at higher core counts. This is mainly due to expensive ping-pong of shared data between private caches of the participating cores. Traditionally, spin-based synchronization primitives, such as locks are realized using atomic instructions (e.g., load-link and store-conditional instructions) to update shared data in a thread safe fashion. However, to realize such synchronization, a separate lock variable needs to be acquired before getting into critical code section. At lower core counts and under low contention, spin-lock is efficient as it enables concurrent execution of critical sections. However, when there is contention on shared data, the threads often fail to acquire the lock, therefore they spin over the shared lock variable until it is available. This process easily boosts the locking overheads due to instruction retries and ping-ponging of the lock variable between cores. In addition, as the number of cores in the system increases, the cost of lock acquisition drastically goes up due to increased network latency.

The locking overheads can be eliminated by directly utilizing atomic instructions. Instead of acquiring a lock variable to protect a critical code region, standalone atomic instructions are employed. These instructions are implemented using the hardware coherence protocol, where each atomic instruction performs an exclusive read to lock the cache line in the level-1 cache, performs the operation, and stores the result before unlocking the cache line. If another core wants to perform an atomic operation on the same cache line, then it needs to acquire exclusive copy to perform the operation atomically. However, the shared cache lines still bounce between cores when multiple threads access them temporally. Therefore, as the number of cores increases, the bouncing affect leads to degradation in performance due to elevated network latency. Another key limitation of atomic instruction based synchronization is the limited number of operations in the ISA for implementing a diverse set of critical code sections. As a result, as opposed to spin-lock based synchronization, they are not applicable to any arbitrary critical section.

In this article, for more efficient and generic thread synchronization, a novel moving compute to data (MC) model is proposed and evaluated against both spin-based (Spin) and atomic instruction (Atomic) based synchronization models. The MC model and its architectural implementations are discussed in detail in subsequent sections.

2.1 Moving Compute to Data Model

Moving computation toward data technique has gained tremendous popularity in recent years due to explosion of computing on massive datasets. Traditionally, distributed computing domain has deployed computation migration to mitigate performance and energy bottlenecks of moving large amount of data between server nodes. In this model, a data segment is pinned to a node, and the executable is moved toward it. As the executable is significantly smaller than the data, moving overhead is also notably smaller. In a single chip multicore processor, we propose to utilize this approach to mitigate the bottleneck of shared memory thread synchronization, especially as the core count approaches 1,000-core scale.

In the proposed MC model, the protected shared data are mapped to a dedicated core and updated only at that specific core by moving computation toward it using explicit messages. In the context of fine-grain synchronization, locks and atomic instructions in a traditional shared memory application are eliminated, and the critical code sections are moved to the dedicated core, termed as *service thread*. The remaining cores are utilized as *worker threads*. The *workers* execute the application work, and send explicit request messages to invoke fine-grain synchronization at the *service thread*. Deploying only a single core as *service thread* may result in higher serialization overhead, hence multiple cores are assigned as *service thread* to exploit concurrency across independent critical code sections. In this case, shared data are distributed among the available *service threads*, and the *workers* forward their requests to the corresponding *service thread* by utilizing a software lookup function. The amount of data that is sent for a critical section request depends on the application. While some workloads only require a single word, others need multiple words of data. The *service thread* then receives the required number of words in the order they were sent, and execute the critical code section. An application may or may not require the *service thread* to send a reply back to the requesting *worker thread*. This decision may be needed to ensure data consistency.

Barrier synchronization is an example of coarse-grain synchronization in which blocking communication is required. Instead of loading and updating the barrier variable atomically by each core, the cores send "barrier" messages to a predefined *service thread*. After sending the "barrier" message they wait for an explicit reply from the *service thread* that manages the barrier. When the *service thread* receives all the messages, it replies with a "continue" message to each participating core. This way, instead of spinning over a shared variable, and ping-ponging the cache line between threads, synchronization is done by communicating with a *service thread* via explicit messages.

This article employs one of the workers (Core 0) as the *service thread* to manage barrier synchronization. When the Core 0 completes its worker task, it starts handling the barrier messages.

The proposed MC model can be realized either by employing software shared memory buffer based inter-core messages, or by introducing in-hardware explicit messaging. Both implementations are discussed in the next subsections.

2.1.1 Moving Compute to Data Model Using Shared Memory Messaging (MC_shmem). Similar to explicit communication in MPI (Gropp 2002), the messaging between worker and service threads is accomplished using a shared software buffer per service thread. However, as opposed to MPI programming model, MC_shmem utilizes the shared memory programming model. MC_shmem approach is similar to RCL (Lozi et al. 2012) work in which locking is done in remote cores, and requests are delivered using a shared request buffer per server core. Similarly, in MC_shmem, a shared buffer per service thread is utilized for the communication between worker and service threads. Each buffer slot contains a flag and a place holder for the data to be sent. To be able to send a message to a particular service thread, a worker atomically increments the write pointer of the corresponding buffer, then places its data into the slot, and sets the flag. The atomic increment on the pointer makes sure that multiple workers do not write to the same slot. The service thread starts from beginning of the buffer, checks the flag of each buffer entry, reads the data, and performs the critical section. If the flag is not set, then the service thread spins over the flag until data are available. The shared buffer is implemented in a way that it has enough capacity to hold all the requests. It can also be implemented with limited capacity as a ring buffer. However, experimental results for the workloads of interest suggest that utilizing a regular buffer outperforms the ring buffer. Therefore, a large shared buffer per service thread is utilized in this article.

The MC_shmem model pins shared data at the service threads to exploit locality, and also benefits from non-blocking communication. However, the shared buffer data itself bounces between worker and service threads. The aforementioned non-blocking communication may ease the cost of ping-ponging by allowing the worker threads to hide communication latency. However, at higher core counts, the impact of cache line ping-pong is expected to limit performance. As a result, to enable efficient implementation of the MC model, hardware support for explicit messaging is introduced as an auxiliary mechanism.

2.1.2 Moving Compute to Data Model Using In-hardware Explicit Messaging. Four explicit messaging instructions for low-latency core-to-core communication are introduced in the ISA, and the required micro-architectural support is added to each core (cf. Section 3).

(1) **Send** instruction does not block the pipeline. It requires the destination core's address along with the data to be sent from the sender's register file to the receiver's register file.
(2) **Recv** instruction stalls the pipeline if the data are not present at the receive queue of the core that issued this instruction. Once the data arrive, the *recv* instruction pulls the data from the receive queue and places it into the register file.
(3) **Sendr** (send with rendezvous) instruction is a blocking send instruction in which an explicit reply is expected from the destination core.
(4) **Resumer** (resume rendezvous) instruction is a non-blocking special send instruction that is used to reply to the *sendr* messages.

The details of the explicit messaging protocol and the architectural extensions are discussed in Section 3. By utilizing these low-latency messaging instructions, worker and service cores exchange messages between their register files without involvement of cache coherence traffic. The worker threads make use of the *send* instruction to request critical section execution by pairing it with a *recv* instruction at the corresponding service thread.

In-hardware MC model provides two key advantages. First, it prevents unnecessary ping-pong of shared data by pinning it to service threads. However, for some applications, certain shared data benefits from concurrent reads by worker threads for work efficient implementation of the algorithm. The MC model utilizes hardware cache coherence to enable efficient movement of such data between cores. The second advantage is that if non-blocking communication is utilized, the MC model enables efficient overlap of communication traffic with other computation. In addition, the worker threads can have multiple back to back in-flight request messages, and possibly further ease the overheads of worker to service thread communication. However, the shared memory based models suffer from the overheads of cache line ping-pong. The benefits of the MC model are expected to be more notable at higher core counts as the distance between sharer cores increases.

2.1.3 Worker and Service Thread Distribution. The main challenge with the MC model is the determination of the right number of worker and service threads in the spatial setting to exploit application parallelism. This approach requires tuning the ratio of *worker* and *service* threads to achieve near-optimal performance, otherwise the system suffers from load imbalance. Another approach is to utilize two contexts per core and temporally employ the same core for both worker and service threads. In the following two subsections, both distribution approaches are discussed in more detail, and they are evaluated in Section 6.

Spatial Moving Compute to Data Model: A naive way to achieve thread mapping is to perform an exhaustive search by varying the number of *worker* and *service threads*, and determine the best performing mapping. This approach may be used at low core counts, however it gets time consuming as the number of cores increases. In addition, each workload is expected to require different ratio due to its unique data structure and synchronization requirements. This article proposes a profiling based heuristic, which relies on the correlation of the number of *service threads* and the percentage of shared work in a given workload. In this method, the shared memory version of the application is profiled to obtain the average time spent in critical code sections (shared work) compared to the total completion time. If the time spent in critical code sections is high, then the required *service thread* count is also high and vice versa. If the shared work percentage results in less than one *service thread* due to very small shared work, then it is assigned a single service thread. Moreover, at most half of the cores are assigned to the *service thread* task, because the work being done by each worker thread increases as the number of workers decreases.

Temporal Moving Compute to Data Model: To support the temporal MC model, each core needs to be extended with two register files, an explicit messaging-aware switching policy logic, and selection logic for register reads/writes to support hardware multi-threading. Each hardware context in a core is then mapped to a single *service thread* and a single *worker thread* for temporal mapping. Hence, the number of worker and service threads is always equal to the number of used cores. This approach eliminates the need to tune the number of *worker* and *service threads*. However, it requires an additional context and special switching policy that takes explicit messaging into account for fast context switching. In addition, the number of threads participating in synchronization becomes two times the number of cores, which may incur additional communication stalls at higher core counts. Consequently, the spatial MC model is preferred at higher core counts, since (1) the available cores are relatively easier to load balance, and (2) the number of threads participating in synchronization must be kept in check to minimize unnecessary communication stalls.

3 IN-HARDWARE IMPLEMENTATION OF THE MC MODEL

The baseline is a tiled shared memory multicore architecture. Figure 1 shows a logical view of a tile within the proposed processor. The tiles are interconnected with a two-dimensional (2D) mesh

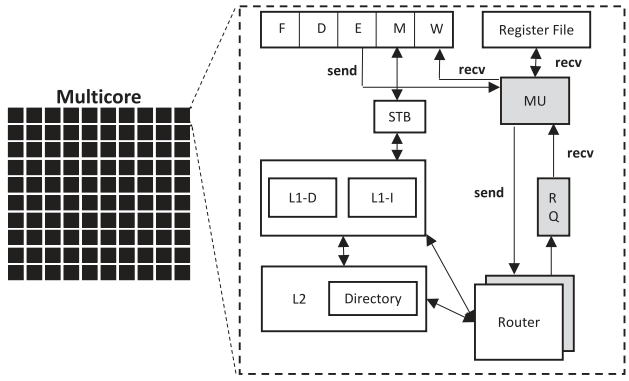

Fig. 1. Overview of a tile and architectural extensions.

on-chip network. Each tile includes a single issue RISC-V (Waterman et al. 2014) core, private level-1 instruction and data caches, a shared last-level cache slice with an integrated directory for MESI cache coherence protocol, and a network router for inter-core communication. Memory controllers are attached to some of the tiles to enable off-chip memory accesses. Four explicit messaging instructions are added into the RISC-V ISA, namely *send, recv, sendr, and resumer*. The shaded modules in each tile are introduced to support these instructions. A receive queue (RQ) per tile is introduced to buffer explicit messages. The on-chip network is also extended to ensure safe transmission of messages. Messaging unit (MU) generates the necessary control signals to interact with the core pipeline and the RQ. It also creates the packet for send instructions, and then forwards the packet to the router. Similarly, received data are read from the RQ, and placed into the specified registers by the MU. Note that the shared memory cache coherence is retained in the system, and the explicit messaging support is added as an auxiliary support to achieve efficient implementation of the proposed MC model. In addition to explicit messaging capability, per-core four-way SIMD that can operate on four 16-bit floating point numbers is added to enable state-of-the-art implementations of machine-learning algorithms. Associated instructions, such as fused-multiply-add are adopted as an extension to the RISC-V ISA. Furthermore, RISC-V ISA's standard extension for atomic instructions (Waterman et al. 2014) are implemented. These instructions, such as load-reserve and store-conditional are employed to implement shared memory synchronization primitives.

In default mode, the cores are single-threaded, and the application threads are spatially distributed among available cores. In addition to spatial mode, multiple threads per core with hardware level context switching are supported to enable the temporal MC model (cf. Section 2.1.3). Each core is extended with two register files, an explicit messaging-aware switching policy logic, and selection logic for register reads/writes to support hardware multithreading. The switching policy interacts with the receive queue to initiate thread switching when a message arrives. In the temporal MC model, it is crucial for *service thread* to perform its work prudently, because, otherwise, the receive queue may suffer from contention, and possibly also lead to application level deadlock. Therefore, the *service thread* is given higher priority, and whenever the receive queue receives a message, the policy switches to the *service thread* and all messages in the queue are processed.

3.1 Explicit Messaging Protocol

As discussed in Section 2.1, the MC model utilizes both blocking and non-blocking communication to accelerate fine and coarse-grain synchronization. The introduced explicit messaging support provides the capability to realize both types of communication between worker and service threads.

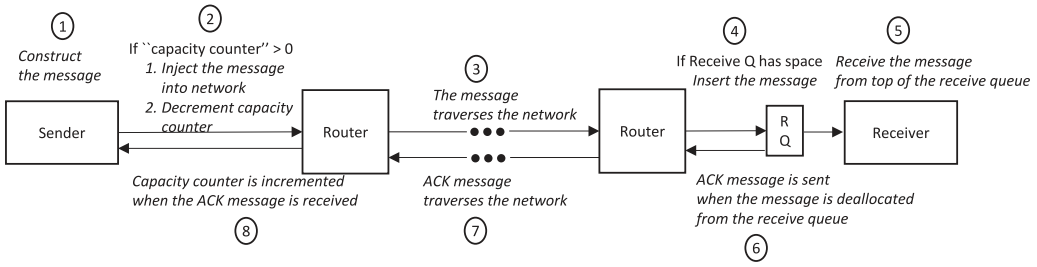

Fig. 2. Explicit messaging protocol.

Non-blocking Communication: This communication type is utilized to implement the MC model to achieve fine-grain non-blocking synchronization. A *send* instruction at the worker thread is paired with a corresponding *recv* instruction at the service thread to implement non-blocking core-to-core communication. A *send* instruction does not block the pipeline if the messaging network is available to inject the message. This allows worker thread to continue with other useful work while the message traverses the network to its destination. Moreover, the worker can have multiple in-flight messages as long as the network flow-control permits. This type of communication helps overlap communication latency with other computations.

Figure 2 illustrates the protocol implementation for core-to-core non-blocking communication. First, the destination address is calculated using the receiver *CoreID*, and placed into an architected register. Then, a message is constructed by the sender core's pipeline by executing *send* instruction with the address and the data ①. The constructed message consists of a header containing the destination address, message size, and the payload. Each send instruction supports up to four words. The message can contain a pointer to a function along with the necessary data to be executed, or just arbitrary data that the destination core needs to perform some computation. The programmer needs to make sure that the receiver side decodes what type of message, and how many words are being sent to it. The protocol utilizes a special per-core counter called *"capacity counter"* ②, and an implicit *ACK* message ⑥ to enable flow-control for messaging. The *capacity counter* tracks the number of in-flight messages, and the senders cannot have more in-flight messages than the set *capacity counter* value. This counter is essential for supporting thread migration and virtualization in the proposed architecture. The programmer sets this counter by setting a special register at the beginning of the program execution. When a message is inserted into the network, the corresponding core decrements its capacity counter. When the counter value reaches to 0, the *send* instruction is stalled in the pipeline. When the message is injected into the network, it is routed to the destination core using the on-chip network ③. For this protocol to work correctly, it is assumed that the messages from a source to a destination are ordered in the network. In addition, the routing algorithm is assumed to be deadlock-free. Once a message arrives at the destination's receive queue ④, it is pulled by the destination core's *recv* instruction ⑤. The *recv* instruction always blocks the pipeline. If the core executes the *recv* instruction before the message arrival, then it stalls until the data arrive at the receive queue. The programmer is responsible for adding subsequent code to decode the received message, and initiate execution of the appropriate code region using the received data. After each message is read from the receive queue, an implicit *ACK* message is generated to traverse back to the source core ⑥, ⑦. The *send* and *ACK* messages use separate networks (in addition to the ones used for cache coherence) to avoid deadlock, as utilizing the same network for both type of messages may lead to circular dependencies in the network. The *capacity counter* is incremented implicitly upon receiving the *ACK* ⑧, and the sender core (if stalled) is allowed to proceed.

Blocking Communication: In several application scenarios, strong consistency is required, or a piece of data is needed from the destination core. In this case, the sender waits for an explicit reply from the receiver. It can be implemented by executing a *recv* instruction followed by a *send* instruction in the sender core, and a *send* instruction followed by a *recv* instruction in the receiver core. Unfortunately, *send* and *recv* instruction pairs may result in a deadlock if the receive queue has finite size, and both communicating cores use the same network to send their messages. For example, consider a master core that dispatches work to the worker cores. All the workers send work request messages to the master, and then wait for their explicit reply by executing a *recv* instruction. If the number of messages sent are more than the receive queue size of the master core, then the messages block the network responsible for the send traffic. When the master tries to inject a send message to the router for replying to one of the workers, it cannot proceed, because the send network is filled with the overflown messages. Moreover, the master core cannot pull any more data from the receive queue, because it is stuck at executing the send instruction. Hence, the deadlock situation occurs. Therefore, for the blocking communication, special *sendr* and *resumer* instructions are implemented. In this case, the explicit reply messages are always sent using a *resumer* instruction that flows on the dedicated reply network with *ACK* messages.

Unlike *send*, the *sendr* instruction blocks the pipeline until the *resumer* reply message is received at the sender core. At the receiver core, the *recv* instruction is utilized to receive the *sendr* message. However, the sender address is stored to be utilized by the *resumer* instruction. This explicit *resumer* reply message is routed back to the sender core. This message is directly delivered to the pipeline without getting into the receive queue. Upon receiving the message, the *sendr* instruction completes. The implementation of the MC based barrier as described in Section 2.1 is realized employing these two instructions. The workers participating in the barrier utilize *sendr* instruction for barrier message to the master core. The master core receives all the messages with *recv* instruction, and resumes the participating cores with *resumer* instruction.

Application Level Requirements: To avoid application level deadlock, the proposed architecture allows messages from different sender cores to arrive in *any order* at the destination core. Ordered message arrival can only be enforced if the architecture enables receive queues for each sender core, which is an unnecessary burden on the hardware. To keep the overhead of receive queue per core low, the unordered message arrival must be handled in the application software. The programmer must decode each received message, and invoke the appropriate software routine(s) to handle the request from the corresponding sender core.

3.1.1 Message Consistency. An application may require message consistency, i.e., a sender thread must ensure the delivery of a message to its destination before commencing with other work. The ISA is extended with a *message fence* instruction, which ensures that all pending messages are observed at the receiving side. This is ensured by monitoring the *capacity counter*, since it tracks all in-flight messages whose *ACKs* have not been observed yet. Once the *capacity counter* reaches its initialized value, all sent messages have been observed at their respective destination. At this point, the message fence instruction commits.

3.1.2 Thread Migration and Multiprocessing Support. Supporting thread migration is a necessity for a general purpose processor. However, the proposed architecture can deadlock if in-flight explicit messages are not dealt with properly. This can happen if an in-flight message is delivered to a core where the thread is not running any more. To properly handle this situation, a cleanup mechanism is required to ensure all in-flight messages are delivered before thread migration can commence. The operating system (OS) halts all cores from injecting messages into the network. After that, the OS monitors the *capacity counter* of each core and waits for them to get back to their initialized values. This signifies that all cores have received *ACKs* for their in-flight messages, and

there is no explicit message in the network. At this point, the OS can perform thread migration. It also updates the thread-to-core mapping so that future messages can get to their destination properly. Hardware virtualization can also be supported based on this mechanism.

3.2 Explicit Messaging Hardware Overhead

The architecture requires a receive queue per core to support the proposed protocol, as seen in Figure 1. The size of each core's receive queue is determined empirically by conducting a study similar to the one presented in Dogan et al. (2017). All workloads are run, and a counter is utilized in the simulator to determine maximum utilization of receive queues at any given time for each workload. Then, the maximum utilization among all the workloads is used to size the receive queue. The study is repeated by varying the *capacity counter*. As the *capacity counter* value increases, the maximum utilization at the receive queue boosts, because each core is allowed to have more in-flight messages. However, it also improves performance. Therefore, it is important to determine a value that balances maximum receive queue utilization and performance. For this work, the *capacity counter* is determined to be 4, and the receive queue size per core is determined to be 2.4KB. In addition to the receive queues, cache coherence and explicit messaging traffic is separated from each other using independent on-chip networks to avoid deadlock.

3.3 Prototyping Explicit Messaging and Cache Coherence on TILE-Gx72 Machine

Tilera's *TILE-Gx72* processor is a commercially available machine that enables similar capabilities to the proposed messaging protocol (Dogan et al. 2018). It is a tiled multicore architecture with 72 tiles interconnected with 2D mesh networks-on-chip. Each tile consists of a 64-bit VLIW core, private level-1 data and instruction caches, and a shared level-2 (L2) cache. The common architectural features are as follows:

(1) Directory based cache coherence protocol for data movement controls
(2) Atomic instructions at hardware level for thread synchronization
(3) Auxiliary in-hardware explicit messaging (similar to send and receive) using a User Dynamic Network (UDN).

The explicit messaging network in TILE-Gx72 supports both blocking and non-blocking communication. However, since it does not utilize separate instructions for the reply messages of the blocking communication, it may result in deadlock situation as discussed in Section 3.1. The blocking communication is only used to implement barrier synchronization in this article, therefore, the offered explicit messaging support is sufficient to realize the barrier implementation.

4 PROGRAMMING WITH MOVING COMPUTE TO DATA MODEL

The proposed MC model utilizes the shared memory parallel programming model. Threads are created within a process using the Pthreads library, and all threads are allowed access to shared data structures. Even though Pthreads is utilized in this article, OpenMP programming model can be easily adopted. The programming model replaces traditional thread synchronization with the explicit messaging based MC protocol. Similar to any traditional shared memory application, threads are created and distributed by either jumping to worker routine or service routine. The service threads perform critical section execution with the request of the worker threads, as discussed in Section 2.1. The process of transforming an application to the MC model can be automated by detecting thread synchronization points in the code. The identified critical sections can be moved to a separate procedure, then service threads are to be assigned to these procedures. As similar to RCL (Lozi et al. 2012), Coccinelle (Padioleau et al. 2008) or similar refactoring tools can be easily utilized to transform existing applications. However, this article performs manual transformation

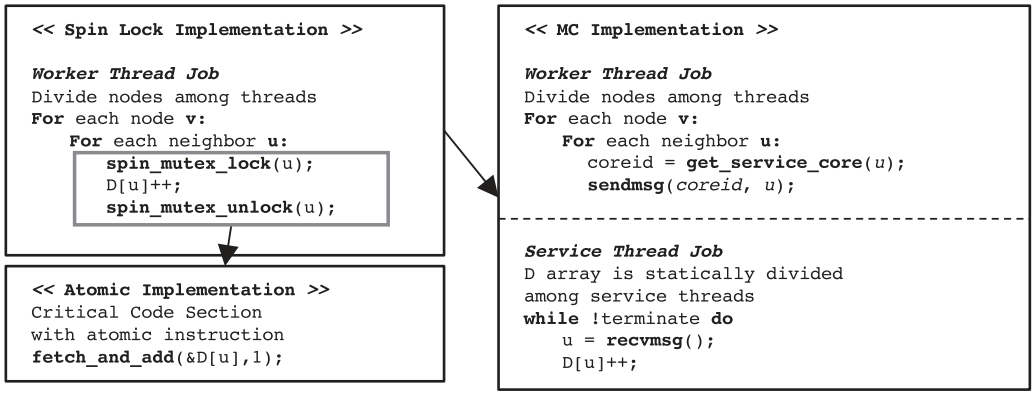

Fig. 3. Pseudo code of triangle counting implementation using Spin, Atomic and MC.

of representative applications to illustrate how shared memory synchronization can be ported to the MC model.

4.1 Triangle Counting (TC)

Triangle counting (TC) is a well-known graph algorithm that counts triangles in a graph for various statistical purposes in an application. Figure 3 demonstrates the implementation of TC using spin-based locks, atomic instructions and the MC model. A shared data structure is maintained to count the connectivity of each node, and it is updated atomically using spin locks (upper left box in Figure 3). After counting the connectivity for each node, all the participating threads hit a barrier. Then each thread calculates its local triangle count using a heuristic. Finally, the total triangle count is determined by aggregating the local counts. TC does not include any test before critical section, which results in acquiring a lock multiple times for each node. Therefore, the contention on shared data are expected to be high for this algorithm. Consequently, the lock acquisition overhead is also elevated due to retries and cache line ping-pong for both shared data and the lock variables. The algorithm can be implemented using lock-free data structure by employing the atomic fetch-and-add (FAA) instruction (lower left box in Figure 3). This significantly reduces the overhead of acquiring locks as the atomic FAA instruction does not fail. However, the shared data themselves still ping-pong between cores.

The MC model pins all shared data structures at dedicated cores and ships the critical section work to *service threads* (right side of Figure 3). Only the neighbor node ID is needed for critical section, hence the *worker thread* sends one word of data as message to the corresponding *service thread*. Similarly to Atomic, the MC model also eliminates locks and related overheads. In addition, it also pins shared data to *service threads*, and prevents cache lines from unnecessarily bounce between cores. Moreover, by utilizing non-blocking communication, it overlaps communication overheads with other useful work. For instance, while one request is being propagated in the network, the worker can load the next neighbor ID, and execute lookup function to determine the *service thread* ID for the next request.

4.2 Single Source Shortest Path (SSSP)

SSSP is used to compute the shortest path for a user defined source node in a graph. The algorithm is parallelized using an outer loop parallelization strategy in which the nodes are accessed in a controlled manner. The range of nodes that can be visited is calculated until all the nodes are accessible. The nodes in each range are divided among cores, and the cores visit and relax the

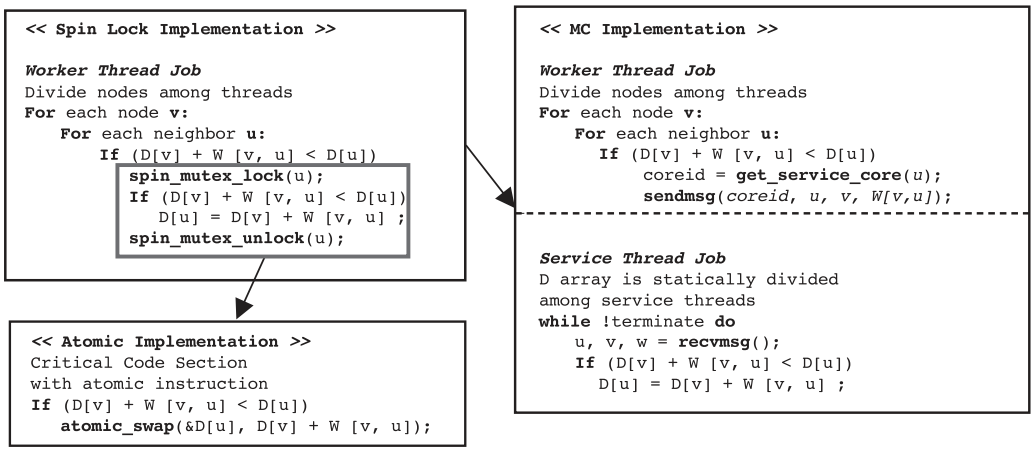

Fig. 4. Pseudo code of SSSP implementation using Spin, Atomic and MC.

neighbors of their nodes one at a time. As seen in Figure 4 (upper left), the node distances are updated using locks, as threads may update the distances of common neighbors at the same time. The lock is acquired only when the test before lock acquisition fails to eliminate unnecessary locking. However, since the algorithm may not converge easily depending on the input, the test may not fail as often. Therefore, this results in multiple lock acquisitions per node, hence the contention on the shared data increases. Similar to TC, SSSP is also accelerated using an atomic swap instruction by removing the locks and related overheads.

The MC implementation of SSSP, similar to TC, moves the critical section to *service threads* (the right box in Figure 4). The relaxation critical code section is executed by the *service thread*. Even though it causes the distance D-array to be read by the workers and results in additional coherence traffic, the test before critical section is retained to make sure the algorithm is work efficient. This prevents sending unnecessary critical section requests to the *service threads*. Contrary to TC, SSSP's critical section requires three words to execute. The workers perform the test as in the case of spin and atomic versions, and if the check fails, they send the critical section execution request with the required data words to the corresponding *service thread*. The *service threads* receive their request messages, and relax the shared distance D-array. In SSSP, the shared D-array may ping-pong between worker and service threads, since the tests before sending the request message require the workers to read shared data. However, all updates to the D-array are performed at a single location at the dedicated service thread. The MC model still benefits from non-blocking communication. As this algorithm requires loading multiple data items before sending a request message, it offers a lot of room to overlap communication overheads with the memory stalls.

4.3 AlexNet, a Deep Convolution Neural Network

AlexNet consists of five convolutional layers and three fully connected layers. Since most of the computation is in convolutional layers, here we only discuss parallelization of the convolutional layer. Briefly, other layers are parallelized by dividing the neurons among all available threads.

A naive coarse-grain parallelization strategy is that all the neurons are tiled, and tiles are divided among available threads. Each thread performs the computations for the neurons in its tiles. For parallelization, each layer passes its output data to the next layer by incorporating barrier synchronization at the end of each layer.

An optimized implementation makes use of fine-grain parallelization that further improves data reuse per thread. This is achieved by dispatching multiple threads to work on a single neuron. This

```
                    Worker Thread Job

Divide the channels among group of threads
start = thredId      * nChannels/nThreads
stop  = (threadId+1) * nChannels/nThreads

For each ch in range(start,stop)
   For each y in range(0, outH)
      For each x in range(0, outW)
         Perform convolution for one channel and send
         psum = convolution(filter, input, ch, y, x);
         sendmsg(AccumCore, psum, nrnId)
```

```
        Service Thread Job

while !terminate do
    psum, nrnId = recvmsg();
    Output[nrnId] += psum;
```

Fig. 5. Pseudo code for fine-grained parallelization of convolution layers using MC model.

requires updating the same neuron output by multiple threads. The update by multiple threads can be realized using shared memory spin locks. However, it does not scale as well as the coarse-grain approach due to the large overheads of shared memory locks. It can also be implemented using atomic floating point fused-multiply-and-add instruction. However, this type of operation is not available as a single atomic instruction. Therefore, we have implemented it using the MC model with the explicit messaging capabilities, as depicted in Figure 5. As mentioned earlier in Section 2.1, MC generalizes atomics to any set of operations. Hence, a load, store and multiply-add operation on floating point variables is easily realized with the MC model. As seen in the algorithm, the cores are clustered into small thread groups, and each group works on a tile of neurons. One of the threads in each group is assigned as *service thread* to accumulate the partial sums. To calculate a neuron output, the kernel channels are divided among the worker threads in a group, and the partial sum of each neuron for each kernel channel is calculated by the workers, and sent to the *service thread* (the left box in Figure 5). The *service thread* receives the partial sums, and accumulates for the corresponding neuron (the right box in Figure 5). Since this approach deploys a fine-grain parallelization strategy, it enables higher concurrency without losing the data reuse benefits.

This article also evaluates an inception neural network, SQUEEZENET, which is implemented using the coarse-grain parallelization approach. This network incorporates a small number of channels, and thus assigning multiple threads to work on a single neuron is found to be not beneficial.

4.4 More Complex Critical Sections

The focus of this article is to utilize the MC model to accelerate critical sections in a shared memory application at 1,000-core scale. Therefore, the most common critical sections on regular data structures are the focus of the article. As seen in the examples in previous subsections, the process of converting an application to utilize the MC model involves moving the critical sections to dedicated service threads, and dividing the shared data between the dedicated threads to maintain atomicity of operation. This idea can be generalized to any data structure and critical sections. For example, the nodes in a linked list can be mapped to service threads based on their memory addresses. However, the critical sections may become more complex. Traditionally, a concurrent linked list is implemented using hand-over-hand locking (Bayer and Schkolnick 1988). To be able to perform an operation on a certain node, the lock for the current node is acquired before releasing the lock of the previous node. The implementation of MC in this case requires communication between service threads as a result of sequential nature of the linked list. Similar to any other previously discussed workload, the workers initiate critical section requests by sending a message to the service thread mapped to the head of the list. Then the service thread directs the message to

Table 1. Architectural Parameters for Evaluation

Architectural Parameter	Simulator	TILE-Gx72
Number of Cores	up to 1,024 @ 1GHz	72 @ 1GHz
Compute Pipeline per Core	In-Order, Single-Issue	64-bit VLIW
Memorgy Subsystem		
L1-I Cache per core	8-32KB, four-way Assoc., 1 cycle	32KB, two-way Assoc., 2 cycles
L1-D Cache per core	8-32KB, four-way Assoc., 1 cycle	32KB, two-way Assoc., 2 cycles
L2 Inclusive Cache per core	16-256KB, eight-way Assoc.	256KB, four-way Assoc., 10 cycles 2 cycle tag, 4 cycle data
Directory Protocol	Invalidation-based MESI ACKwise$_4$ (Kurian et al. 2010)	Invalidation-based Fullmap
Num. of Memory Controllers	4 to 16	4
DRAM Bandwidth per Controller	10GBps	12GBps
Electrical 2D Mesh with XY Routing		
Hop Latency	Two cycles (1-router, 1-link)	
Contention Model	Only link contention (Infinite input buffers)	
Flit Width	64 bits	64 bits
Explicit Communication		
Receive queue per core	2.4KB	4×128 words

the service thread where the next node is mapped to, and is blocked until it gets a reply back from that service thread. Two service threads are blocked at a time similar to hand-over-hand locking. This types of data structures are contended and hard to scale, hence by pinning the nodes to prevent ping-ponging and utilizing non-blocking communication on the worker side, it is expected to get benefits from the MC model. However, since the purpose of this article is to focus on the most common critical sections on regular data structures, further discussions and analysis are not included in the article.

5 EVALUATION METHODOLOGY

5.1 Tilera Multicore Machine

The TILE-Gx72 multicore platform is deployed for evaluation of the proposed synchronization models. As discussed in the Section 3.3, the machine incorporates 72 cores. Each tile contains 32KB private level-1 instruction and data caches, and a 256KB shared level-2 cache slice. It executes at 1GHz and is equipped with 16GB of DDR3 main memory. The architectural features of the processor are summarized in Table 1. The platform supports a linux version that is modified for the Tilera architecture. A modified version of GCC 4.4.7 that supports Tilera specific features is utilized for compilation of the benchmarks. Up to 64 cores in the system are utilized for performance evaluation. While running experiments, no other program that can interfere with the application is active. Completion time is measured by running each workload to completion. Memory allocations, initialization of data, and thread spawning overheads are not taken into account for performance measurements. Each run is repeated ten times and the average number is reported to obtain more accurate benchmarking time.

5.2 RISC-V Multicore Simulator

To evaluate the proposed synchronization models at up to 1,000-core scale, the TILE-Gx72 is supplemented with a multicore simulation environment.

5.2.1 Simulator Setup. The proposed architecture is implemented using an in-house industry-class RISC-V simulator and the associated tool chains. The simulator utilizes an Architecture Description Language (Kahne 2013) for functional model implementation, which in turn drives the performance models. Table 1 summarizes the architectural parameters of the simulated system. Similar to TILE-Gx72, a futuristic tiled multicore processor with a private L1 and shared L2 cache hierarchy per core is evaluated. The number of simulated cores are varied from 64 to 1024. When increasing the core count, the cache size is kept similar to TILE-Gx72's total on-chip cache capacity of 22MB by adjusting the per tile cache sizes. The number of memory controllers are increased when increasing the core count. While 4 memory controllers (40 Gbps) are utilized at 64 cores, 16 memory controllers (160Gbps) are utilized at 1,024 cores. Single threaded cores are utilized for the spatial MC model, as well as Spin and Atomic models. However, two threads per core are evaluated for the temporal MC model implementation.

5.2.2 Compiler Support. RISC-V tool chain is used for compiling benchmark applications. Since ISA extensions are not recognized by the compiler, the wrapper functions that contain explicit messaging instructions using GCC extended asm blocks are used to direct the compiler to use specific registers.

5.2.3 Performance Models. The performance models used in the simulator are ported from the Graphite multicore simulator (Miller et al. 2010). The simulator implements the following models; core pipeline, cache hierarchy, cache coherence protocol, and on-chip network. The XY routing is utilized for the mesh interconnection network. The per hop delay is set to 2-cycle, and the network model accounts for the pipeline latencies related to loading and unloading the packets to the network routers (Dally and Towles 2004; Park et al. 2012). It also includes the contention delays. In addition, the explicit messaging instructions, and the related protocol overheads are integrated into the performance models.

McPat (Li et al. 2009) is utilized to acquire per event dynamic energy numbers for both the core energy and memory system energy using 22nm technology. Then, the numbers are scaled down to 11nm by using the scaling constant from Huang et al. (2011). The receive queues are also modeled in addition to other components of the core energy. Moreover, DSENT (Sun et al. 2012) toolchain is deployed to obtain the network-on-chip per event energy numbers.

5.2.4 Evaluation Metrics. Each benchmark is run to completion, and the completion time and energy consumption is measured in the same regions as described for the TILE-Gx72 setup. The measured completion time is broken down into the following categories: *(1) Compute Stalls* is the time spent retiring instructions, waiting for functional unit (ALU, FPU, Multiplier, etc.), and the stall time due to mis-predicted branch instructions. *(2) Memory Stalls* is the stall time due to load/store queue capacity limits, fences, and waiting for load completion and L1 instruction cache misses. *(3) Communication Stalls* is the stall time due to explicit messaging instructions.

Dynamic energy is also measured and broken down into the following components: Core energy, L1 and L2 cache energy, Network energy, and DRAM energy.

5.3 Benchmarks and Inputs

Table 2 shows the six graph benchmarks from the CRONO (Ahmad et al. 2015) suite, and two machine-learning workloads, AlexNet and SqueezeNet. Note that both machine-learning benchmarks are realized using 4-way SIMD with 16-bit floating point instructions. These benchmarks are ported using Spin, Atomic and MC models. For all models, pthread library is used to spawn threads, and each thread is pinned to a physical core based on the thread ID. For the temporal MC model, two threads are utilized, and the threads are again pinned to their respective hardware

Table 2. Problem Sizes for Our Parallel Benchmarks

Benchmark	Input Dataset
Graph Analytics (CRONO (Ahmad et al. 2015))	
PAGERANK, TRIANGLE COUNTING COMMUNITY DETECTION, BFS CONNECTED-COMP, SSSP	California Road Network (Leskovec et al. 2009) Facebook (Leskovec and Sosivc 2016)
Machine Learning	
CNN-ALEXNET (KRIZHEVSKY ET AL. 2012) CNN-SQUEEZENET (IANDOLA ET AL. 2016)	ImageNet (Deng et al. 2009)

contexts. For evaluation, two real world graphs with uniform weights are chosen to explore input diversity in graph workloads, as summarized in Table 2. For machine-learning workloads an image from the ImageNet dataset is classified.

5.4 Configurations

(1) **Spin:** This is the baseline system that relies on spin locks to implement both fine and coarse-grain synchronization.
(2) **Atomic:** This model utilizes standalone atomic instructions to implement critical section code.
(3) ***MC:*** The default moving computation to data model with spatial distribution of *worker* and *services* threads implemented using in-hardware explicit messaging support (cf. Section 2.1.2).
(4) ***MC_shmem:*** Moving computation to data model with spatial distribution of *worker* and *services* threads implemented using shared memory cache coherence support (cf. Section 2.1.1).
(5) ***MC_tmp:*** Moving computation to data model with temporal distribution of *worker* and *services* threads (cf. Section 2.1.3).

6 EVALUATION

The core scaling results for the spatial MC model are first compared to Spin, Atomic, and MC_shmem models using the TILE-Gx72 machine and the RISC-V multicore simulator. The performance scaling and dynamic energy evaluations of the MC model with respect to Spin and Atomic models are also presented. Furthermore, temporal MC and MC_shmem models are compared against the spatial MC model.

6.1 Core Scaling on TILE-Gx72 and the Simulator

Figure 6 shows the average speedup of the spatial MC model over the Spin, Atomic and MC_shmem models as the core count increases. While the core count is varied from 8 to 64 in TILE-Gx72, it is scaled up from 8 to 1024 in the RISC-V simulator. There are some noteworthy differences between simulated architecture and the Tilera TILE-Gx72. First, each tile in Tilera utilizes a VLIW core that contains three parallel pipelines that do not have support for explicit floating point units. However, the simulator deploys in-order single-issue pipelined cores with support for 16-bit four-way SIMD instructions. Second, Tile-Gx72 uses data replication in L2 cache slices, whereas the cache lines are not replicated in the L2 cache slices of the simulated multicore. Third, Tilera utilizes atomic compare-and-swap for spin locks, whereas the simulation environment utilizes load-link and store-conditional instructions for spin-based synchronization. Overall, the performance trends are very similar between TILE-Gx72 and the simulated machine at 8–64 core counts.

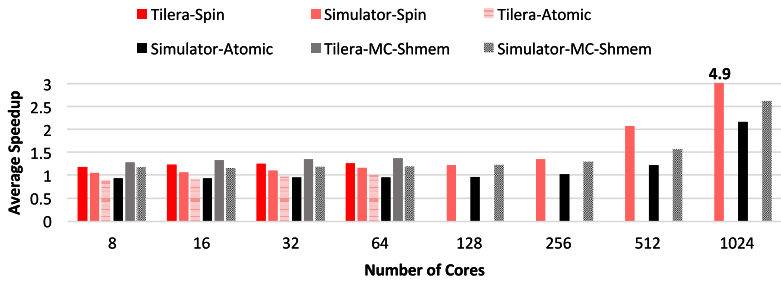

Fig. 6. Average speedup of spatial MC over Spin and Atomic models as the core counts increase.

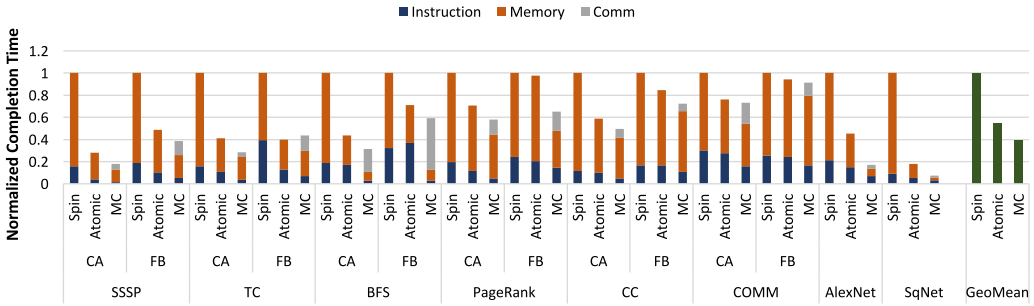

Fig. 7. Completion time results for Spin, Atomic, and MC at 512 cores; all normalized to Spin.

The relative performance of spatial MC model improves as the core count goes up in both TILE-Gx72 and the simulator. The MC model outperforms Spin at all core counts, since it does not suffer from instruction retries and cache line ping-pongs. It is also more efficient than the MC_shmem model at all core counts. As discussed in Section 2.1.1, the communication between worker and service threads is realized using a shared buffer per service thread. Whenever a message is being sent, the cache lines of the shared buffer bounce between the worker and service threads. Hence, even at smaller core counts, the performance of MC_shmem is worse than the MC model. The Atomic model fares well as it provides more concurrent execution of critical code sections, as well as other work. The MC model suffers from the challenge of load balancing work between the *worker* and *service* threads at lower core counts. However, the Atomic model utilizes all its cores but suffers from some cache line ping-pong overhead, which impacts performance as core counts increase. At smaller core counts, the Atomic model outperforms the MC model by more than 10%. The MC model closes the gap, and provides superior performance as core counts approach 64 and higher. The relative performance of the MC model significantly improves beyond 256 cores, and delivers significant advantages at both 512 and 1,024 cores. Although not shown here, the Atomic model delivers performance scaling for all benchmarks at 512 cores, but not at 1,024 cores. However, the spatial MC model consistently delivers performance scaling at both 512 and 1,024 cores. Detailed performance and energy consumption results are discussed for 512 cores, followed by various sensitivity analysis of the synchronization models.

6.2 Evaluation of 512-core Multicore

6.2.1 Performance Evaluation. Figure 7 illustrates the performance results of Spin, Atomic, and the spatial MC implementations of graph and machine-learning benchmarks at 512 cores. For graph workloads, results of both California Road Network (CA) and Facebook (FB) graphs are presented separately. Each data point is normalized to its Spin model's completion time. The

geometric mean shows that the MC model outperforms Spin by 60%, and the Atomic model by 27%.

Graph Workloads with Fine-grained Synchronization: SSSP, TC, and BFS benchmarks significantly benefit from the Atomic model as it removes locks, and uses an atomic instruction to implement each critical code section. As a result, instruction counts and memory stalls are drastically alleviated. However, the Atomic model does not remove cache line ping-pong, hence it is still limited in performance compared to the MC model. TC with Facebook graph is the only data point where Atomic slightly surpasses the MC model. There are two reasons that contribute to this performance loss for the MC model. TC does not involve any test to eliminate redundant critical section executions, as discussed in Section 4. So it requires atomic update for each neighbor. Therefore, it requires higher concurrency in the execution of its critical code section. In addition, Facebook is a sequential graph that does not have many common neighbors between its graph chunks. Hence, the shared data bouncing between cores is limited. Therefore, TC with Facebook graph benefits from higher concurrency. However, the CA graph contains more random connections, which leads to shared data bouncing in the Atomic model. Therefore, the MC model enhances the execution time as a result of pinning shared data, and overlapping the communication latency using non-blocking send instructions. In addition, the MC model's barrier synchronization eschews instruction retries and improves instruction stalls. However, it incurs communication stalls due to the explicit messaging instructions.

On the contrary to TC, the BFS algorithm guarantees that each critical section is executed at most once during its program execution. Therefore, higher concurrency in the execution of critical code sections is not as helpful for BFS. However, barrier synchronization becomes dominant at 512 cores. BFS is an iterative algorithm and it involves multiple thread barriers in each iteration. Hence, most of the communication stalls in completion time distribution are due to barriers. Consequently, employing an efficient MC barrier leads to the observed performance gain by reducing instruction count, and preventing the barrier variable ping-pong between cores.

The SSSP benchmark is in between BFS and TC. It involves a test to prevent redundant critical section executions, which reduces the number of critical sections in each iteration, as similar to BFS. As the algorithm converges, the number of active nodes goes down. However, unlike BFS, it does not guarantee only one critical section invocation per node. Hence, the parallelism needed to execute critical sections is not as small as BFS, but also not as much as TC. As a result, better performance is observed for both graphs under the MC model. The MC model also does not prevent shared data bouncing in SSSP, as discussed in Section 4.2. Hence, the main advantage of MC over atomic instructions for SSSP is latency hiding using the non-blocking explicit message requests. In addition, similar to BFS, SSSP also involves multiple barriers per iteration. Therefore, using MC barrier helps improve performance by removing instruction retries, and expensive shared variable bouncing between cores.

As previously discussed, the MC model takes advantage of latency hiding using the non-blocking send instructions. To better understand this performance enhancement, the MC model is also implemented with blocking *sendr* instruction (MC-reply) to prevent more than one in-flight request per core. This averts overlapping communication stalls with other useful work in the *worker threads*. Figure 8 shows the performance comparison of the default spatial MC with Atomic and the MC-reply models. As seen, when implemented with MC-reply, both SSSP and TC lose their performance gains. The outcome is more severe in TC as most of its work is shared work. The performance gets worse, because concurrency is limited with MC as compared to the Atomic model. However, BFS does not show any change in completion time. This suggests that BFS does not benefit from non-blocking send messages, because the *workers* temporally send requests after significant local computations.

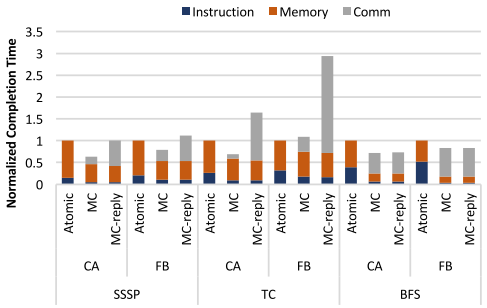

Fig. 8. Performance comparison of MC-reply with default MC and Atomic at 512 cores; all normalized to Atomic.

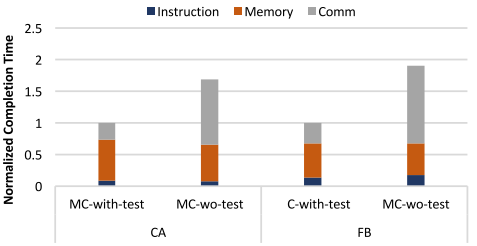

Fig. 9. Performance comparison of SSSP under MC with and without test before critical section request at 512 cores; all normalized to MC.

Graph Workloads with Coarse-grained Synchronization: For PAGERANK, CC and COMM, the Atomic model reduces instruction count by replacing the lock in the barrier implementation with an atomic fetch-and-add instruction. The decrease in instruction count depends on whether the barrier variable is contended or not. If there is load imbalance and threads reach the barrier at different timestamps, then utilizing the atomic instruction does not help much. It makes updating the barrier variable more efficient; however, it still requires spinning until all threads arrive at the barrier. However, if threads participate in the barrier at similar timestamps, the shared variable gets contended at 512 cores, which leads to more costly barrier implementation with Spin. Atomic eliminates this costly lock acquisition but the shared variable still bounces between cores. Hence, the penalty of contention on barrier becomes a noticeable portion of the completion time even though these benchmarks are highly parallel and the input graphs are sufficiently large in size. However, the MC model eliminates instruction retries and expensive shared barrier variable ping-pongs. The spinning cost is replaced with more efficient explicit communication that leads to enhanced performance compared to both Spin and Atomic models.

Machine Learning Workloads: The MC models yields significant performance improvements for the two machine-learning workloads. For ALEXNET, it utilizes the fine-grain parallelization strategy explained in Section 4.3, while SQUEEZENET is realized using coarse-grain parallelization. As load imbalance is very small in both workloads, the threads reach barrier synchronizations at similar timestamps. Therefore, the barriers are contended. Consequently, as a result of more efficient barrier implementation, the Atomic model improves performance over the Spin model. However, the Atomic model also suffers from bouncing the shared barrier variable between cores. The MC model improves performance by removing the cache line ping-pongs. SQUEEZENET contains less work between barriers, and the number of barriers are also more than ALEXNET. Therefore, it benefits more from explicit messaging based barrier.

The Role of Hardware Cache Coherence: The above discussions show that core-to-core direct communication is an effective approach to mitigate bottlenecks of the shared memory based synchronization. However, hardware cache coherence is still needed to effectively move data at fine (cache line) granularity between cores. For example, SSSP contains a test before sending critical section invocations. The test ensures that no redundant messages are being sent, hence this results in a work efficient parallel implementation. The *workers* read the shared distance array to perform a test to determine if critical code section execution should be invoked or not. As the cache coherence protocol thrives on exploiting locality in read data sharing, the overhead of performing the test is more work efficient even though it results in some coherency traffic. Under all synchronization models, the redundant critical code section invocations are eliminated, which results in

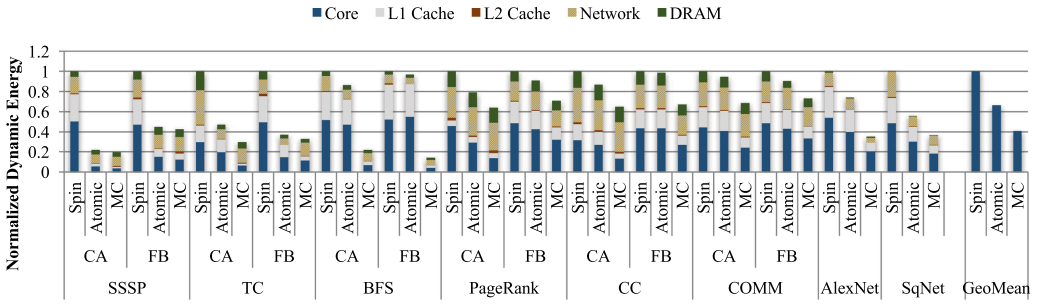

Fig. 10. Dynamic energy results for Spin, Atomic, and MC at 512 cores; all normalized to Spin.

superior performance. To evaluate this hypothesis, the SSSP benchmark is also implemented without the test, and its performance is compared against the spatial MC implementation with test. Figure 9 illustrates this result. By eliminating the test, memory stalls slightly go down due to reduced cache coherence traffic. However, the communication stalls drastically increase due to the elevated serialization at *service threads*, since the workers send a lot more critical section requests.

6.2.2 Dynamic Energy Evaluation. Figure 10 illustrates the dynamic energy results at 512 cores. As seen, the spatial MC model provides a geometric mean of 60% and 39% better dynamic energy consumption as compared to the Spin and Atomic models, respectively.

The dynamic energy trends for SSSP, TC, and BFS are similar to their respective completion time results. In general, reductions in instruction and memory stalls also show up in the dynamic energy. The Atomic model reduces core and L1 cache energy by removing synchronization overheads due to instruction retries. Furthermore, the MC model notably reduces both components due to reasons discussed in Section 6.2.1. Moreover, the network energy drastically reduces from the Spin to Atomic model, since lock acquisition related network messages are removed. However, the MC model increases network energy compared to Atomic for SSSP and TC. This is due to the fact that the MC model adds critical section request messages, whereas Atomic only involves network activities related to the atomic operation. Other notable observation is that dynamic energy benefit for BFS is way more than its performance gain. This is due to the fact that the biggest portion of the completion time breakdown is communication stalls, which do not contribute to the dynamic energy in the MC model as the core stays idle during this stall time.

The figure also shows the results for graph workloads with coarse-grain synchronization. As seen, dynamic energy again follows very similar trends with performance. As mentioned previously, one important advantage of the MC model is that since it just utilizes blocking *sendr* instruction to implement barrier synchronization, the communication stall seen in the completion time does not show up in dynamic energy. However, both Spin and Atomic models need to execute some instructions and perform memory accesses while waiting on the barrier. This can be clearly observed in the COMM benchmark. Due to load imbalance, both Atomic and Spin barriers need to execute many instructions to wait for the other threads, which increases both memory and core energy. However, the MC model stalls the pipeline and does not execute any instructions or make memory accesses. Similar discussions are also applicable to the two machine-learning benchmarks.

6.3 Performance Scaling as On-Chip Core Counts Varied from 64 to 1024

Figures 11, 12, and 13 show the performance scaling results of all benchmarks as the number of cores per multicore chip are increased from 64 to 1,024. The total on-chip cache capacity is kept nearly constant at ~22MB, that is to say that per tile cache sizes are scaled down as more cores

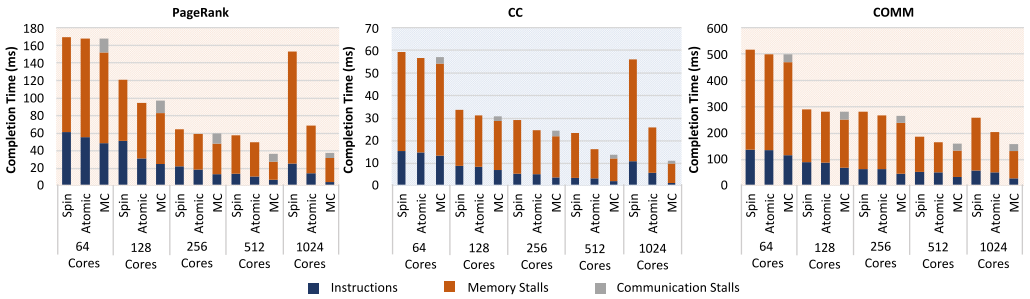

Fig. 11. Core scaling results for Spin, Atomic, and MC implementations of PAGERANK, CC, and COMM.

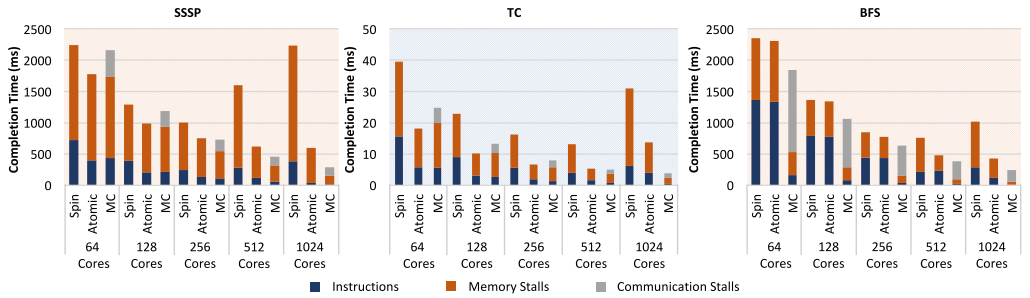

Fig. 12. Core scaling results for Spin, Atomic, and MC implementations of SSSP, TC, and BFS.

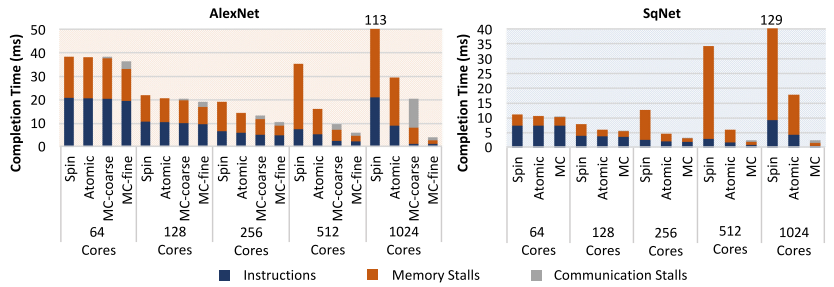

Fig. 13. Core scaling results for Spin, Atomic, and MC implementations of ALEXNET and SQUEEZENET.

are integrated on-chip. The results of all graph benchmarks are the average of California Road Network and Facebook graphs, and reported as raw completion times in each figure.

Figure 11 shows that graph workloads with coarse-grain synchronization scale up to 512 cores for all three communication models. The main reason is that the communication is not the bottleneck, since much computation is performed locally in parallel. However, barrier synchronization overhead shows up in completion time beyond 512 cores, and prevents the Spin model to scale. The Atomic model scales better than the Spin model, since it utilizes atomic fetch-and-add instruction for its barrier variable update. However, it slows down drastically as core counts are increased from 512 to 1024. However, the spatial MC model achieves superior performance scaling compared to both Spin and Atomic models.

Figure 12 shows the core scaling results for the benchmarks with fine-grain synchronization. When the core count rises, the overheads boost exponentially for the Spin model after 256 cores in SSSP. Both the number of instructions and memory stalls blow up due to instruction retries and expensive cache line ping-pongs. Better completion time is accomplished with the Atomic model

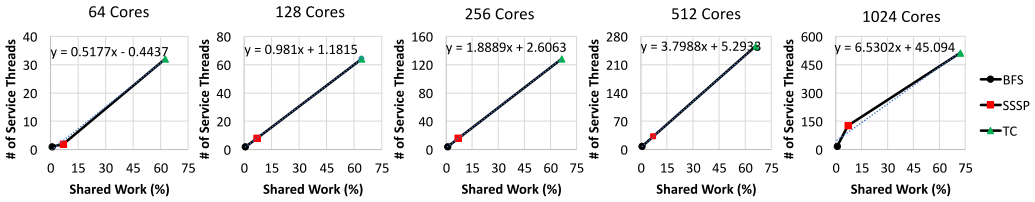

Fig. 14. Correlation of service core count with shared work for SSSP, TC and BFS at different core counts.

by employing more efficient barrier and lock-free data structures. However, its performance also slows down after 512 cores due to the increased sharing and enlarged network size, which make atomic updates more costly. The MC model helps SSSP scale to 1,024 cores. BFS also follows similar trends as observed for SSSP. The Spin version stops scaling beyond 256 cores, and Atomic achieves performance scaling up to 512 cores. However, the MC model provides superior performance up to 1024 cores. Unlike SSSP, barrier synchronization is the biggest bottleneck in BFS, since the locks are not contended (cf. Section 6.2.1). Subsequently, providing faster barrier with the MC model pushes scaling to thousand cores.

Figure 13 shows the scaling of both machine-learning benchmarks, ALEXNET and SQUEEZENET. The four-way SIMD with 16-bit floating point capability per core significantly reduces both instruction counts and memory stalls for these benchamrks. Hence, synchronization at the end of each layer becomes important at higher core counts. As SQUEEZENET contains more layers (and barriers), it experiences performance degradation for the Spin model at 256 cores and higher. The Atomic model also stops scaling at 512 cores. The MC model, on the other hand, scales to 1,024 cores for SQUEEZENET with more efficient barrier synchronization. Similarly, the MC model also helps achieve superior performance for ALEXNET up to 1,024 core. However, as discussed in Section 4.3, it has two implementations for the spatial MC model. One is naive implementation (MC-coarse) that only replaces the barrier synchronization. This implementation does not scale to 1,024 cores as it suffers from load imbalance due to limited concurrency. The fine-grain strategy exposes more concurrency without sacrificing data reuse. Hence, MC-fine provides performance improvements up to 1,024 cores.

6.4 Determining Service Thread Count in the Spatial MC Model

Despite its notable performance achievements, the spatial MC model has a challenge to tune the right number of *worker* and *service threads* for fine-grain synchronization. This is important, because it changes from workload to workload, and using the same *service thread* count for two different workloads may result in significant performance loss. For example, sweep study for SSSP at 64 cores reveals that it requires only 2 *service threads*. If the same service thread count is deployed for TC, then it results in 3× worse performance as compared to Spin. Therefore, TC also requires a separate search. As the number of cores goes up, the search space for the best performing ratio also increases. Hence, it gets more time consuming. Therefore, we propose a profiling driven heuristic to determine the near-optimal service thread count.

As discussed in Section 2.1.3, the number of *service threads* are expected to correlate with the average amount of time spent in the critical code sections. The Spin version of benchmarks with fine-grain synchronization is profiled to obtain the percentage shared work each thread performs at 64, 128, 256, 512, and 1,024 cores. Also, for each core count, a sweep study is conducted to obtain the best performing *service thread* count. Figure 14 shows the shared work against the best performing number of *service threads*. As seen, there is a linear correlation between the best performing *service thread* count and the shared work for all benchmarks. This suggests that by profiling the Spin version, one can easily determine the required number of *service threads*. For

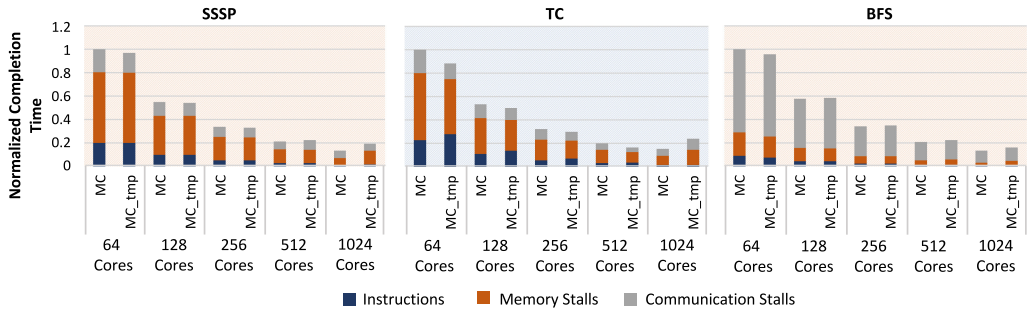

Fig. 15. Normalized core scaling results of MC, and MC_tmp; all normalized to MC at 64 cores.

example, at 512 cores, SSSP has 7% shared work, which results in 35 *service threads*. This number is very close to the optimal number of 32 acquired from the sweep study. As discussed in Section 2.1.3, the *worker* to *service thread* ratio is bound to at most 50%. Therefore, in some cases such as TC where the shared work is more than 50% of the total completion time, no more than half the cores are assigned as *service threads*. At 512 cores, picking the right number of *service threads* with this heuristic on average causes only 5% performance loss compared to the near optimal performance obtained with exhaustive sweep study.

6.5 Spatial versus the Temporal MC Model

This section evaluates the spatial MC model against the MC_tmp model that eliminates the need for tuning the service thread count. It does so by utilizing temporal mapping of *service* and *worker threads* in the same core, as explained in Section 2.1.3. Figure 15 demonstrates the results of spatial MC and the temporal MC model for three fine-grain graph benchmarks. The presented results are the average of California Road Network and Facebook graphs, and the results are normalized to the spatial MC model at 64 cores.

MC_tmp improves performance over the spatial MC model at 64 core count. At lower core counts, finding the optimal *service thread* count is easier, however load balancing the *service* and *worker threads* is difficult. If the thread count assigned for critical section execution is higher, then it hurts performance by taking away parallelism from the *worker threads*. However, if it is smaller, it may create serialization at the *service threads*. The temporal approach makes load balancing easier as each core is both a *worker* and a *service thread*. Using same number of *service* and *worker threads* with the MC_tmp model helps improve concurrency on both algorithm work, as well as the critical section execution. However, as the number of cores increases, the benefits of the MC_tmp model diminish. At higher core counts, the number of cores are abundant, hence the work per thread is smaller. Therefore, it is easier to spare some of the cores as *service thread* with the spatial approach. In addition, 2× more threads participate in barrier synchronization with MC_tmp, and this enlarges the barrier communication overheads. Moreover, assigning two different tasks in the same core stress the private cache capacity, which results in higher memory stalls. Consequently, employing the MC_tmp model at 512 cores and beyond leads to degradation in performance. Overall, these results suggest that even though MC_tmp does not require tuning the *service thread* count, it does not scale as well as the spatial MC model at high core counts. In addition, it also requires two hardware contexts per core, and context switching policy logic that takes explicit messaging into account.

6.6 Evaluation of MC against MC_shmem

This section discusses the shared memory version of the moving computation to data model. As discussed in Section 2.1.1, the spatial MC model is implemented using the shared memory cache

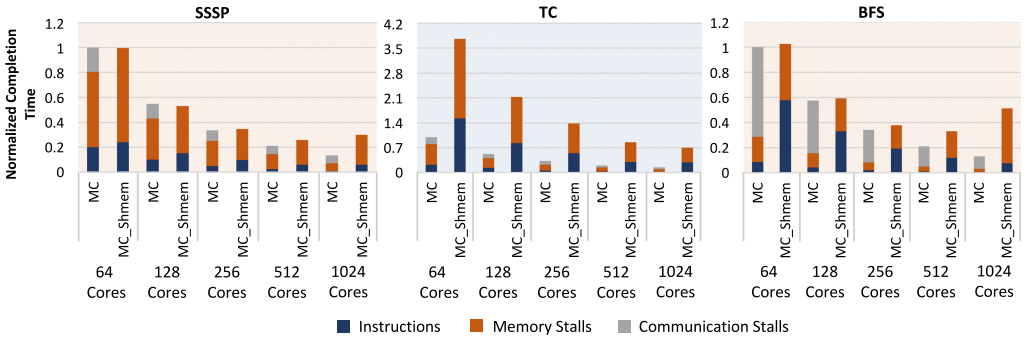

Fig. 16. Normalized core scaling results of MC, and MC_shmem; all normalized to MC at 64 cores.

coherence protocol without in-hardware explicit messaging support. Figure 16 shows the normalized completion time of spatial MC and the MC_shmem models for the SSSP, TC and BFS benchmarks at different core counts. The presented results are the average of California Road Network and Facebook graphs, and the results are normalized to spatial MC at 64 cores.

The figure shows that the performance scaling of BFS follows almost the same trend with the Atomic model in Figure 12. As discussed in Section 6.2.1, BFS is not contended, hence the implementation of the critical section does not make much difference in performance. Most benefits come from the explicit messaging based barrier implementation. Therefore, at higher core counts, the performance of the MC_shmem model degrades due to ping-pong of the shared barrier variable. In the case of SSSP, MC_shmem provides similar performance as the MC model up to 256 cores. At 512 cores, the performance of MC_shmem is 1.22× worse than the MC model with explicit messaging. This is better than the completion time of the Atomic model at the same core count (1.35× worse than MC), because MC_shmem benefits from non-blocking critical section requests. However, due to ping-ponging of the shared buffer between worker and service threads, the performance benefit is still limited. In addition, similar to Atomic, the barrier synchronization also becomes a limiting factor at these core counts. At 1,024 cores, the ping-ponging affect becomes worse, hence more than 2× performance difference is observed. TC is the only workload in which MC_shmem is always worse than the MC model at all core counts. Unlike other two workloads, the MC model version of TC prevents ping-ponging of shared data, because the shared data is not accessed anywhere outside the service thread task. MC_shmem also pins the shared data in the service thread and enables non-blocking communication. However, it adds constant ping-ponging of the shared buffer to enable communication between worker and service threads. This happens even when there is no contention on the shared data. In the case of Atomic model, if there is no contention on shared data, there is no bouncing, hence it can benefit from elevated concurrency on the critical section. As a result, it provides better performance. However, as a result of constant ping-ponging of the shared buffer, the performance of the MC_shmem model is always worse than both Atomic and MC models. The results show that on average the MC model with explicit messaging is 2.3× faster than the equivalent shared memory implementation, MC_shmem. Therefore, it suggests that in-hardware explicit messaging support is required to enable efficient implementation of the MC model.

7 RELATED WORK

Parallel architectures that combine shared memory paradigm and explicit messaging have been explored by researchers. Alewife and ActiveMsg (Kubiatowicz and Agarwal 1993; von Eicken et al. 1992) have integrated the idea of message passing into shared memory multiprocessors to mitigate

the bottleneck of inter-processor communication. More recently, Tesseract (Ahn et al. n.d.) utilizes message passing only architecture, and demonstrates the benefits of such communication style. It utilizes a near memory approach in which a high number of simple cores are located closer to a 3D stacked memory. However, as it does not support shared memory paradigm, it differs from the investigated architecture. ADM (Sanchez et al. 2010) supports both shared memory coherence and hardware messaging, and tries to accelerate task scheduling by employing core-to-core messaging. However, it falls short of exploring it for general purpose thread synchronization.

The commercial Tilera (Wentzlaff et al. 2007) architecture implemented a multicore processor that supports both cache coherence and hardware messaging using a UDN. However, its messaging capability is not fully explored with novel communication models. Barrier synchronization is investigated using the low-latency hardware messaging by TSHMEM (Lam et al. 2013) work. However, it does not explore these capabilities for real parallel workloads to accelerate synchronization. Moreover, Tilera does not offer products with higher core counts, which as shown to benefit most from the proposed spatial MC model.

The idea of MC style critical section execution is investigated in RCL (Lozi et al. 2012). It utilizes a software only approach without any hardware support. The critical section requests are placed into a shared buffer, then server thread executes them as remote procedure calls. They only target 48 cores system and the investigated applications contain only a single lock. As it utilizes a shared buffer for the critical section requests, it is expected to limit performance at higher core counts. It is also not clear how the proposed model works for applications with fine-grain critical sections. In this article, a similar implementation of MC termed as MC_shmem is presented to illustrate the shortcomings of such approach.

Similar to RCL, ACS (Suleman et al. 2009) explores the critical section migration, however, with hardware support. ACS also ships the critical section block to a dedicated core. However, its target architecture is a small-scale heterogeneous multicore, where it contains several small cores and a large core. It ships the critical section execution to the large core. However, the proposed MC model is implemented targeting symmetric multicores at 1,000-core scale. In both approaches, the serialization of the critical section plays a significant role in performance. ACS tries to solve the serialization problem at the dedicated core by utilizing two approaches. The first one is to use simultaneous multithreading to enable multiple threads to execute critical sections concurrently. The second method is to utilize a serialization detection scheme to determine whether or not to offload the critical section execution to the large core. Our work addresses the serialization problem by assigning multiple cores to concurrently perform the execution of critical code sections. For this purpose a profiling based heuristic method is proposed to determine the number of service cores that are dedicated for managing work on shared data. Moreover, our work utilizes emerging workload domains from graphs and machine learning to evaluate the proposed spatial MC model.

Recently, Active Messages (AM) (Harting and Dally 2014) also showed the usage of hardware message passing on top of a shared memory architecture. However, it only explores a model similar to the temporal MC model in which a separate hardware context is used as a message handler. Our work investigates both spatial and temporal approaches and shows better scaling with the spatial MC model as compared to the temporal model. In addition, AM has not been evaluated against efficient atomic instruction based synchronization in real workloads.

HAQu (Tiwari et al. 2011) and CAF (Wang et al. 2016) demonstrate that fine-grain synchronization can be accelerated in multicores using hardware queues. HAQu accelerates queues in the program's address space with the extension of new instructions. However, CAF utilizes new hardware extensions to the on-chip network to mitigate queuing bottlenecks. Both approaches investigate similar models to the proposed MC model using the architectural extensions. However,

the application domains differ, and the proposed spatial MC model is shown to accelerate thread synchronization at 1,000-core scale.

8 CONCLUSION

This article proposes a novel moving compute to data (MC) model for accelerating synchronization on a 1,000-core scale single-chip multicore processor. The proposed model accelerates synchronization by executing critical code sections at dedicated cores using low-latency and non-blocking core-to-core explicit messaging hardware. The spatial MC model is evaluated against atomic instructions and the traditional spin-lock based synchronization primitives for graph analytics and machine-learning benchmarks. Variants of the MC model are also evaluated, such as the MC_tmp and the MC_shmem models. The experimental results show that the spatial MC model scales performance up to 1,000 cores. It offers an average 60% improvement over lock based synchronization, and 27% better performance over atomic instruction based synchronization at 512 cores. Furthermore, the MC model achieves an average of 39% efficiency on dynamic energy as compared to the atomic instruction based synchronization model.

REFERENCES

M. Ahmad, F. Hijaz, Qingchuan Shi, and O. Khan. 2015. CRONO: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'15)*. 44–55. DOI: https://doi.org/10.1109/IISWC.2015.11

J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. [n.d.]. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM.

R. Bayer and M. Schkolnick. 1988. Concurrency of Operations on B-trees. In *Readings in Database Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 129–139.

William J. Dally and Brian Towles. 2004. *Principles and Practices of Interconnection Networks*.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 248–255.

H. Dogan, M. Ahmad, J. Joao, and O. Khan. 2018. Accelerating synchronization in graph analytics using moving compute to data model on Tilera TILE-Gx72. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'18)*.

H. Dogan, F. Hijaz, M. Ahmad, B. Kahne, P. Wilson, and O. Khan. 2017. Accelerating graph and machine-learning workloads using a shared memory multicore architecture with auxiliary support for in-hardware explicit messaging. In *Proceedings of the Annual International Parallel and Distributed Processing Symposium (IPDPS'17)*.

William Gropp. 2002. MPICH2: A new start for MPI implementations. In *Proceedings of the European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 7–7.

R. Curtis Harting and William J. Dally. 2015. On-chip active messages for speed, scalability, and efficiency. *IEEE Transactions on Parallel and Distributed Systems* 26, 2 (2015), 507–515.

W. Huang, K. Rajamani, M. R. Stan, and K. Skadron. 2011. Scaling with design constraints: Predicting the future of big chips. *IEEE Micro* 31, 4 (2011), 16–29.

Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size. *arXiv:1602.07360 preprint* (2016).

Brian Kahne. 2013. FreescaleADL: An Industrial-Strength Architectural Description Language For Programmable Cores. Retrieved from http://opensource.freescale.com/fsl-oss-projects/.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*.

John Kubiatowicz and Anant Agarwal. 1993. The anatomy of a message in the alewife multiprocessor. In *Proceedings of the Industrial Control Systems Cyber Security Conference (ICS'93)*.

G. Kurian, J. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. Kimerling, and A. Agarwal. 2010. ATAC: A 1000-core cache-coherent processor with on-chip optical network. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*.

B. C. Lam, A. D. George, and H. Lam. 2013. TSHMEM: Shared-memory parallel computing on tilera many-core processors. In *Proceedings of the 2013 IEEE International Symposium on Parallel Distributed Processing*. 325–334. DOI: https://doi.org/10.1109/IPDPSW.2013.154

J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

Jure Leskovec and Rok Sosivc. 2016. SNAP: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.* 8, 1 (2016), 1.

Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'42)*. ACM, New York, NY, 469–480. DOI: https://doi.org/10.1145/1669112.1669172

Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia L. Lawall, Gilles Muller, et al. 2012. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the USENIX Annual Technical Conference*. 65–76.

J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the IEEE Symposium on High Performance Computer Architecture (HPCA'10)*.

Samuel K. Moore. 2016. Breaking the Multicore Bottleneck. Retrieved from https://spectrum.ieee.org/semiconductors/processors/breaking-the-multicore-bottleneck.

Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *ACM SIGOPS Operating Systems Review*, Vol. 42. ACM, 247–260.

Sunghyun Park, T. Krishna, C. Chen, B. Daya, A. Chandrakasan, and Li-Shiuan Peh. 2012. Approaching the theoretical limits of a mesh NoC with a 16-node chip prototype in 45nm SOI. In *Proceedings of the Annual Design Automation Conference (DAC'12)*.

D. Sanchez, R. M. Yoo, and C. Kozyrakis. 2010. Flexible architectural support for fine-grain scheduling. In *Proceedings of the 15th Annual Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, 311–322.

M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. ACM.

C. Sun, C. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L. Peh, and V. Stojanovic. 2012. DSENT-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proceedings of the IEEE/ACM International Symposium on Networks on Chip (NoCS'12)*. IEEE, 201–210.

D. Tiwari, J. Tuck, Solihin Y, and S. Lee. 2011. HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*.

T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. 1992. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the International Symposium on Computer Architecture (ISCA'92)*. 11.

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. 1997. Baring it all to software: Raw machines. *IEEE Computer* 30, 9 (1997), 86–93.

Y. Wang, R. Wang, A. Herdrich, J. Tsai, and Y. Solihin. 2016. CAF: Core to core communication acceleration framework. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT'16)*. ACM.

Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovi. 2014. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0*. Technical Report. Department of Electrical Ingineering and Computer Sciences, University of California–Berkeley.

D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. Brown III, and A. Agarwal. 2007. On-chip interconnection architecture of the tile processor. *IEEE Micro* 27, 5 (Sep. 2007), 15–31.

Received July 2018; revised November 2018; accepted December 2018