Accelerating Synchronization on Futuristic 1000-cores Multicore Processor with Moving Compute to Data Model

Halit Dogan
University of Connecticut - Storrs, halitdoganeem@gmail.com

Accelerating Synchronization on Futuristic 1000-cores Multicore Processor with Moving Compute to Data Model

Halit Dogan, Ph.D.

University of Connecticut, 2018

ABSTRACT

Single chip multicore processors are now prevalent and processors with hundreds of cores are being proposed and explored by both academia and industry. Shared memory cache coherence is the state–of–the–art technology for these processors to enable synchronization and communication between cores. However, since the synchronization of cores on shared data using hardware cache coherence suffers from instruction retries and cache line ping-pong overheads, it prevents performance scaling as core counts increase on a chip.

This thesis proposes to utilize a novel moving computation to data model (MC) to overcome this synchronization bottleneck in a 1000-cores scale shared memory multicore processor. The proposed MC model pins shared data to dedicated cores called service cores. The execution of critical code sections is explicitly requested from worker cores to be performed at the service cores. In this way, the cache line bouncing between cores is prevented, hence data locality optimization is enabled. The proposed MC model utilizes auxiliary in-hardware explicit messaging for the critical section requests to enable efficient fine-grained blocking and non-blocking communication between communicating cores. To show the effectiveness of the proposed model, workloads with wide range of synchronization requirements from graph analytics, machine learning and database domains are implemented. The proposed model is then prototyped and exhaustively evaluated on a 72 core machine, *Tilera*® *Tile–Gx72*™ multicore platform, as it incorporates in–hardware core–to–core messaging as an auxiliary capability to the shared memory cache coherence paradigm. Since the *Tile-Gx72*™ machine includes only 72 cores, it is deployed for evaluation at 8 to 64 core count scale. For further analysis at higher core count, a simulated RISC–V multicore environment is built and utilized, and the performance and dynamic energy scaling advantages of the MC model is evaluated against various baseline synchronization models up to 1024 cores.

Accelerating Synchronization on Futuristic 1000-cores Multicore Processor with Moving Compute to Data Model

Halit Dogan

B.E., Inonu University, Malatya, Turkey, 2008

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2018

APPROVAL PAGE

Doctor of Philosophy Dissertation

Accelerating Synchronization on Futuristic 1000-cores Multicore Processor with Moving Compute to Data Model

Presented by

Halit Dogan, B.E.

Major Advisor	Omer Khan
Associate Advisor	John Chandy
Associate Advisor	Marten Van Dijk

University of Connecticut

2018

ACKNOWLEDGMENTS

There are many people whom I have sincere gratitude for their contribution to my time during my PhD studies. First, I am sincerely thankful to my advisor, Prof. Omer Khan. He was there with me throughout my PhD studies whenever I need his support, guidance, and encouragement. Even during my personal struggles, he supported me with utmost sincerity. I am indebted to him for the knowledge and invaluable experiences I have gained during my PhD.

I would like to thank my former and current lab mates, Farrukh Hijaz, Hamza Omar, Masab Ahmad, and Qingchuan Shi for their continued support, and great friendship. Having them around made my PhD much more delightful experience. Special thanks should go to Farrukh Hijaz as I owe him great deal during my first year in the lab. He mentored me and initiated my PhD research.

I am also grateful to Republic of Turkey, Ministry of National Education for providing me the graduate fellowship to pursue my PhD. I would also like to acknowledge the funding received for supporting my PhD research from US National Science Foundation, Semiconductor Research Corporation, and NXP Semiconductors. I would specially like to express my appreciation to Brian Kahne of NXP Semiconductors for his continuous support on the development of the multicore simulator I employed in my research. I would also like to thank José Joao of Arm Research, and Christopher Hughes of Intel Corporation for their valuable discussions and feedback towards the evolution of my research.

I also feel obliged to express my thanks to my friends outside of my lab. During my PhD, I was surrounded by an amazing group of people. Specially, the moral and spiritual support of my friends in Bloomfield Muslim Community Center was invaluable for me.

I also would like to thank my unconditionally loving and supporting family. Knowing that my parents, Huseyin and Nejla Dogan, always keep me in their prayers gave me great strength when I was under a lot of stress. I wouldn't have been able to be where I am without their support and encouragements.

Last but not least, I am grateful to my beloved wife, Esra, for being beside me, with a great patience and support during the most stressful times. When I was almost always working, and busy, she was the one who took care of everything else in my life, including our two unstoppable sons. My older son, Kerem (4), is almost at the same age with my PhD, and younger son, Mirza, joined our family in the last year of the PhD. Kerem grew up seeing me staring at the black screen of my laptop. I believe he should be pretty familiar with my research now. I thank both my sons for being patient with their daddy when he was busy with his PhD study.

Contents

List of Figures

Page

List of Tables

Chapter 1

Introduction

With the proliferation of shared memory processors with hundreds of cores on chip [1] [2], both fine and coarse–grain 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 [3] [4] [5] [6] [7]. At small core counts, spin-based synchronization primitives are efficient. However, the overheads of such primitives exponentially aggravate as the core count increases [8] [9]. This primarily happens due to expensive cache line ping–pong between cores as a result of the increased latency of the on-chip network. 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–pongs between cores, and the expensive coherence traffic leads to performance scaling challenges at higher core counts. Moreover, the type of atomic operations are limited, hence they may not be generalized for arbitrary critical code sections. Therefore, this thesis proposes to utilize a novel moving computation to data (MC) model to overcome the synchronization challenges for futuristic 1000–cores scale multicores.

The key idea is to keep shared memory cache coherence and accelerate thread synchronization using the MC model. In the MC model, shared data is 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 shared data, as it is pinned to a dedicated thread. For example, the critical sections can be offloaded to the service thread to accelerate fine–grained synchronization, and barrier type of coarse–grained synchronizations can be implemented in a more efficient way. The MC model can be implemented by utilizing shared memory cache coherence in which a software based shared buffer is deployed to communicate messages between workers and the service thread. RCL [10] 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 the same synchronization challenges at higher core counts. Therefore, in this thesis, *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.*

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 critical code section executions with other useful work. In addition, as compared to the lock based critical sections, it gets rid of the lock acquisition overheads, such as instruction retries and the mutex variable ping–ponging, by completely eliminating the locks. Utilizing a single *service thread* may become performance bottleneck due to serialization of multiple requests. Therefore, to exploit concurrency in critical section execution requests, multiple service cores are assigned as *service threads* and the shared data is divided among them. The remaining worker threads exploit concurrency in the underlying algorithmic code, and direct the requests to the corresponding service threads. This thesis 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 [11]. 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 outperforms the spatial model at lower core counts, however as the number of cores approaches 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 core counts can be challenging as it is workload dependent. Therefore, this thesis explores a novel heuristic to determine load balanced mapping of the worker and service threads for the spatial MC model. 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 shared work, while the worker cores optimally exploit concurrency in the remaining algorithmic work for a given workload.

The contributions of this thesis 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 [12] [13]. These key aspects deliver high performance scaling for both fine and coarse grain synchronization in parallelized workloads from the machine learning, graph processing and database domains. Moreover, 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.

2. The MC model is prototyped and evaluated extensively on *Tilera*® *Tile-Gx72*™ multicore machine [14] as it has the largest core count among commercially available processors with in–hardware explicit messaging. A scaling study up to 64 cores is conducted, and enhanced performance benefits of the MC model over the traditional shared memory synchronization are shown as the core count increases.

3. Since Tilera machine contains only 72 cores, to explore 1000-cores scale multicore system, a RISC-V based multicore simulation environment is built and utilized to characterize the spatial MC model, and compare performance and energy consumption over both spin-lock and atomic instruction based synchronization models [15]. In addition, the spatial MC model is also evaluated against the temporal MC model. Furthermore, a software shared buffer based moving compute to data model is implemented, and the MC model with in–hardware explicit messaging is evaluated against it. The spatial MC model with hardware support is shown to enable superior performance scaling up to 1024 cores.

The rest of the thesis is organized as follows. Chapter 2 talks about the related work. Thread synchronization in traditional shared memory paradigm and the proposed MC model are discussed in Chapter 3. Architectural extensions required to enable efficient implementation of the MC model is described in Chapter 4. Chapter 5 discusses the programming model and how a given shared memory based workload is implemented using the MC model. The evaluation of the MC model on *Tilera*® *Tile-Gx72*™ multicore platform is discussed in Chapter 6. The evaluation of the MC model on 1000-cores scale multicore simulator is presented in Chapter 7. Finally, Chapter 8 concludes the thesis.

Chapter 2

Related Work

Parallel architectures that combine shared memory paradigm and explicit messaging have been explored by researchers. Alewife and ActiveMsg [16, 17] have integrated the idea of message passing into shared memory multiprocessors to mitigate the bottleneck of inter-processor communication.

More recently, Tesseract [18] 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 [19] supports both shared memory coherence and hardware messaging, and tries to accelerate task scheduling by employing the core–to–core messaging. However, it falls short on exploring it for general purpose synchronization.

The commercial Tilera [20] architecture implemented a multicore processor which supports both cache coherence and hardware messaging using a User Define Network (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 [21] work. However, it does not these capabilities for real parallel workloads to accelerate synchronization. Moreover, Tilera falls short to analyze the benefits of explicit messaging based communication model at higher core counts.

The idea of MC style critical section execution is investigated in RCL [10]. 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 single lock. As it utilizes a shared buffer for the critical section requests, it is expected to limit the performance at higher core counts. It is also not clear how the proposed model would work for the applications with fine–grained critical sections. In this thesis, a similar implementation of MC termed as MC_shmem is presented to illustrate the shortcomings of such approaches and the default MC model is evaluated against it.

Similar to RCL, ACS [22] explores the critical section migration, however, with hardware support. Similar to proposed MC model, 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. On the other hand, the proposed MC model is implemented targeting symmetric multicores at 1000–cores 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. This 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, I utilize emerging workload domains from graphs, and machine learning to evaluate the proposed spatial MC model.

Active Messages (AM) [11] also explores 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. This work investigates both spatial and temporal approaches and shows better scaling with the spatial MC model as compared to temporal. In addition, AM has not been evaluated against efficient atomic instruction based synchronization in real workloads.

HAQu [23] and CAF [24] 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. On the other hand, 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 between their work and the model presented in this thesis. Moreover, this thesis studies the benefits of the MC model on 1000-cores scale multicores, and compares to both spin-based shared memory synchronization as well as efficient atomic instruction based synchronization.

Chapter 3

Thread Synchronization Models

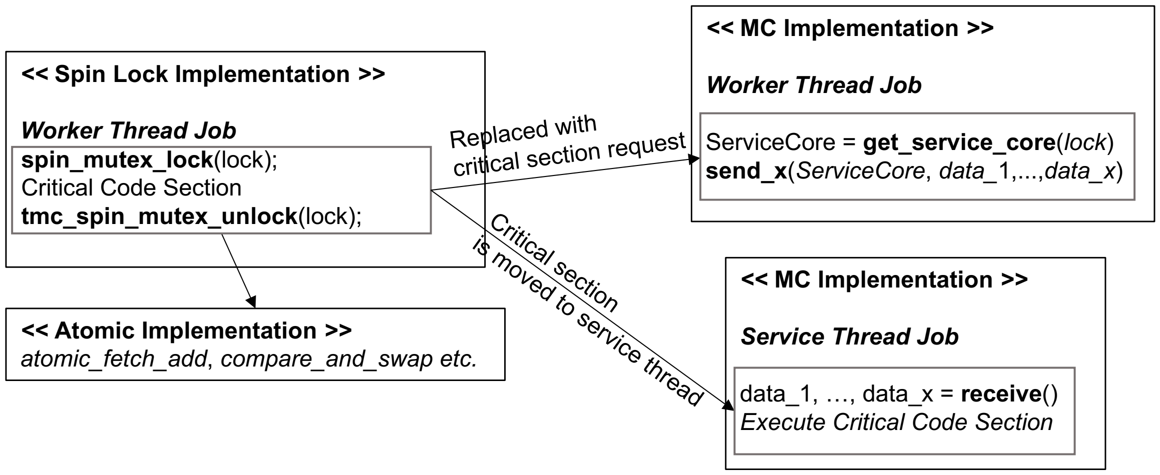
This chapter first outlines the traditional cache coherence based thread synchronization, and discusses its shortcomings. The MC model is then explained to solve the challenges of the traditional synchronization primitives. The chapter concludes with a discussion on the challenges of the MC model, and the proposed solutions.

3.1 Shared Memory Synchronization

Shared memory cache coherence provides ease of programming, flexibility on sharing data between threads, and seamless data movement. However, thread synchronization under shared memory cache coherence at higher core counts becomes a significant performance bottleneck. This is mainly due to expensive ping–ponging 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, in order to realize such synchronization, a separate lock variable needs to be acquired before getting into critical code section, which incurs additional overheads. At lower core counts, and under low contention, these primitives are efficient as they enable concurrent execution of the critical sections. However, when there is contention on shared data, the threads can 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 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, 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 (1000–cores scale), the bouncing affect leads to degradation on performance scaling 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 work, for more efficient and generic thread synchronization, the MC model is proposed and evaluated against both spin-based (Spin) and atomic instruction (Atomic) based synchronization. The MC model and the architectural extensions for efficient implementation of it are discussed in detail in the subsequent sections.



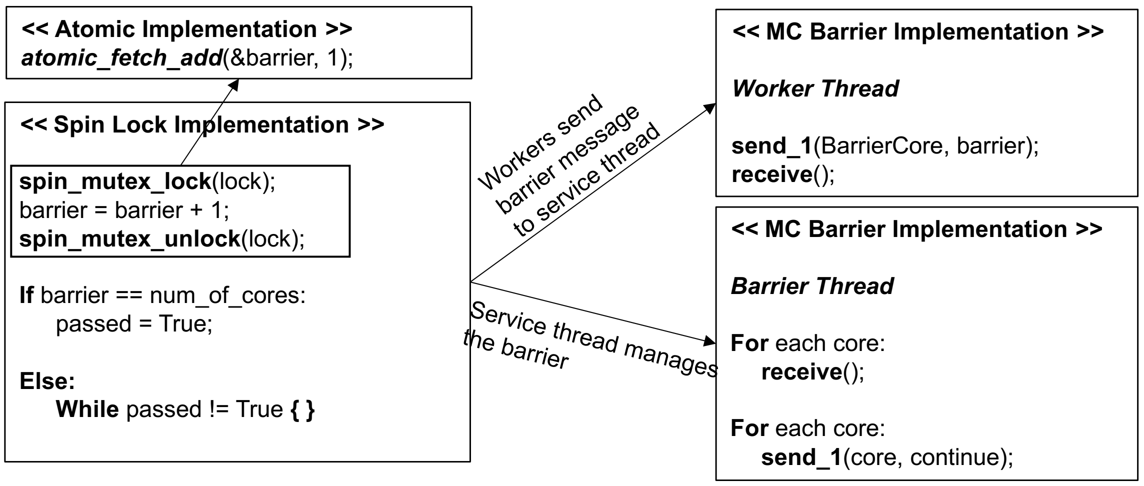
<< Spin Lock Implementation >>
Worker Thread Job
spin_mutex_lock(lock);
Critical Code Section
tmc_spin_mutex_unlock(lock);
<< Atomic Implementation >>
atomic_fetch_add, compare_and_swap etc.
Replaced with critical section request
Critical section is moved to service thread
<< MC Implementation >>
Worker Thread Job
ServiceCore = get_service_core(lock)
send_x(ServiceCore, data_1,...,data_x)
<< MC Implementation >>
Service Thread Job
data_1, …, data_x = receive()
Execute Critical Code Section

Figure 3.1.1: Implementation of fine-grain synchronization using spin-lock, atomic, and MC models.

3.2 Moving Compute to Data Model

Moving computation towards data technique has gained tremendous popularity in recent years due to explosion of computing on massive datasets [25]. 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 towards it. As the executable is significantly smaller than the data, moving overhead is also notably smaller. In a single chip multicore processor, I propose to utilize this approach to mitigate the bottleneck of shared memory thread synchronization, as the core count approaches the 1000-cores scale.

In the proposed MC model, the protected shared data is mapped to a dedicated core and updated only at that specific core by moving computation towards it using explicit messages. In the context of fine–grained synchronization, as demonstrated in Figure 3.1.1, the locks and atomic instructions in a traditional shared memory application (the pseudo code in the left two boxes) are eliminated, and the critical code sections are moved to the dedicated core, termed as *service thread* (the pseudo code in the bottom right box). The remaining cores are utilized as *worker threads*



<< Atomic Implementation >>
atomic_fetch_add(&barrier, 1);
<< Spin Lock Implementation >>
spin_mutex_lock(lock);
barrier = barrier + 1;
spin_mutex_unlock(lock);
If barrier == num_of_cores:
passed = True;
Else:
While passed != True { }
Workers send barrier message to service thread
Service thread manages the barrier
<< MC Barrier Implementation >>
Worker Thread
send_1(BarrierCore, barrier);
receive();
<< MC Barrier Implementation >>
Barrier Thread
For each core:
receive();
For each core:
send_1(core, continue);

Figure 3.1.2: Implementation of barrier synchronization using spin-lock, atomic, and MC models.

(the pseudo code in the upper right box). The *workers* execute the application work, and when they need to execute the critical section, they 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, the shared data is 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 the critical section request depends on the application requirements. 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*. This decision may be needed to ensure data consistency in certain scenarios and may impact performance.

Barrier synchronization is an example of coarse–grained synchronization in which blocking communication is required. Instead of loading and updating the barrier variable atomically by each core as illustrated in Figure 3.1.2 (the pseudo code in the left boxes), the cores send “barrier” messages to a predefined *service thread* (the pseudo code in the right box). 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 messaging. This work employs one of the workers (Core 0) as the *service thread* to manage the 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. Even though the implementation using software messaging is not expected to be efficient, for completeness, the discussion and the evaluation is included as one of the baselines in the work.

3.2.1 Moving Compute to Data Model in Shared Memory (MC_shmem)

In this approach, similar to the explicit communication in MPI [26], 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 shared memory programming model (c.f. Section 5.1). MC_shmem approach is very similar to RCL [10] work in which the locking is done in remote cores, and the 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 and checks the flag of each buffer entry one by one and reads the data and performs the critical section. If the service thread reaches to a buffer slot in which the flag is not set yet, it spins over the flag until the data is 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, the experimental results for the workloads of interest suggest that utilizing a regular buffer outperforms the ring buffer. Therefore, in this thesis, a large shared buffer per service thread is utilized.

This implementation of the MC model pins the shared data block in the service threads and benefits from non–blocking communication. However, the shared buffer utilized for explicit communication bounces between worker and service threads. In the case of Spin and Atomic based synchronization, if there is no contention in the shared data, the ping–ponging affect can completely vanish. On the other hand, implementing MC approach in shared memory leads to constant ping–ponging of the shared buffer. The aforementioned non–blocking communication may ease the cost of ping–ponging by allowing worker threads to hide the communication latency. However, at higher core counts, the impact of ping–ponging is expected to limit performance. As a result, in order to enable efficient implementation of the MC model, hardware support for explicit messaging is introduced as an auxiliary mechanism.

3.2.2 In–hardware Core–to–Core Explicit Messaging

As discussed in the previous section, achieving moving compute to data model using solely cache coherence has similar limitations with other shared memory synchronization models. Therefore, architectural support for core–to–core communication is required to efficiently scale to 1000–cores. In this regard, this work introduces auxiliary support for low-latency core–to–core communication.

Four explicit messaging instructions are introduced in the ISA, and the required micro–architectural support is added to each core pipeline (c.f. Section 4).

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 is not present at the receive queue of the core that issued this instruction. Once the data arrives, 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 which is used to reply to the *sendr* messages.

The details of the explicit messaging protocol and the architectural extensions are discussed in Section 4. By utilizing these low–latency messaging instructions, worker and service cores exchange messages between their register files without involvement of the cache coherence. In the case of fine–grained synchronization where non–blocking communication can be used, the worker threads make use of the *send* instruction to request critical section executions, and it is paired with *recv* instruction on the service thread side. If a blocking communication is needed, then *sendr/resumer* instructions are utilized. Thus, the cache line ping–ponging never happens when exchanging messages between cores.

Once the messaging is implemented with hardware support, the MC model provides two key advantages. First, it prevents unnecessary ping–pong of shared data by pinning it to service threads. However, for certain applications, shared data can be read by other threads to enable work efficient execution of the algorithm. The second advantage is that if non–blocking communication is utilized in the worker side, the MC model enables to efficiently overlap communication overheads with computation. With this approach, *worker* and *service* tasks are pipelined, and additional communication stalls are hidden. In addition, the workers can have multiple back to back in–flight request messages, and possibly further ease the overheads of worker to service thread communication. The shared memory based models, on the other hand, suffer from cache line ping-pong and blocking communication stalls. The benefits of MC with in–hardware explicit messaging are expected to be more notable at higher core counts as the distance between sharer cores and the network congestion increases.

3.2.3 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 in a load–balanced way. This approach requires tuning the ratio of *worker* and *service* threads to achieve near-optimal performance, otherwise the system suffers from the work imbalance. The other approach is to utilize two context 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.

Spatial Moving Compute to Data Model

In this distribution approach, *worker* and *service threads* are spatially assigned to the available cores as shown in Figure 3.2.1. A naive way to achieve the right 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. To overcome these challenges, this work proposes to deploy a profiling based heuristic. The heuristic relies on the

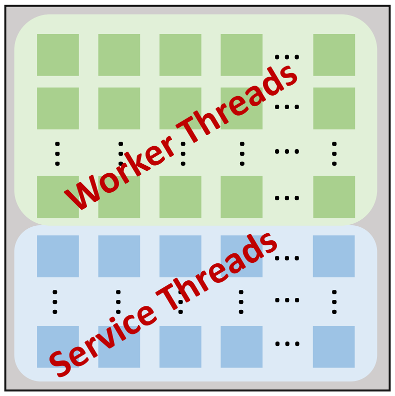


Figure 3.2.1: Spatial distribution of worker and service threads.

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, the required *service thread* count is also high, and vice versa. If the shared work percentage results in less than 1 *service thread* due to very small shared work, 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.

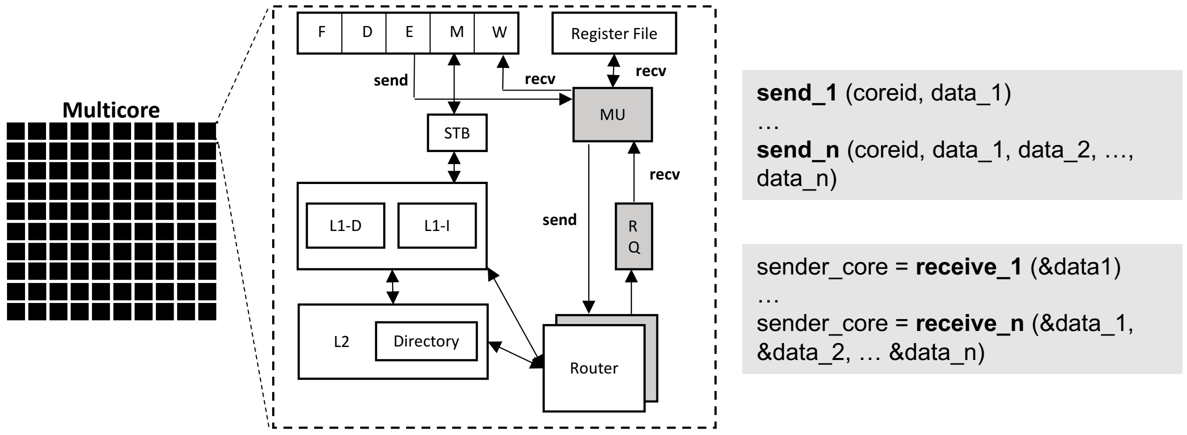
Temporal Moving Compute to Data Model

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 the synchronization becomes 2 times the number of cores which may incur additional communication stalls at higher core counts. Consequently, the spatial 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.

Chapter 4

Architectural Extensions for Efficient Implementation of the MC Model

As discussed in the previous chapter, in–hardware explicit messaging is required for efficient implementation of the proposed moving computation to data model. Hence, this section is dedicated to discuss the baseline multicore architecture, and the hardware extensions to enable fast core–to–core direct communication for the MC model. Furthermore, detailed description of the explicit communication protocol is also provided. The discussed architecture and the messaging protocol is modeled in a simulator environment to evaluate MC model at 1000–cores scale. In addition, the proposed model is also prototyped on Tilera Tile-Gx72 multicore system as it offers hardware support for direct core–to–core messaging on top of shared memory cache coherence. Therefore, the chapter is concluded with a section on the architectural details of the Tile-Gx72 machine.



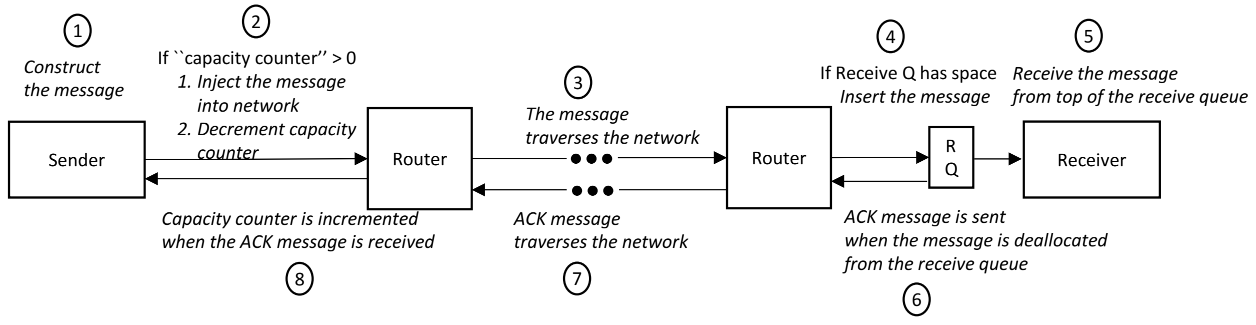
```
send_1 (coreid, data_1)
…
send_n (coreid, data_1, data_2, …,
data_n)

sender_core = receive_1 (&data1)
…
sender_core = receive_n (&data_1,
&data_2, … &data_n)
```

Figure 4.1.1: Overview of a tile and architectural extensions.

4.1 Architectural Overview

The baseline is a tiled multicore architecture. Figure 4.1.1 shows a logical view of a tile within the proposed multicore processor. The tiles are connected to each other with a 2-D mesh interconnection network. Each tile includes a single issue RISC-V [27] core, private level-1 instruction and data caches, a shared last-level cache 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 tiles are extended and shaded modules are introduced to support these instructions on top of shared memory. The syntax of the send/receive operations is shown in the figure. When a send instruction is executed, messaging unit (MU) reads the CoreID and the data from the specified registers, and creates a packet to be sent to the on-chip network. The extended on-chip network transmits the message to the receive queue (RQ) at the destination core, and the messages are buffered in the receive queue until the receive instruction is executed. Once the instruction is executed, MU pulls the data from the RQ and places it into the specified registers. The detailed description of the protocol is discussed in Section 4.2. Note that 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 4-way SIMD that can operate on four 16-bit floating point numbers are added to have state–of–the–art implementations of machine learning algorithms. Associated instructions, such as fused–multiply–add are also integrated into the ISA. Furthermore, RISC-V ISA's standard extension for atomic instructions [27] are implemented. These instructions such as load–reserved 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 is also supported to enable temporal thread distribution of threads as discussed in Section 3.2.3. To support hardware multi–threading, each core is extended with two register files, an explicit messaging–aware switching policy logic, and selection logic for register reads/writes. The switching policy interacts with the receive queue to initiate thread switching when a message arrives. These supports in microarchitecture are utilized to implement the temporal MC model. 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 threads* are given a 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.



1
Construct the message
2
If ``capacity counter'' > 0
1. Inject the message into network
2. Decrement capacity counter
Sender
Router
3
The message traverses the network
Router
4
If Receive Q has space
Insert the message
R Q
5
Receive the message from top of the receive queue
Receiver
Capacity counter is incremented when the ACK message is received
8
ACK message traverses the network
7
ACK message is sent when the message is deallocated from the receive queue
6

Figure 4.2.1: Explicit messaging protocol.

4.2 Explicit Messaging Protocol

As discussed in Section 3.2, the MC model utilizes both blocking and non–blocking communication to accelerate fine and coarse–grained synchronization. The introduced explicit messaging support provides capability to realize both types of communication between worker and service threads.

4.2.1 Non–blocking Communication

This communication type is utilized to implement the MC model to achieve fine–grained non–blocking synchronization. A *send* instruction at the worker core is paired with a corresponding *recv* instruction at the service core 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 the worker core 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 4.2.1 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 and the message size, and the payload. Each send instruction supports up to 4 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 knows 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 same source to a same 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, it stalls until the data arrives 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.

4.2.2 Blocking Communication

In several application scenarios, a 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 the *recv* instruction in the receiver core. Unfortunately, the *send – 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, 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 which flows on the dedicated reply network with *ACK* messages.

Unlike *send* instruction, the *sendr* 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 3.2 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.

4.2.3 Deadlock Freedom

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.

Protocol Level Deadlock Freedom

Limited buffering in receive queues can lead to protocol level deadlocks. In the proposed protocol that does not impose ordering of message arrivals, the application software ensures forward progress. However, if threads impose order of arrival restriction, the finite size of receive queues can lead to protocol level deadlock. This scenario can be resolved by always replying to the sender either explicitly (through *resumer*) or implicitly (through an *ACK* message). This reply message in turn increments the *capacity counter* at the sender, which is used to avoid overflowing the finite sized receive queues.

Network Level Deadlock Freedom

It is assumed that the network guarantees the message arrival orders from the same source to the same destination. In addition, the routing algorithm is assumed to be deadlock–free, or it is assumed to have a deadlock recovery mechanism. Even with these assumptions, the system can deadlock if the same network is deployed for both request messages and the reply messages. For example, if the *ACK* message uses the same network with the *send* message, it can cause circular dependency between threads, and lead to deadlock. Hence, a dedicated network is utilized for *ACK* and *resumer* messages to separate them from *send* and *sendr* messages to remove possible deadlock scenario.

4.2.4 Message Consistency

Certain application communication patterns may require message consistency, i.e., a sender thread must ensure the delivery of prior messages to their destination before commencing with other work. The ISA is extended with a "message fence" instruction, which ensures that all pending messages are pushed into the network and 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.

4.2.5 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 because an in–flight message can be delivered to a core where the thread is not running any more. To properly handle this situation, a clean–up mechanism is required to ensure all in–flight messages are delivered before thread migration can occur. The clean–up mechanism works as follows. The OS halts all the cores from injecting any more 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 the cores have received *ACKs* for all their in–flight messages and there is no explicit message in the network. At this point, the OS can perform the thread migration. It also updates the thread–to–core mapping so that future messages can get to their destination properly. Hardware virtualization support can also be added based on this mechanism.

4.3 Explicit Messaging Hardware Overhead

The architecture requires a receive queue per core to support the proposed protocol as shown in Figure 4.1.1. The design space study is presented in [12] to empirically determine the per core receive queue size using the MC model. In this study, the service thread count is set to the best performing service core count for the workloads that require fine–grained synchronization. Then, the per thread sender capacity counter is swept from 2 – 8 with an increment of 2. For each sender capacity counter setting, the maximum utilization among the receive queues, as well as the corresponding performance speedup is measured. It is shown in the paper that beyond a sender capacity of 4, the performance does not improve and results in an increase in the receive queue size. However, from sender capacity of 2 to 4, the performance speedup is considerable while the receive queue size only increases slightly. Therefore, in this thesis, the per thread sender capacity is set to 4. Using this capacity counter value, a study is conducted to determine the required receive queue size. The workloads of interest in this work are run to completion, and their respective maximum receive queue utilization is obtained. Then, the required receive queue size is determined by choosing the largest receive queue utilization among all the workloads. Finally, $2.4KB$ is determined to be the receive queue size per core in this thesis. 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.

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

Tilera® 's Tile-Gx72™ processor is one of the few commercially available machines that enable similar capabilities to the proposed explicit messaging protocol on top of hardware cache coherence. It is a tiled multicore architecture with 72 tiles interconnected with 2-D mesh networks-on-chip, called iMesh Interconnect. Each tile consists of a 64-bit VLIW core, 32 KB private level-1 data and instruction caches, and a 256 KB shared level-2 (L2) cache. A directory is integrated into the L2 cache slices to support a directory–based cache coherence protocol. *Tile-Gx72™* architecture also offers various configurations for data placement and caching schemes. By default, a cache line is homed at an L2 cache using a hardware hashing scheme, and also replicated in the L2 slice of the requesting core. Experiments with and without replicating cache lines in the local L2 slice of the requesting core varied performance by an average of 1%. Hence, in the rest of the thesis the default L2 homing scheme is utilized. In addition, networks-on-chip are included in each tile to communicate with other tiles, I/Os and the on-chip memory controllers. There are two groups of networks in the system. One is a set of “Memory Networks”, which are utilized for memory and coherence traffic. The other one is a set of “Messaging Networks”, which are deployed to explicitly send messages between tiles (User Dynamic Network (UDN)) and the I/O (I/O Dynamic Network (IDN)) devices. While User Dynamic Network (UDN) is used to enable tile–to–tile direct messaging, I/O Dynamic Network (IDN) is used to send messages to the I/O. UDN is leveraged for moving compute to data model in this work.

Each tile contains four UDN queues for explicit messaging. These queues are implemented as small FIFO queues. Each queue is mapped to a special purpose register, which is used to send and receive data between execution units without any involvement of the cache coherence protocol and traffic. For example:

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

move r0, udn0 is a receive operation in which the sent data is received and placed in one of the queues, and since the queues are mapped to special UDN registers, the data is read from the corresponding register ($udn0$ in this case) when it arrives in the specified queue. If the message does not make it to the queue when the "*move*" instruction is executed, this operation stalls the core pipeline.

Tile-Gx72^{TM} supports Tilera Multicore Components (TMC) library [28] to initialize and make use of the UDNs. Hence, low level instructions are not used for explicit communication. For this work, the library calls provided by TMC library for tile–to–tile messaging are used (c.f. Section 5.2). To be able to make use of the UDN networks, threads are pinned to cores based on their thread IDs in an ascending order. In *Tile-Gx72*^{TM}, the threads are spatially distributed among available cores.

UDNs in *Tile-Gx72*^{TM} supports both blocking and non–blocking communications using the library calls for send/receive operations. 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 4.2. However, the blocking communication is used to implement barrier synchronization and the database workload in this thesis, and the offered explicit messaging support is sufficient to realize both database workload and the barrier without deadlocking the system.

Chapter 5

Programming with Moving Compute to Data Model

The proposed moving compute to data model and the architectural extensions for efficient implementation of it are discussed in the previous chapters. Since the proposed approach utilizes a different model to accelerate synchronization, it is also significant to discuss how a given application is programmed with the MC model. Therefore, this chapter is dedicated to discuss programming model, the synchronization primitives and the messaging API in *Tile-Gx72^{TM}* and Simulator, and the illustration of how a shared memory application is seamlessly translated to the proposed MC model.

5.1 Programming Model

The proposed MC model utilizes shared memory parallel programming model. Threads are created within a process using the Pthreads library [29], and all threads are allowed access to shared data structures. Even though Pthreads is utilized in this work, OpenMP [30] programming model can

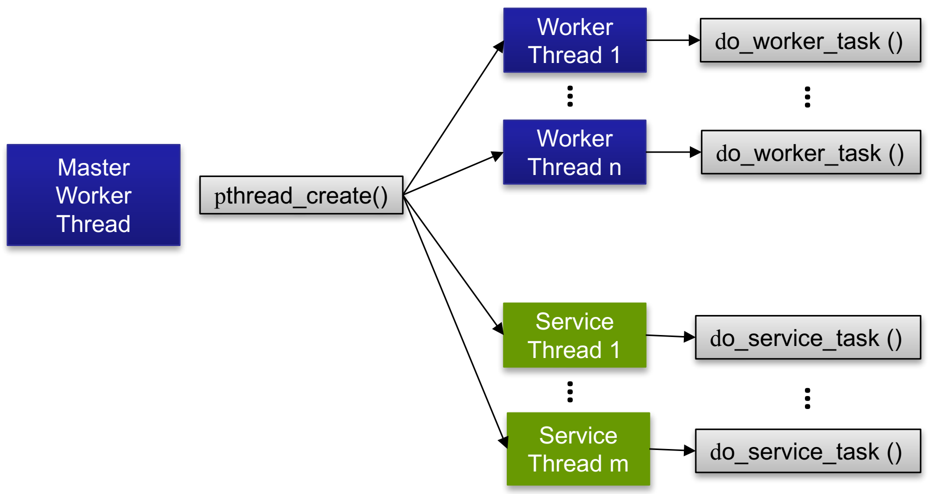


Figure 5.1.1: Thread management with the MC model.

easily be adapted as well. The only difference from traditional thread creation is that two sets of threads are created as shown in Figure 5.1.1, and distributed by either jumping to worker routine or service routine. Then, the programming model replaces traditional thread synchronization with the explicit messaging based MC protocol. The service threads perform critical section execution with the request of the worker threads, as discussed in Section 3.2. 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 [10], Coccinelle [31] or similar refactoring tools can be easily utilized to transform existing applications. However, this work does not utilize any tool to implement the moving computation to data model. Manual transformations of the representative applications are illustrated to show how shared memory synchronization is ported to the MC model.

5.2 Synchronization on *Tile-Gx72™*

Tile-Gx72™ platform provides Tilera Multicore Components (TMC) library to make use of various synchronization and communication capabilities of the machine. The provided library contains various spin–based synchronization primitives such as simple spin–locks (`tmc_spin_mutex_t`) and barriers. In addition, it also offers queue based spin–locks (`tmc_spin_queued_mutex_t`) which provide better fairness as compared to the simple spin–lock. For the workloads presented in this thesis, the performance difference between simple and queue based locks is negligibly small, hence simple locks are utilized. The TMC library also includes atomic memory operations to serialize updates on a shared data. In this work, `arch_atomic_val_compare_and_exchange()` and `arch_atomic_increment()` are used to implement critical sections of graph workloads as discussed in Section 5.4.1. Similarly, *atomic increment* is also utilized for barrier implementation.

The MC model is implemented using the UDN capability of the *Tile-Gx72™*. The TMC library provides interfaces to enable and utilize the inter–core communication. First of all, the main process that creates the threads executes `tmc_udn_init()` which forms a UDN hardwall with the given `cpu_set_t`. The `cpu_set_t` specifies all the CPUs that are granted access to the messaging network, and `tmc_udn_init()` routine creates a rectangle that covers all the specified CPUs. After the threads are created, based on the given thread id, the threads are pinned to the cores using `tmc_cpus_set_my_cpu()` routine. Then, each thread calls `tmc_udn_activate()` to enable communication using the messaging network within the created UDN hardwall.

Tile-Gx72™ contains 4 UDN demux queues, hence a message can be sent and received using any of the queues. The programmer needs to make sure that the queue tag provided while sending the data is matched on the receiver side by calling the proper receive routine. For example, in the MC implementation, demux queue zero is utilized to send/receive the critical section requests. As shown in Algorithm 1, `tmc_udn_send_n()` routine is used to send the request messages to the

Algorithm 1 Messaging API between worker and service cores on *Tile-Gx72*^{TM}.

```
1: << Worker thread sends x words >>
2: tmc_udn_send_n (header, UDN0_DEMUX_TAG, data_1, data_2, ..., data_n)
3:
4: << Service thread receives x words >>
5: data_1 = tmc_udn0_receive()
6: data_2 = tmc_udn0_receive()
7: ...
8: data_n = tmc_udn0_receive()
```

service thread where the message is being placed into the demux 0. It accepts up to 20 words of data, meaning that n can vary from 1 to 20. The send operation requires a $header$ which is necessary to route the data to the receiver core. The $header$ for each core is statically created before getting into the parallel region by calling `tmc_udn_header_from_cpu(coreid)`, and placed into an array of headers. Before sending a message, with a simple lookup, the header for the corresponding core is obtained and the message is sent. Similarly, on the service core side, the data is being received from demux 0 by calling `tmc_udn0_receive()` routine n times.

5.3 Synchronization on Simulator

The simulator supports RISC–V's standard extension of atomic instructions in addition to load–reserve/store–conditional instructions. The load–reserve/store–conditional instructions are deployed to implement spin–locks as similar to TMC library of Tilera. The barrier synchronization is implemented using both spin–lock and the atomic fetch–and–add instruction, `amoadd`. For the critical section implementations, `amoadd, and amoswap` instructions are utilized. Since RISC–V tool chain supports the GCC builtins for atomic memory operations, in the actual implementation, such builtins are utilized. For example, for atomic *fetch–and–add*, `__sync_fetch_and_add ()` is deployed.

Algorithm 2 Messaging API between worker and service cores using send/receive instructions on *Simulator*.

```
1: << Worker thread sends x words >>
2: sendmsg_n (coreid, data_1, data_2, ..., data_n)
3:
4: << Service thread receives x words >>
5: src_core = recvmsg_n (&data_1, &data_2, ..., &data_n)
```

The MC model is implemented using the send, receive instructions added in the ISA. Similar to Tilera, an API library for messaging is implemented. The syntax of the API is similar to *Tile-Gx72^{TM}* with some differences. In the simulator, there is only a single receive queue per core, hence there is no need to tag the messages. The data is sent and received as shown in Algorithm 2. Instead of a special header, the send operation receives core id as an argument to determine the destination. The receive instruction returns the sender core id in case a reply message needs to be sent. The capacity counter discussed in Section 4.2 for send operation can be initialized using `set_thread_capacity ()`. If it is not initialized by the programmer, default value of 1 is used.

Algorithm 3 Messaging API between worker and service cores using sendr/resumer instructions on *Simulator*.

```
1: << Worker thread sends x words >>
2: return_data = sendr_n (coreid, data_1, data_2, ..., data_n)
3:
4: << Service thread receives x words and resumes the worker >>
5: src_core = recvmsg_n (&data_1, &data_2, ..., &data_n)
6: resumer (src_core, reply_data)
```

As discussed in Section 4.2.2, in order to implement blocking communication, sendr/resumer instruction pairs are utilized. Various versions of sendr and resumer operations are implemented in the runtime library. An example is illustrated in Algorithm 3. In the algorithm, the worker sends n number of words and waits until it receives a single word as a reply. The core id of the sender is stored in line 5, then used in the $resumer$ operation to send explicit reply. There are possible variations in which the sender sends a word of data, then expects multiple words of reply, but these variations are not used in this thesis.

5.4 Workload Illustrations

5.4.1 Graph Workloads

Six graph workloads are ported from CRONO benchmark suite [8]. Single Source Shortest Path, Breadth First Search and Triangle Counting involve both fine–grained and coarse–grained synchronization while PageRank, Connected Components and Community Detection contain coarse–grained synchronization such as barriers and reduction. The workloads with coarse–grained synchronization are implemented by replacing the shared memory barriers with the MC barriers. PageRank and Connected Components also involve reduction phases at the end of each iteration. These reductions are also implemented using the MC based reduction similar to the barrier implementation. The details of the workloads with coarse–grained synchronization are not discussed further. In the following subsections, the transformation of the workloads with fine–grained synchronization are discussed in detail.

Triangle Counting (TC)

TC is a well-known graph algorithm that counts triangles in a graph for various statistical purposes in an application [32]. Figure 5.4.1 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 the figure). After counting the connectivity for each node, all the participating threads hit a barrier. Then each

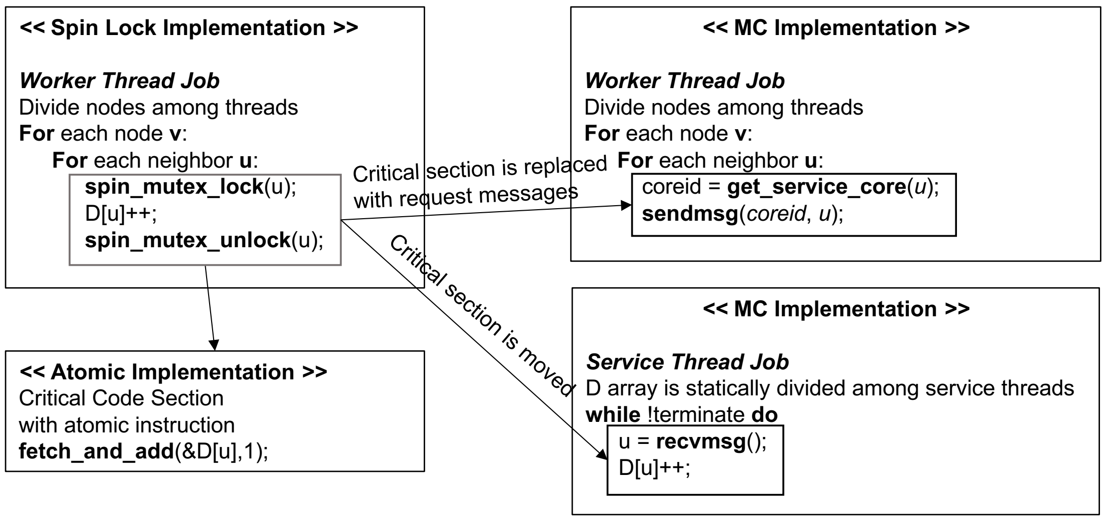
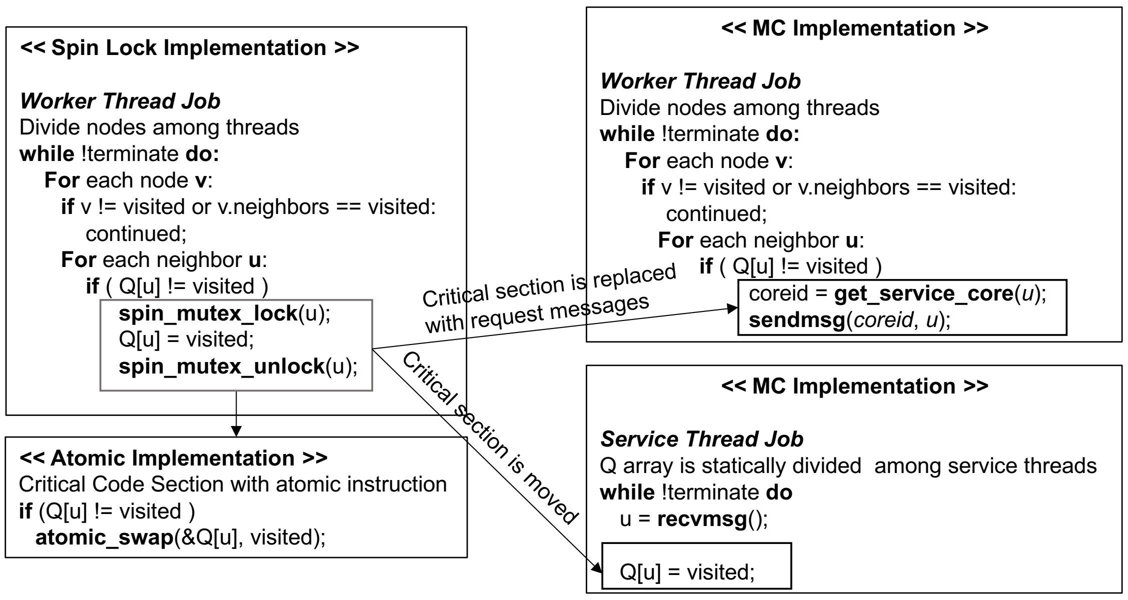


Figure 5.4.1: Pseudo code of triangle counting implementation using Spin, Atomic and MC.

thread calculates their 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 the critical section, which results in acquiring a lock multiple times for each node. Therefore, the contention on shared data is expected to be high for this algorithm depending on the graph input and the number of cores. As a result of contention, the lock acquisition overhead is also elevated due to retries and cache line ping–pong for both shared data and the lock variables. Implementing the algorithm using lock–free data structure by employing the atomic fetch–and–add (FAA) instruction (lower left box in the figure) removes the locks. This significantly reduces the overheads of acquiring locks as the atomic FAA instruction does not fail. However, the shared data itself still ping pongs between cores.

The MC model pins the shared data structures D at dedicated cores and ships the critical section work to *service threads* as discussed in Section 3.2. The MC implementation is presented in the right side of the figure. The code section that needs atomic updates is migrated to *service threads* (see Figure 5.4.1). Only the neighbor node id is needed for critical section, hence the workers send one word of data as message to the corresponding *service thread*. Similar to Atomic, MC



```
<< Spin Lock Implementation >>

Worker Thread Job
Divide nodes among threads
while !terminate do:
  For each node v:
    if v != visited or v.neighbors == visited:
      continued;
    For each neighbor u:
      if ( Q[u] != visited )
        spin_mutex_lock(u);
        Q[u] = visited;
        spin_mutex_unlock(u);
```

```
<< Atomic Implementation >>
Critical Code Section with atomic instruction
if (Q[u] != visited )
  atomic_swap(&Q[u], visited);
```

```
<< MC Implementation >>

Worker Thread Job
Divide nodes among threads
while !terminate do:
  For each node v:
    if v != visited or v.neighbors == visited:
      continued;
    For each neighbor u:
      if ( Q[u] != visited )
        coreid = get_service_core(u);
        sendmsg(coreid, u);
```

```
<< MC Implementation >>

Service Thread Job
Q array is statically divided  among service threads
while !terminate do
  u = recvmsg();
  Q[u] = visited;
```

Critical section is replaced with request messages
Critical section is moved

Figure 5.4.2: Pseudo code of BFS implementation using Spin, Atomic and MC.

also eliminates the locks and related overheads. In addition, it also pins the shared data to *service threads* and prevents cache lines from unnecessarily bouncing 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 workers can load the next neighbor id, and execute lookup function to determine the *service thread id* for the next request.

Breadth First Search (BFS)

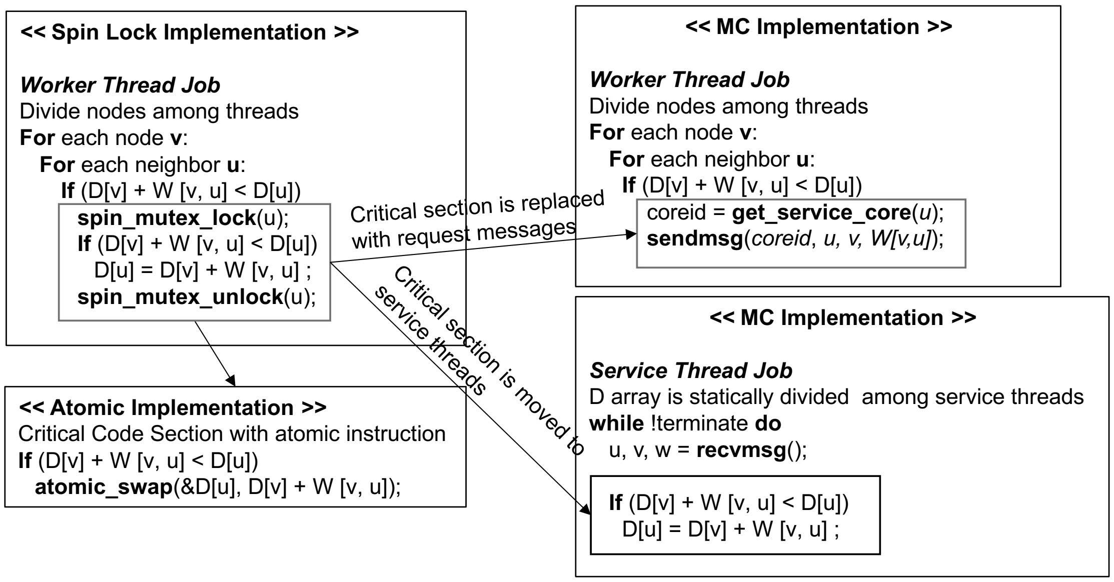
Breadth First Search (BFS) is quite different algorithm as compare to TC. Figure 5.4.2 illustrates the pseudo code of BFS algorithm using spin–locks, atomic instructions and the MC model. The spin–lock implementation is shown in the upper–left box. Each thread goes through its part of the graph, and visits the nodes iteratively by opening new pareto fronts in each iteration. The visiting part is protected with fine–grained locks to ensure that no other thread visits the same node. Unlike TC, the algorithm contains tests which prevent redundant lock acquisitions. It tries to guarantee that each node is visited only once in whole program execution. This significantly reduces the contention on the shared data, and the amount of shared work in this algorithm. Hence, fine–grained synchronization is not significant part of the workload's completion time. Similar to TC, the locks are removed as shown in the lower–left box, and visit operation is done using atomic compare–and–swap instruction to eliminate the lock overheads.

The MC version is also implemented as shown in the figure (the pseudo code in the right two boxes). The tests to reduce the redundant visits stay in the worker side to eliminate redundant critical section requests. Even though this causes shared data to ping-pong between *worker* and *service threads*, it leads to more work efficient execution, hence yields better performance.

As mentioned earlier, due to the tests that filter out redundant critical section executions, fine grain synchronization does not contribute much work to the total execution time in BFS. However, barrier synchronization becomes dominant for this workload since worker threads do not perform in a lockstep fashion due to work efficiency optimizations. Therefore, implementing barrier synchronization using the MC model is expected to help performance scaling as compared to the Spin and Atomic versions.

Single Source Shortest Path (SSSP)

SSSP is used to compute the shortest path for a user defined source node in a graph. In this work, 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 are calculated until all the nodes are accessible. The nodes in each range are divided among cores, and the cores visit and relax the neighbors of their nodes one at a time. As seen in Figure 5.4.3 (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 the unnecessary


Critical section is replaced with request messages
Critical section is moved to service threads

```
<< Spin Lock Implementation >>

Worker Thread Job
Divide nodes among threads
For each node v:
  For each neighbor u:
    If (D[v] + W [v, u] < D[u])
      spin_mutex_lock(u);
      If (D[v] + W [v, u] < D[u])
        D[u] = D[v] + W [v, u] ;
      spin_mutex_unlock(u);
```

```
<< Atomic Implementation >>
Critical Code Section with atomic instruction
If (D[v] + W [v, u] < D[u])
  atomic_swap(&D[u], D[v] + W [v, u]);
```

```
<< MC Implementation >>

Worker Thread Job
Divide nodes among threads
For each node v:
  For each neighbor u:
    If (D[v] + W [v, u] < D[u])
      coreid = get_service_core(u);
      sendmsg(coreid, u, v, W[v,u]);
```

```
<< MC Implementation >>

Service Thread Job
D array is statically divided among service threads
while !terminate do
  u, v, w = recvmsg();
    If (D[v] + W [v, u] < D[u])
      D[u] = D[v] + W [v, u] ;
```

Figure 5.4.3: Pseudo code of SSSP implementation using Spin, Atomic and MC.

locking. However, since the algorithm may not converge easily depending on the input, the test may not fail as often. Therefore, it results in multiple lock acquisitions per node, hence the contention on the shared data also gets higher depending on the input graph and the number of cores. Similar to TC and BFS, 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 left box in Figure 5.4.3). The whole relaxation code is executed by the *service thread*. Contrary to TC and BFS, SSSP's critical section requires three words to operate on. 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 the request messages, and relax the shared distance array. Even though it causes D array to be read by the workers and results in additional coherence traffic, the test before the critical section remains there to make sure the algorithm is work efficient. This prevents sending unnecessary critical section requests and jamming the *service threads* with messages. The MC model enables pipelining the critical section work with the worker tasks by using non–blocking communication. In addition, as this algorithm requires to load multiple data before sending a request message, it offers a lot of room to overlap communication overheads with the memory stalls.

5.4.2 Machine Learning Workloads

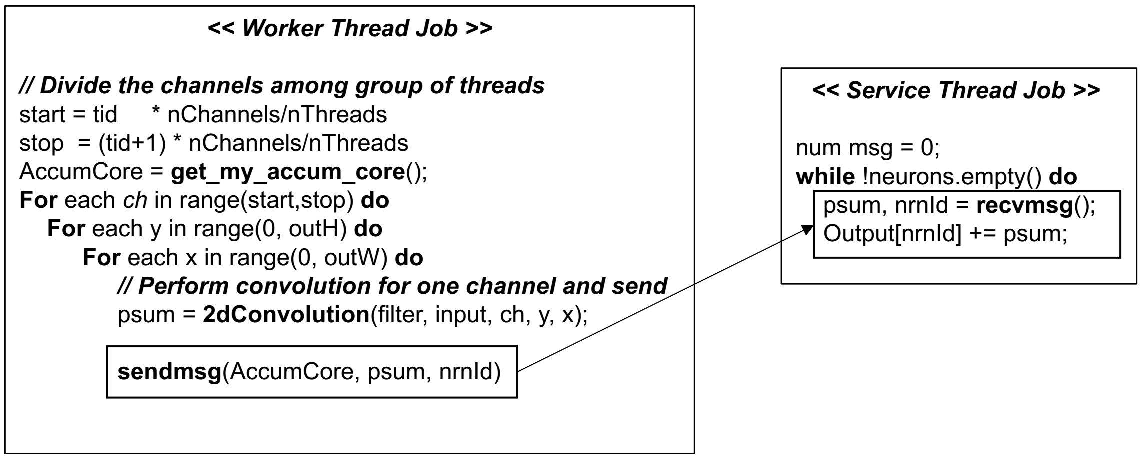
The recent success of deep neural networks (DNNs) on computer vision [33] [34] [35] and natural language processing [36] have attracted the attention of both academia and industry. In this context, many accelerators are proposed for both high performance [37] and low energy applications [38] [39]. Specially, GPUs are shown to be effective in processing of DNNs due to their high FLOP rate, memory bandwidth, and large concurrency capabilities. In this thesis, two machine learning workloads, namely AlexNet [35] and SqueezeNet [40] are developed and evaluated to show that if the proper computation mechanisms and communication models are in place, a general purpose multicore architecture can be utilized for these workloads. Because the convolutional layers account for most of the computation for these workloads, only the implementation of the convolutional layers are discussed. As mentioned in Section 4, the simulator supports 4–way SIMD instructions with 16–bit floating point to improve the computational power of the system and reduce the pressure on the memory subsystem. In this way, the machine learning workloads are implemented using these supports to have state–of–the–art performance.

Convolutional Layer Implementation

Convolutional layers are the computationally expensive part of a neural network [41] [42]. Each layer comprises multiple kernels to produce multiple output feature maps. Each kernel has multiple channels corresponding to input channels. Each neuron output is calculated with the accumulation of 2D convolution operations using the channels of the corresponding kernel and input. All three data structures are reused for various purposes. Each kernel is reused for each neuron in an output feature map, and the input is also reused with each kernel to generate different output feature maps. Similarly, each output is also reused when accumulating the 2D convolutions. To minimize performance overheads, reuse of these data structures in the L1 cache must be maximized. Different parallelization strategies can be utilized based on the filter size of the given neural network.

A coarse–grained parallelization strategy is that all the neurons in a layer are tiled, and tiles are divided among the available threads. Each thread performs all the computation for the neurons in its tiles. At the end of each layer, the threads are synchronized with a barrier. This approach enables reuse of the kernel data. Each kernel channel is brought to L1 data cache one by one, and reused for all the neurons in each tile. This approach also allows data reuse for the output data structure if it fits in the level-1 cache with the 2D kernel and the corresponding input. That is to say, when bringing the next kernel channel for the same tile, the outputs are still in the L1 cache and loaded without expensive L1 miss. However, this approach creates imbalance when using larger tile size in some of the cases. For example, layer 3 of AlexNet contains 384 13×13 neurons. If 13×13 tiles is used to have good reuse of filter and output data in L1, when using 256–core system, some of the cores get more work than others. This causes underutilization of the system. If the tile size is reduced to have more concurrency, then it hurts the data reuse.

More optimized implementation makes use of fine–grained parallelization strategy. It is achieved by dispatching multiple threads to work on one neuron. This requires updating the same neuron by multiple threads. Therefore, this must be implemented using critical code sections. The critical section work can be realized utilizing shared memory spin locks. However, it does not perform as well as the naive implementation due to the large overheads of shared memory locks. Therefore, it is only implemented using the MC model, as depicted in Algorithm 5.4.4. In this approach, as it is seen in the algorithm, the cores are clustered into small thread groups and each group works on a



```
<< Worker Thread Job >>

// Divide the channels among group of threads
start = tid     * nChannels/nThreads
stop  = (tid+1) * nChannels/nThreads
AccumCore = get_my_accum_core();
For each ch in range(start,stop) do
   For each y in range(0, outH) do
       For each x in range(0, outW) do
           // Perform convolution for one channel and send
           psum = 2dConvolution(filter, input, ch, y, x);

           sendmsg(AccumCore, psum, nrnId)
```

```
<< Service Thread Job >>

num msg = 0;
while !neurons.empty() do
   psum, nrnId = recvmsg();
   Output[nrnId] += psum;
```

Figure 5.4.4: Pseudo code for fine–grained parallelization of convolution layers using the MC model.

tile of neurons. To calculate a neuron output, the kernel channels are divided among the threads in the group (the most outer loop in the pseudo code), and each thread calculates partial sums for the neurons in its tile using its kernel channels (see the remaining two loops in the pseudo code). One of the threads in each group is employed to accumulate the partial sums. The partial sum of each neuron for each kernel channel is calculated, and sent to the accumulation thread using non–blocking send operation. The service thread receives the partial sums from other threads, and accumulates it over the corresponding neuron. Figure 5.4.5 shows the distribution of the data structures for each thread group. As seen, in this approach, the neuron outputs reside in separate service cores, and they are reused in their level-1 cache. Similarly, the kernel channels are also reused for all the neurons in the tile. Because this approach deploys a fine–grain parallelization strategy, it enables higher concurrency without loosing the data reuse benefits. However, it introduces additional communication overhead. In order to alleviate the communication overhead, the threads in each cluster are mapped to close proximity to each other. In addition, utilizing non–blocking communication hides most of the communication latency. While one partial sum is traversed in the network and processed by the service thread, the workers start calculating the next partial sums. To

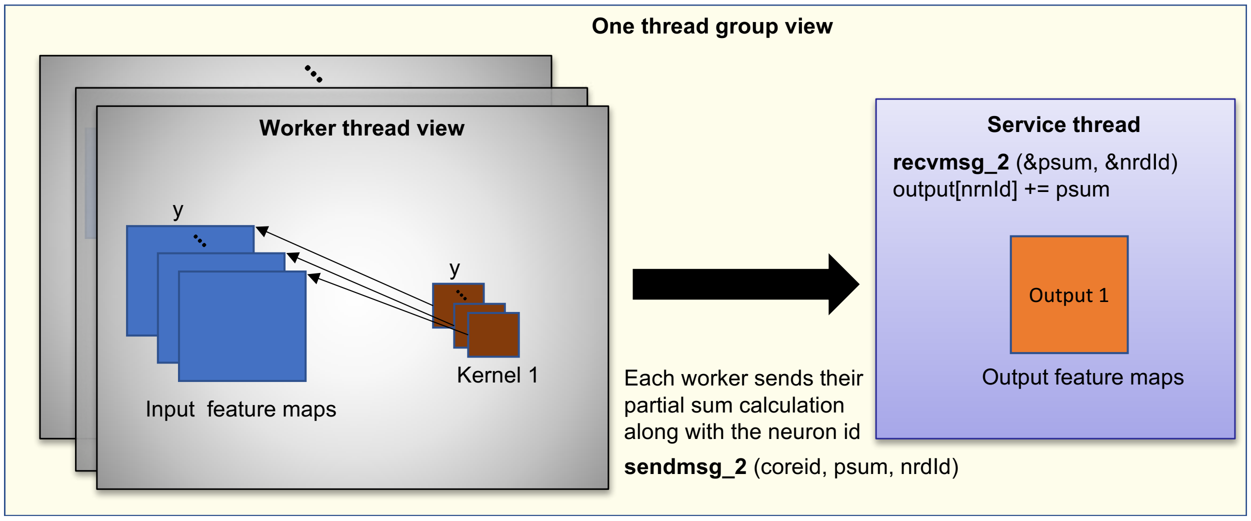


Figure 5.4.5: Overview of the convolutional layer in fine–grained parallelization using the MC model.

get the best performance out of this approach, one needs to adjust the number of threads per group. For this work, the optimal number of threads per group is empirically decided to be 8.

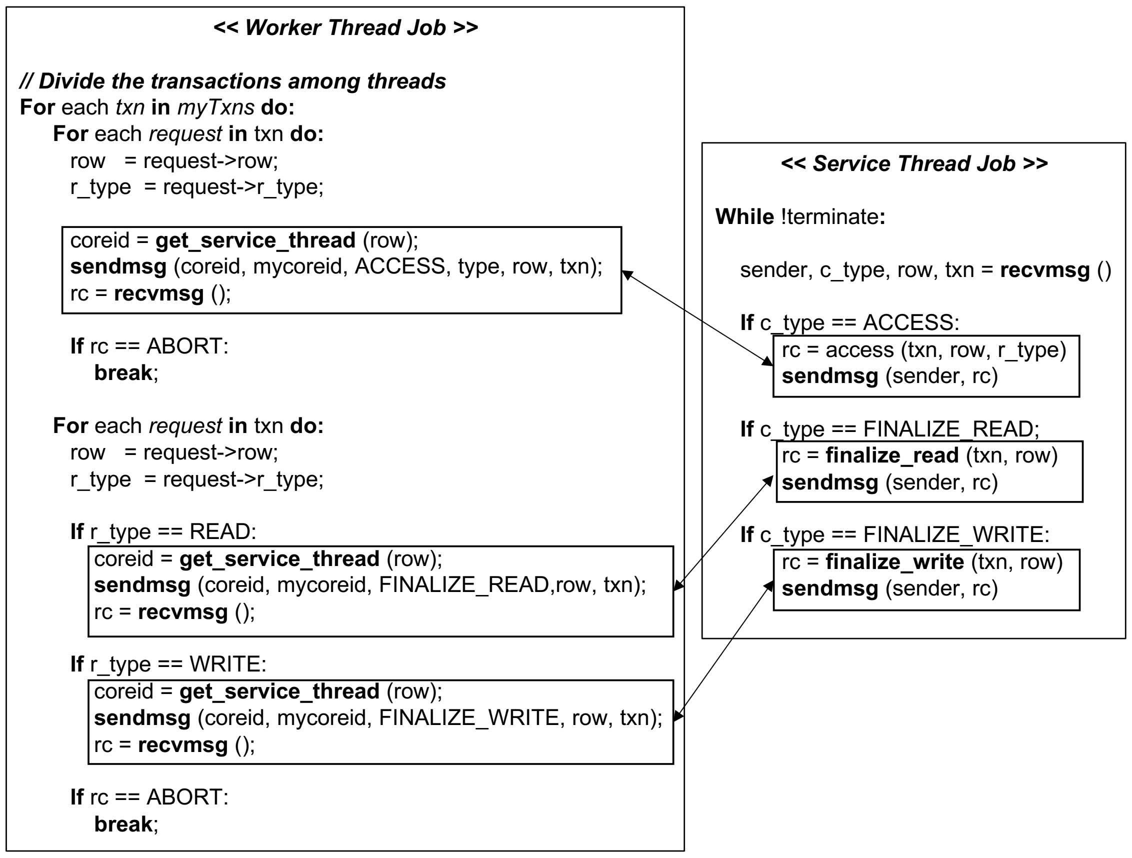
The convolutional layers in AlexNet are implemented using both aforementioned coarse and fine–grained approaches. In the coarse–grained parallelization, the only synchronization point is the barrier at the end of each layer. Hence, this configuration is implemented using various barriers implementations discussed in Section 3.2. In addition, AlexNet is also implemented with fine–grained parallelization. In the case of SqueezeNet, both the number of channels and the filter sizes are small with the exception of a few layers out of 26 layers. Therefore, it does not have opportunity to assign multiple threads to work on a single neuron. Consequently, it is implemented using only the naive parallelization (coarse–grained) approach. Note that *Tile-Gx72^{TM}* machine does not have any special support for floating point SIMD. Therefore, these workloads are implemented using scalar floating point in that machine. In addition, it only contains 72 cores, hence there is enough concurrency and data reuse opportunity. Therefore, fine–grained parallelization is omitted in *Tile-Gx72^{TM}*.

```
<< Worker Thread Job >>

// Divide the transactions among threads
For each txn in myTxns do:
  For each request in txn do:
    row    = request->row
    r_type  = request->type

    spin_mutex_lock (row->lock)
    rc = access (txn, row, r_type)
    spin_mutex_unlock (row->lock)

    If rc == ABORT:
      break

  For each request in txn do:
    row    = request->row
    r_type  = request->type

    If r_type == READ:
      spin_mutex_lock (row->lock)
      rc = finalize_read (txn, row)
      spin_mutex_unlock (row->lock)

    If r_type == WRITE:
      spin_mutex_lock (row->lock)
      rc = finalize_write (txn, row)
      spin_mutex_unlock (row->lock)

    If rc == ABORT:
      break
```

Figure 5.4.6: Pseudo code of YCSB implementation using shared memory synchronization.

5.4.3 Database Management System, Yahoo! Cloud Serving Benchmark

The database benchmark, Yahoo! Cloud Serving Benchmark (YSCB) [43], is ported from [44]. The paper discusses various concurrency control schemes with two–phase locking and timestamp ordering. In this thesis, one of the modern concurrency control schemes, Hekaton [45], is employed. The scheme is a timestamp ordering scheme, which keeps track of write accesses by using per row write history tables. In the commit time, the database management system checks if the reads of the current transaction overlaps with the other concurrent writes. If there are overlapping writes, the transaction is aborted. If there are no overlapping writes, the changes in the transaction are applied to the database.



```
<< Worker Thread Job >>

// Divide the transactions among threads
For each txn in myTxns do:
    For each request in txn do:
      row    = request->row;
      r_type  = request->r_type;

      coreid = get_service_thread (row);
      sendmsg (coreid, mycoreid, ACCESS, type, row, txn);
      rc = recvmsg ();

      If rc == ABORT:
         break;

    For each request in txn do:
      row    = request->row;
      r_type  = request->r_type;

      If r_type == READ:
         coreid = get_service_thread (row);
         sendmsg (coreid, mycoreid, FINALIZE_READ,row, txn);
         rc = recvmsg ();
      If r_type == WRITE:
         coreid = get_service_thread (row);
         sendmsg (coreid, mycoreid, FINALIZE_WRITE, row, txn);
         rc = recvmsg ();

      If rc == ABORT:
         break;
```

```
<< Service Thread Job >>

While !terminate:

   sender, c_type, row, txn = recvmsg ()

   If c_type == ACCESS:
      rc = access (txn, row, r_type)
      sendmsg (sender, rc)

   If c_type == FINALIZE_READ;
      rc = finalize_read (txn, row)
      sendmsg (sender, rc)

   If c_type == FINALIZE_WRITE:
      rc = finalize_write (txn, row)
      sendmsg (sender, rc)
```

Figure 5.4.7: Pseudo code of YCSB implementation using the MC model.

Figure 5.4.6 shows the high level implementation of the workload with shared memory synchronization primitives. As shown, the shared memory implementation is realized with multiple critical sections. For a given transaction, the requested row is accessed by acquiring the corresponding lock. After all the requests of a transaction are served, the transaction is finalized. In the finalize stage, the database tuples are again locked for both read and write accesses to prevent race conditions. Contrary to the graph workloads, the critical sections in this benchmark can not be implemented using a single atomic instruction as they are more complex, hence YCSB is implemented using only spin–locks.

The MC implementation moves the three critical sections to the service threads, and replaces them with request messages in the worker threads as depicted in Figure 5.4.7. The worker threads send their request types along with the necessary data. Depending on the request type, the service thread executes the corresponding critical section. Unlike other workloads, YCSB implementation requires blocking communication between worker and service threads because the workers need a return condition to determine if the transaction needs to be aborted or not. Hence, the communication overheads are not hidden in this workload. The performance benefit is expected to come from pinning the shared data structures to the service threads. In this workload, the actual database entries are still accessed by the workers, hence they can still ping–pong. However, this sharing does not affect the performance much because writes to the shared database entry only happens during the commit time. On the other hand, the history table for each database entry is read and written more frequently within the critical sections, hence the contention over the history tables is higher. In the MC implementation, the history tables are hashed among service threads, hence they do not bounce between private caches of the cores, which is expected to offer superior performance at higher core counts.

5.4.4 More Complex Critical Sections

The focus of this thesis is to utilize the MC model to accelerate critical sections in a shared memory application at 1000–cores scale. Therefore, the most common critical sections on regular data structures are the focus of the thesis. As seen in the examples in previous sections, the process of converting an application to utilize MC involves moving the critical sections to dedicated service threads, and dividing the shared data between the dedicated threads to maintain the atomicity of the 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 [46]. 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 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 type 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 benefit from the MC model. However, since the purpose of this work is to focus on the most common critical sections on regular data structures, further discussions and analysis are not included in the work.

Chapter 6

Evaluation on *Tile-Gx72^{TM}*

As mentioned in Section 4.4, *Tile-Gx72^{TM}* is one of the fewest multicore systems with both shared memory cache coherence and hardware explicit messaging. Hence, in this chapter, the MC model is evaluated using Tilera *Tile-Gx72^{TM}* upto 64 cores. This study is important to show the applicability of the proposed model in a real machine setup. The first section talks about the methodology of the study. Then, an extensive study that evaluates the MC model against the shared memory synchronization primitives is discussed. A core scaling study is presented to demonstrate the enhanced performance of MC model as the number of cores are increased.

6.1 Methodology

Tile-Gx72^{TM} multicore processor is deployed for evaluation. It runs at 1 GHz and is equipped with 16 GB of DDR3 main memory. It runs a linux version that is modified for Tilera architecture. A modified version of GCC 4.4.7 that supports Tilera specific features is utilized for the compilation of the benchmarks.

Three thread synchronization models are evaluated in this work. The Spin model is implemented using Tilera's spin–based primitives. The Atomic model implements the critical section using a single atomic instruction. Finally, the moving compute to data (MC) model implements each critical section work by serializing it at a dedicated core. All synchronization models are discussed in Section 5.1.

6.1.1 Performance Metrics

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

- **Completion Time:** Completion time is measured by running all the workloads to completion, and only the parallel region is measured in each application. Memory allocations, initialization of data, and thread spawning overheads are not taken into account. Every run is repeated ten times and the average number is reported.

- **Load Imbalance:** Load imbalance is determined by calculating the variability in the instruction counts of the cores. Number of instructions for each core is determined using the hardware event counters in Tilera. The variability across instruction counts of the cores is calculated using the following formula:

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

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

Table 6.1.1: Input graphs and their respective statistics.

Inputs	**Nodes**	**Edges**	**Degree**
Mouse Brain [47]	562	0.57M	1027
CA Road Network [48]	1.9M	5.5M	2.8
Facebook [49]	2.9M	41.9M	14.3
LiveJournal [49]	4.8M	85.7M	17.6

6.1.2 Benchmarks and Inputs

Six graph benchmarks from the CRONO [8] suite are adopted for this work, namely SSSP, TC, BFS, PAGERANK, CC, and COMM. In addition, a database workload (YCSB) from [44] is deployed. These benchmarks are ported to *Tile-Gx72*^{TM} using Spin, Atomic and MC synchronization models. For all models, pthreads library is used to spawn threads, and each thread is pinned to a physical core based on the thread ID. For evaluation, four real world graphs are chosen to explore input diversity for the graph workloads, as summarized in Table 6.1.1. The inputs to the database workload is synthetically generated. The access to the database entries is controlled by using Zipfian distribution. It includes a parameter called *theta* to control the contention level [50]. Setting *theta* to 0.6 means that 10% of the database is accessed by the 40% of all the transactions. Similarly, using *theta* of 0.8 means that 10% of the database is accessed by the 60% of all the transactions. In this work, the *theta* value is varied from 0.6 to 0.85 with the increment of 0.5, then the average completion time is calculated using these *theta* values for performance comparison. Since *Tile-Gx72*^{TM} does not have hardware support

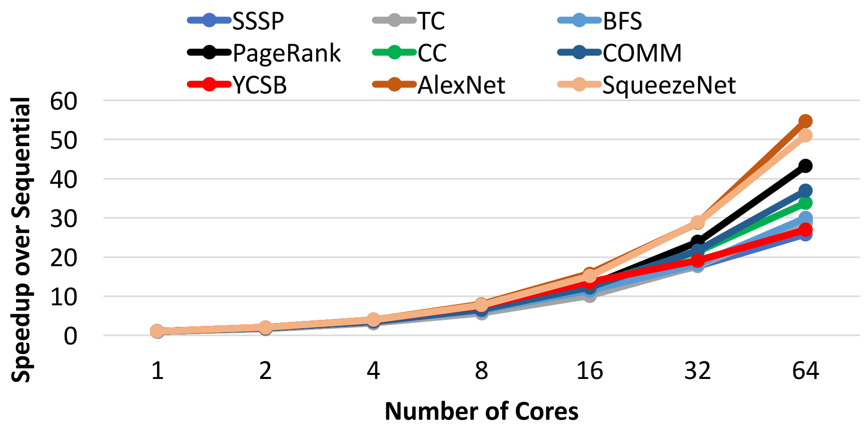


Figure 6.2.1: Average per-benchmark performance scaling results using the Spin model.

for floating point SIMD execution, the machine cannot provide state–of–the–art performance for machine learning workloads (AlexNet and SqueezeNet). However, for completeness, the scalar implementations of these benchmarks are evaluated in this study.

6.2 *Tile-Gx72^{TM}* Evaluation

6.2.1 Performance Scaling of the Benchmarks

A performance scaling study is conducted to illustrate that the spin–lock based shared memory baseline implementations scale to 64 cores on *Tile-Gx72^{TM}* platform. Each benchmark is executed by varying the core count from 1 to 64. The average speedup for each core count is plotted relative to the sequential execution of the benchmark. The sequential implementation spawns a single thread of execution that exploits all on-chip shared cache and memory controller resources. Figure 6.2.1 shows the performance scaling results for the six evaluated graph benchmarks, two machine learning benchmarks, and the database workload. As seen, all benchmarks improve performance up to 64 cores. The benchmarks with coarse–grain communication (such as PAGERANK and ALEXNET)

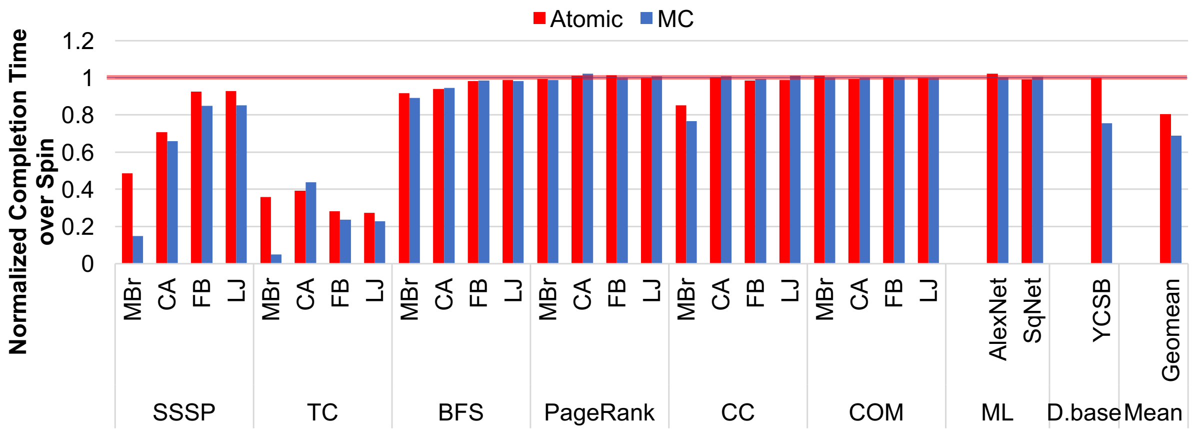


Figure 6.2.2: Completion time results under Spin, Atomic and MC models. All results are normalized to Spin.

scale better than the ones with fine–grain communication (such as TC and YCSB). Specially, the machine learning workloads scale well and provide over 50× speedup as compared to the sequential. Hence, it is not expected to get any benefit by improving thread synchronization in these workloads. In the case of the workloads with fine–grained synchronization, degradation in scaling speedup is expected as contended shared data in several graph benchmarks and the database workload leads to synchronization bottleneck. Overall, the Spin model achieves 26× to 54× performance improvement at 64 cores over sequential.

6.2.2 Performance of MC and Atomic over Spin

Figure 6.2.2 shows the normalized completion time results of Atomic and MC models over the Spin model. Atomic and MC both follow the same trends over the Spin model. There is almost no performance difference between all three synchronization models for the benchmarks with coarse–grain synchronization (PAGERANK, CC, COM, ALEXNET and SQUEEZENET) . Even though both MC and Atomic based barriers are more efficient than Spin, these benchmarks do not show any performance change since each core has a considerable amount of work between barriers. The

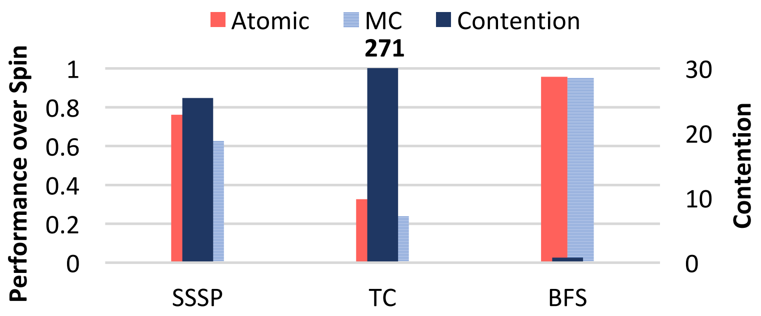


Figure 6.2.3: Contention vs. performance of MC and Atomic models for SSSP, TC, and BFS.

only exception is CC with Mouse-Brain graph because it is a relatively small graph, and CC does not involve much computation between barriers. In this case, the barrier implementation with the MC model provides the best performance.

Unlike the workloads with coarse–grain synchronization, the graph and database workloads with fine–grain synchronization (SSSP, TC, BFS, and YCSB) show some variability across different synchronization models. Here, the contention is an important metric to indicate the cases where performance can be improved using better synchronization primitives. As it is seen from Figure 6.2.2, while MC provides better execution for TC and SSSP, the completion time does not change at all in BFS. This is due to the fact that SSSP and TC involve more contended locks than BFS. Figure 6.2.3 illustrates the contention of each fine–grain graph workload with respect to their performance over Spin. *Contention* is the average number of lock–acquisitions per node in a graph, determined using per-node counters in the critical section. As observed, when contention increases, the performance obtained from MC and Atomic also escalates. Since BFS algorithm guarantees that each lock variable is acquired only once in the whole program execution, the locks are not contended, and the shared work done by each thread is very small compared to the private work. As a result, there is not much to improve with a more efficient synchronization model.

On the other hand, the Spin implementation of TC requires locking of each edge without any condition, as explained in Section 5.4.1. Therefore, the contention is higher as illustrated

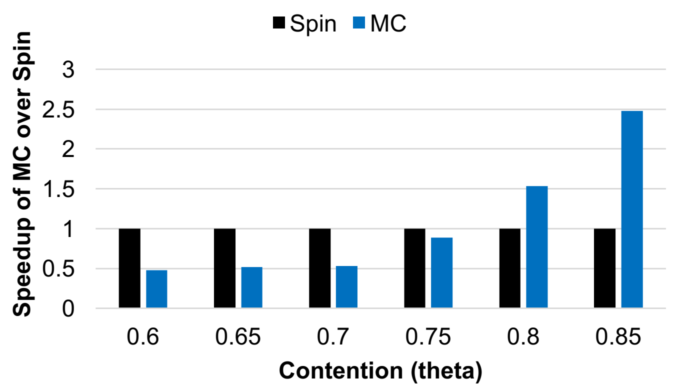


Figure 6.2.4: Contention vs. performance of MC and Spin models for YCSB workload.

in Figure 6.2.3. Consequently, the MC model significantly improves performance for all input graphs. This performance achievement mainly comes from removing the lock acquisition overheads by pinning shared data at dedicated *service cores*, and using low latency non–blocking explicit messages. TC is an ideal showcase of MC as the shared data is neither read nor written by any other core. It totally eliminates sharing of shared data with any other thread, and basically makes it private data to the *service cores*. As a result, it provides an average of 76% performance benefit over Spin.

SSSP is a benchmark where lock acquisition per node (contention) is greater than BFS, but less than TC. It has a test before getting into critical section to make sure no redundant lock acquisition is performed. However, due to its iterative nature, each lock is acquired multiple times in the program execution. Therefore, similar to TC, removing these locks with MC, and shipping critical sections to *service cores* with non–blocking messages help improve performance. On average, it yields 34% efficient program execution compared to Spin.

As discussed in Section 5.4.3, YCSB is distinctive than the graph workloads in a two ways. First of all, as discussed in Section 5.4.3, its critical sections are very complex, hence it does not have a version which is implemented using just standalone atomic instructions. The critical sections are protected using spin–locks. In addition, it involves blocking communication because each worker core expects a return condition from the service thread. Therefore, as seen in Figure 6.2.2,

25% performance benefits of MC stem from the shared data pinning, hence better data locality. Figure 6.2.4 demonstrates the performance of MC over Spin as the contention of the workload increases. As illustrated, when the contention is low meaning that less database entries are accessed by multiple threads, Spin version outperforms the MC approach. When the contention is low, the threads access to different parts of the database, hence the Spin version provides higher concurrency in both algorithmic work and the critical section work. On the other hand, the MC model needs to spare some of the cores as service thread, thus the concurrency in both service and worker thread tasks are limited. As a result, the Spin version provides superior performance. However, as the contention parameter increases, the speedup of MC over Spin reaches to $2.5\times$ at 64 cores. This is because at higher contention, the shared data structures inside the critical sections ping–pong between cores in the Spin version. Pinning the shared data structures to service threads eschews the unnecessary data bouncing, hence provides better performance.

6.2.3 Performance of MC over Atomic

In this section, the MC model is evaluated against Atomic, which is a more efficient implementation of synchronization as compared to Spin. Since the benchmarks with coarse–grain synchronization do not show much performance differences, they are not discussed further. Similarly, YCSB does not have a version with single atomic instructions. Therefore, the Atomic version is same as the Spin version. Figure 6.2.2 shows that on average the MC model accomplishes 15% better performance as compared to Atomic.

As discussed earlier, the contention in BFS is negligibly small, hence MC does not offer any additional performance. Almost all of the performance benefits stem from TC and SSSP. Both algorithms under MC show similar behavior against Atomic, except TC executing with the California road network graph, where Atomic slightly outperforms MC. The main benefits

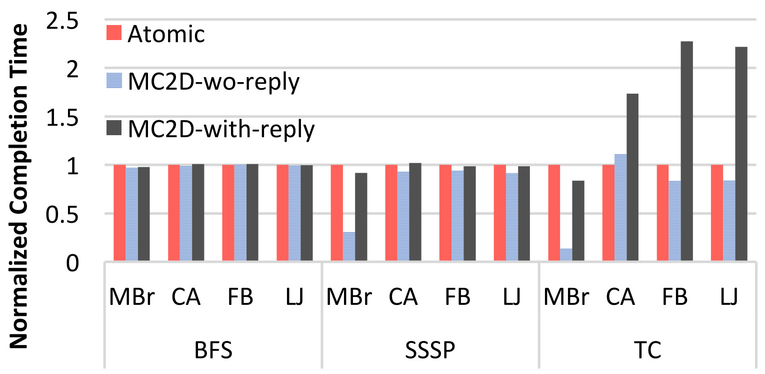


Figure 6.2.5: Normalized performance of MC models with and without reply messages against the Atomic model.

come from overlapping communication stalls with other computation. The MC model utilizes asynchronous messaging for critical section requests. Each *worker core* sends its request to the corresponding *service core* and continues to do other useful work, including subsequent requests for critical section executions. This implicitly pipelines the critical section executions and reduces the overhead of synchronization. To verify the performance advantage of non–blocking messaging aspects of the MC model, a study is performed where each *worker core* waits for an explicit reply message from the corresponding *service core* to ensure the critical section work completed before proceeding. Figure 6.2.5 illustrates the performance comparison for the default MC model without reply, and the MC model with reply. When the MC model waits for the reply, performance gets worse than the Atomic model when contention is high in the benchmark. This illustrates that fine–grain synchronization stalls benefit significantly when they are overlapped with other useful work in the *worker cores*.

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

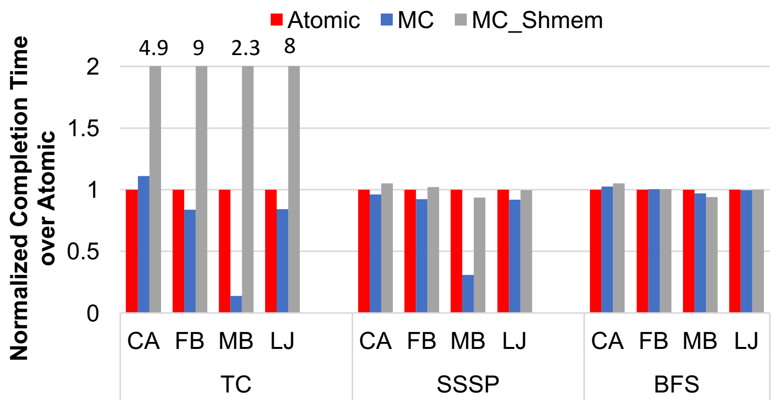


Figure 6.2.6: Normalized performance of MC versus MC_shmem against the Atomic model.

6.2.4 MC versus MC_shmem

This section discusses the results of the shared memory version of the MC model. As discussed in Section 3.2.1, the spatial MC model is implemented using the shared memory cache coherence (MC_shmem) without in–hardware explicit messaging support. Figure 6.2.6 shows the normalized completion time of default MC and MC_shmem for SSSP, TC and BFS over Atomic at 64 cores.

As shown in the figure, both MC and MC_shmem provide similar performance for BFS. As discussed earlier, this workload is not contended, hence the implementation of the critical section does not make any difference in performance at this core count. In the case of SSSP, MC_shmem's performance is almost the same with Atomic but worse than the default MC implementation. The reason for this is that the index to the shared buffer is protected with an atomic instruction, and the buffer entries bounce between the communicating cores. Therefore, it cannot perform as well as the MC model with hardware messaging, even though it also enables non–blocking communication. Contrary to the other two workloads, MC_shmem performs even worse than the Atomic model for all the inputs in TC. As discussed in Section 5.4.1, the worker threads send critical section requests for every edge of the graph in TC. Therefore, there is constant communication between worker and service threads. Even though the shared data array is pinned to the service threads and

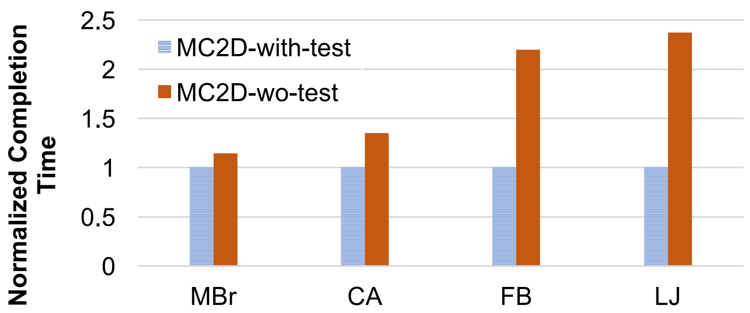


Figure 6.2.7: Performance of SSSP under MC model with and without test before sending critical section invocations.

non–blocking communication is enabled in MC_shmem, the communication via shared buffer leads to continuous cache line bouncing between worker and service threads. This leads to performance degradation when using the MC_shmem approach. Even if there is no contention on shared data, the shared buffer still ping–pongs. In addition, the concurrency on critical sections is also limited to the dedicated service threads. On the other hand, the Atomic model enjoys better concurrency when there is no contention on shared data. Thus, the Atomic model provides better performance than the shared memory version of the MC model. This study shows that low–latency hardware messaging is necessary for efficient implementation of the MC model. Consequently, it leads to superior performance as compared to the Atomic model.

6.2.5 Moving Compute and Cache Coherence

Even though the MC model accelerates synchronization on shared data, it relies on the hardware cache coherence protocol for efficient movement of cache lines between cores. This is specifically important for efficient parallel implementation of work efficient algorithms. For example, SSSP contains a test to ensure that redundant critical section executions are not performed. To implement this test, a *worker core* reads some shared data that is pinned on the *service core*. This data sharing adds coherence traffic overheads, however it prevents unnecessary work. Figure 6.2.7

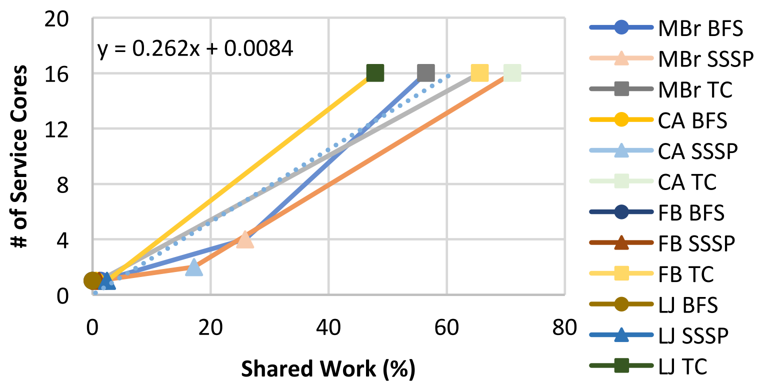


Figure 6.2.8: Correlation of service core count with shared work.

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

6.2.6 Heuristic to Determine Service Core Count

As discussed in Section 3.2.3, tuning the number of *worker* and *service cores* plays a significant role for the MC model to deliver near-optimal performance. So far this tuning is done performing an exhaustive search by varying the number of *service cores*. Due to algorithmic differences, each benchmark requires separate search, which results in expensive sweep studies. A profiling based heuristic is discussed in Section 7.2.4 to reduce the time required for the right mapping of the cores. As the MC model ships the critical section execution to dedicated *service cores*, it is expected that the time spent in critical section shows correlation with the optimal number of *service cores*. A study is conducted to show the effectiveness of the proposed heuristic. For this study, only the graph workloads with fine–grained synchronization are employed as example benchmarks. The spin–lock version of each workload–input combination is profiled and the *shared work* as discussed in Section 6.1.1 is obtained. For the same benchmark–input combinations, a service thread sweep study is performed by running the MC implementations, and the best performing service thread counts are determined. Figure 6.2.8 shows the shared work versus the best performing service thread count for each benchmark–input combination. As seen, it demonstrates strong correlation between the profiled shared work (see Section 6.1.1) and the ideal *service core* counts. As it is observed that BFS has a very small amount of shared work (less than 1%) for all four input graphs, which results in only one *service core* allocation. On the other hand, TC involves notable shared work (grater than 50%), which results in a higher number of *service cores* (16 cores). SSSP's shared work varies depending on the input graph as the convergence of the algorithm depends on the graph itself. Therefore, it requires 1–4 *service cores*. The correlation between shared work and service core count is captured with a simple linear model, as demonstrated in the figure. A linear equation serves as a heuristic to determine the service core count for a given shared work. The heuristic is employed to find the number of service cores for all the benchmark-input combinations, and the result is compared to a service core count determined using an exhaustive search. It is observed that the performance of the heuristic is within 3% of the exhaustive method of determining the right *service core* count.

6.2.7 Implications of Cores Scaling

The MC model is expected to improve synchronization bottleneck in the on-chip network as the core count increases. Therefore, a core scaling study for all three synchronization models is conducted

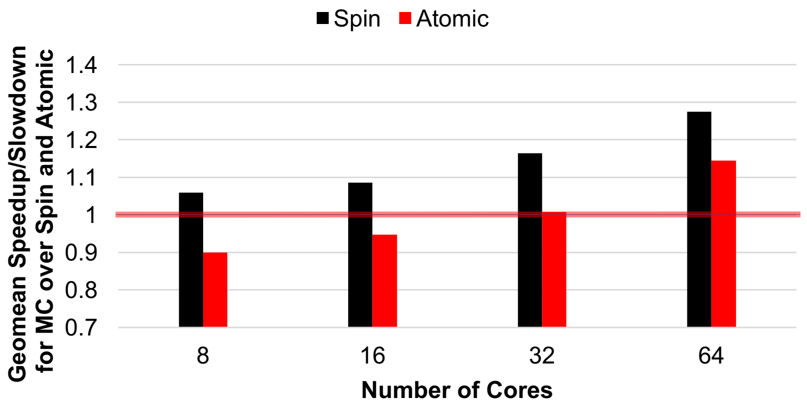


Figure 6.2.9: Average performance scaling results of MC compared to Spin and Atomic models.

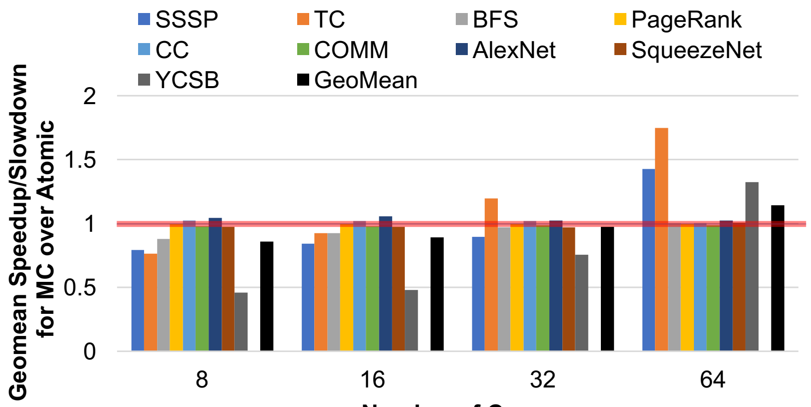


Figure 6.2.10: Average per-benchmark performance scaling results of MC over the Atomic model.

to investigate the impact of core count on performance. All benchmark–input combinations are executed to completion using 8, 16, 32, and 64 cores. Since YCSB does not have an Atomic version, the Spin implementation results are used to calculate the average speedup over Atomic. Figure 6.2.9 shows the average speedup of MC over Spin and Atomic models as the core count is increased. The speedup of MC over both models gets higher with the increase in core count. While MC performs more efficient than Spin (even at 8 cores), the performance over Atomic model diminishes when using less than 32 cores. Figure 6.2.10 demonstrates more detailed view of the core scaling study to investigate the reason behind this performance drop. The performance degradation mainly stems

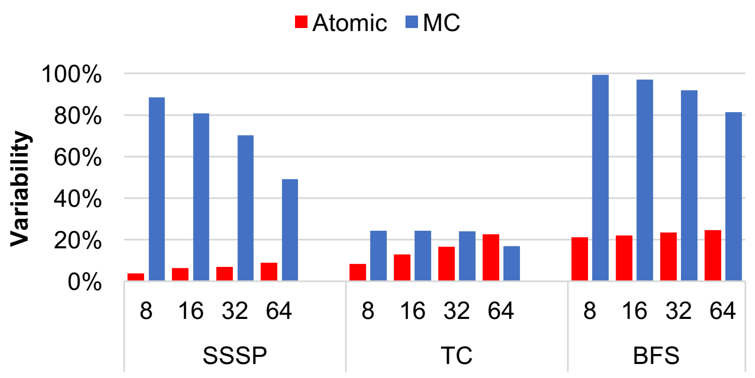


Figure 6.2.11: Load imbalance of Atomic and MC models at various core counts; normalized to respective 8 cores results.

from benchmarks with fine–grain synchronization. As SSSP, TC, BFS, and YCSB require *service core(s)* for critical section work, it is hard to load balance the worker and service cores as the total core count goes down. To demonstrate load imbalance differences between Atomic and MC models, a study is conducted and load imbalance is measured as explained in Section 6.1.1. Only graph workloads are utilized as example benchmarks in this study. Figure 6.2.11 demonstrates the load imbalance for both MC and Atomic models at different core counts. The Atomic model observes less than 20% variability in instruction count, whereas the MC model incurs a much higher variability. This stems mainly from the fact that *service cores* execute much fewer instructions than the *worker cores*, specially in SSSP and BFS. SSSP generally requires one or two *service cores*, while BFS only needs a single *service core*. However, these cores have much less executed instructions than *worker cores*. Sparing 1 or 2 cores out of 64 cores does not hurt performance, even if there is load imbalance between *worker* and *service cores*. However, at lower total core counts, this imbalance shows up in performance degradation. Consequently, performance declines as the core count goes down for benchmarks with fine–grain synchronization.

6.2.8 Summary of Tile-Gx72 Evaluation

In this chapter, the novel moving compute to data MC model is evaluated against state-of-the-art shared memory synchronization models using graph, machine learning, and database workloads on the commercial Tilera TILE-Gx72 multicore machine. By pinning shared data to dedicated cores, the MC model improves data locality. In addition, it is shown that it overlaps communication with computation by utilizing non–blocking messages. The in-hardware messages also enable fast communication without bouncing cache lines between the communicating cores. The results show that MC model improves performance of the evaluated benchmarks by an average of 34% over Spin, and 15% over the Atomic model. Since the Tilera machine contains only 72 cores, further evaluation of the MC model at 1000–cores scale is conducted using a state–of–the–art multicore simulator in the next chapter.

Chapter 7

Evaluation on Simulator at 1000–cores

The study presented in the previous chapter illustrates that as the core count increases, the performance of the proposed model improves over the traditional synchronization primitives. However, the number of cores in Tilera machine is only 72, hence further analysis is required to study the performance scaling of the MC model up to 1000–cores. Since the overheads of shared memory based synchronization primitives gets expensive as the network becomes larger, higher core count study reveals the limits of the proposed model. Therefore,the *Tile-Gx72*^{TM} is supplemented with a multicore simulation environment, and the proposed MC model is evaluated up to 1000–cores.

Architectural Parameter	**Simulator**
Number of Cores	up to 1024 @ 1 GHz
Compute Pipeline per Core	In–Order, Single–Issue
Memorgy Subsystem	
L1–I Cache per core	8-32 KB, 4–way Assoc., 1 cycle
L1–D Cache per core	8-32 KB, 4–way Assoc., 1 cycle
L2 Inclusive Cache per core	16-256 KB, 8–way Assoc.
Directory Protocol	Invalidation–based MESI ACKwise$_4$ [51]
Num. of Memory Controllers	4 to 16
DRAM Bandwidth per Controller	10 GBps
Electrical 2–D Mesh with XY Routing	
Hop Latency	2 cycles (1–router, 1–link)
Contention Model	Only link contention (Infinite input buffers)
Flit Width	64 bits
Explicit Communication	
Receive queue per core	2.4KB

Table 7.1.1: Architectural parameters for evaluation.

7.1 Evaluation Methodology

7.1.1 Multicore Simulator

Simulator Setup

The proposed architecture (cf. Section 4) is implemented using an in–house industry–class simulator and the associated RISC–V tool chains. The simulator utilizes an Architecture Description Language (ADL) [52] for functional model implementation, which in turn drives the performance models. Table 7.1.1 summarizes the architectural parameters of the simulated system. Similar to *Tile-Gx72^{TM}*, 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 $21MB$ 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 (160 Gbps) are utilized at 1024 cores. Single threaded cores are utilized in the default mode for spatial MC, as well as spin and atomic instruction based synchronization models. However, two threads per core are evaluated for the temporal MC model implementation.

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. The programs are then compiled using the RISC-V compiler and the assembly code is obtained. The assembler created using the ADL is employed to generate the object files using the assembly code. Finally, the simulator linker generates the binary file to be executed by the simulator.

Performance Models

The performance models used in the simulator are ported from the Graphite multicore simulator [53]. 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 [54] [55]. It also includes the contention delays. In addition, the explicit messaging instructions, and the related protocol overheads are integrated into the performance models.

McPat [56] 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 [57]. The send and receive queues are also modeled in addition to other components of the core energy. Moreover, DSENT [58] toolchain is deployed to obtain the network-on-chip per event energy numbers.

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-Gx72TM* 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.

7.1.2 Benchmarks and Inputs

Table 7.1.2 shows the six graph benchmarks from the CRONO [8] 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 to ensure state–of–the–art implementations. These benchmarks are ported using Spin, Atomic and MC models. For all models, Pthreads library is used to spawn threads, and each thread is pinned to a physical core based on the thread ID. For the temporal implementation of MC, two threads are utilized, and the threads are again pinned to their respective hardware contexts. For evaluation, two real world graphs with uniform weights are

Benchmark	**Input Dataset**
Graph Analytics (CRONO [8])	
PAGERANK, TRIANGLE COUNTING COMMUNITY DETECTION, BFS CONNECTED–COMP, SSSP	California Road Network [59] Facebook [49]
Machine Learning	
CNN-ALEXNET [35] CNN-SQUEEZENET [40]	ImageNet [60]

Table 7.1.2: Problem sizes for parallel benchmarks.

chosen to explore input diversity in graph workloads, as summarized in Table 7.1.2. For machine learning workloads an image from ImageNet dataset is classified.

7.1.3 Configurations

1. **Spin:** This is the baseline system which relies on spin locks to implement both fine and coarse–grained synchronization.

2. **Atomic:** This model utilizes standalone atomic instructions for both fine and coarse–grained synchronization.

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 3.2.2).

4. ***MC_shmem:*** Moving computation to data model with spatial distribution of *worker* and *services* threads implemented using shared memory cache coherence support. This version is utilized only for fine–grained synchronization (cf. Section 3.2.1).

5. ***MC_tmp:*** Moving computation to data model with temporal distribution of *worker* and

services threads. This configuration is utilized only for fine–grained synchronization (cf. Sec 3.2.3).

7.2 Simulator Evaluation at 1000-cores Scale

The core scaling results for the spatial MC model are first compared to Spin, Atomic, and MC_shmem models using the *Tile-Gx72^{TM}* machine and the simulator. The detailed higher core count evaluations ($>$ 64 cores) are presented using the simulation environment. After the core scaling study, the performance and dynamic energy evaluations of MC with respect to Spin and Atomic are conducted at 512–cores. Furthermore, detailed scaling study of Spin, Atomic and MC models are presented.

7.2.1 Core Scaling on TILE-Gx72 and the Simulator

Figure 7.2.1 shows the average speedup of the spatial MC model over the Spin, Atomic and MC_shmem models across all the benchmark–input combinations as the core count increases. While the core count is varied from 8 to 64 in *Tile-Gx72^{TM}*, it is scaled up from 8 to 1024 in the simulation environment. There are some noteworthy differences between simulated architecture and the Tilera *Tile-Gx72^{TM}*. First, each tile in Tilera utilizes a VLIW core which contains three parallel pipelines that do not have support for explicit floating point units. On the other hand, the simulator deploys in-order single-issue RISC–V pipelined cores with support for 16-bit 4-way SIMD instructions. Second, *Tile-Gx72^{TM}* 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

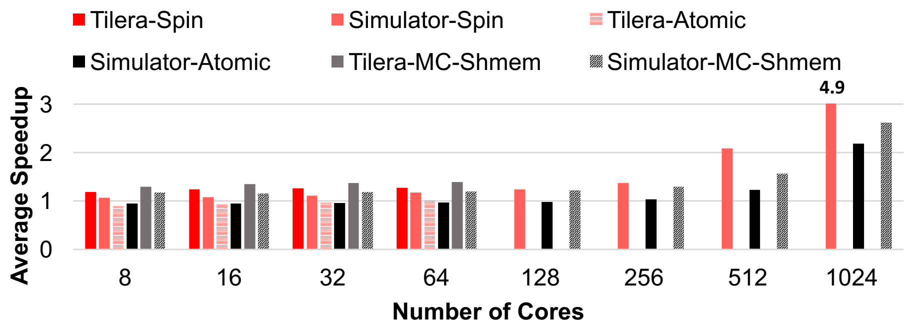


Figure 7.2.1: Average speedup of spatial MC over Spin and Atomic models as the core counts increase.

are very similar between *Tile-Gx72^{TM}* and the simulated machine at 8 – 64 core counts.

The relative performance of spatial MC with respect to Spin, Atomic, and MC_shmem models improves as the core count goes up in both TILE-Gx72 and in the simulation environment. The MC model outperforms Spin at all core counts since it does not suffer from instruction retries and cache line ping-pongs. Similarly, MC is also constantly more efficient than MC_shmem at all the core counts. As discussed in Section 3.2.1, the communication between worker and service threads are realized using a shared buffer per service thread. The cache lines of the shared buffer constantly bounce between worker and service threads whenever a message is being sent. Hence, even at smaller core counts the performance is worse than MC. Atomic model fares well as it provides more efficient execution of critical code sections. At smaller core counts, Atomic outperforms MC by more than 10%. MC closes the gap with the increase in core count, and provides comparable performance at 64 cores in both TILE-Gx72 and the simulator. This is mainly as a result of increased traffic in the on-chip network due to cache line ping-pongs in the Atomic model. However, MC eschews unnecessary ping-pongs but it suffers from the challenge to load balance work between the *worker* and *service* threads at lower core counts. The relative performance of MC significantly improves beyond 256 cores, and delivers significant advantages at both 512 and 1024 cores. Although not shown here, the Atomic model delivers performance scaling for all benchmark–

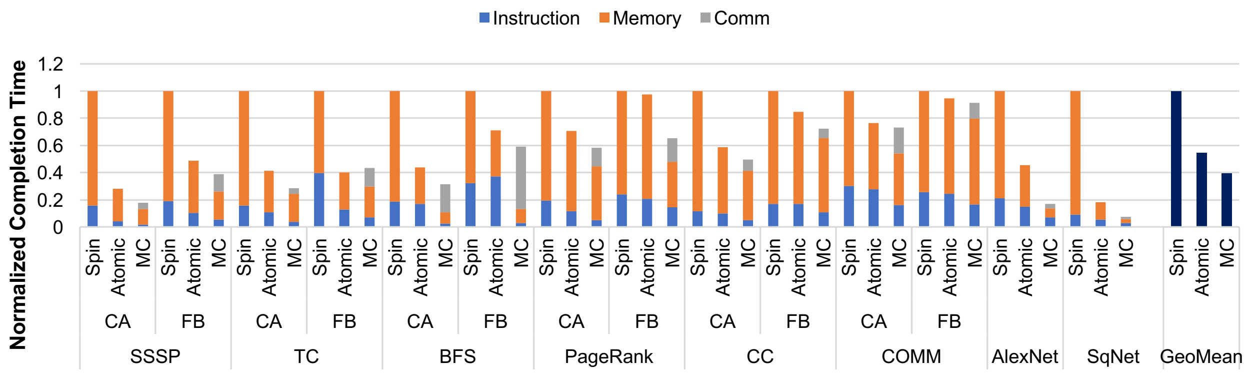


Figure 7.2.2: Completion time results for Spin, Atomic, and MC at 512 cores; all normalized to Spin.

input combinations at 512 cores, but not at 1024 cores. However, the spatial MC model consistently delivers performance scaling at both 512 and 1024 cores. The remaining evaluation discusses the performance and energy results of Spin, Atomic and MC models for 512 cores followed by a detailed sensitivity analysis of all synchronization models for various cores counts. Since the MC_shmem is only utilized for fine–grained synchronization, the detailed study of MC against MC_shmem is discussed separately in Section 7.2.6.

7.2.2 Evaluation of 512–cores Multicore

Performance Evaluation

Figures 7.2.2 illustrates the performance results of Spin, Atomic, and spatial MC implementations of graph and machine learning benchmarks at 512–cores setup. 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 completion time. The geometric mean shows that the MC model outperforms Spin by 60%, and Atomic by 27%.

Graph Workloads with Fine–grained Synchronization: SSSP, TC and BFS significantly benefit from Atomic as it removes the locks and uses a single atomic instruction to implement critical sections. As a result, instruction counts and memory stalls are drastically alleviated in these benchmarks. However, Atomic 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 MC. TC does not involve any test to eliminate redundant critical section executions as discussed in Section 5.4. So it requires atomic update for each neighbor. Therefore, it requires higher concurrency in the execution of the critical code section. In addition, Facebook is a sequential graph which does not have many common neighbors between the graph chunks, hence the shared data bouncing is very limited. Therefore, TC with Facebook graph is expected to benefit from higher concurrency. As Atomic provides higher concurrency for the critical code sections, and MC limits parallelism for the *service threads*, Atomic yields slightly better completion time. CA graph, on the other hand, contains more random connections, which leads to shared data bouncing in the Atomic model. Therefore, MC enhances the execution time as a result of pinning the shared data, and overlapping the communication latency using non–blocking send instructions. In addition, the MC barrier eschews instruction retries and thus improve instruction stalls. However, it incurs the communication stalls due to the explicit messaging instructions.

On the contrary to TC, BFS algorithm guarantees that each critical section is executed at most once in the whole 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 in terms of concurrency requirement. It involves

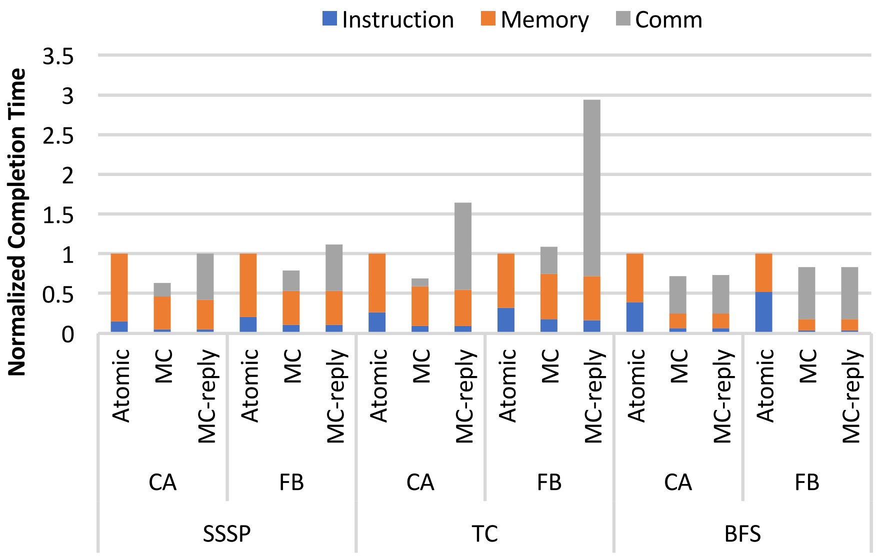


Figure 7.2.3: Performance comparison of MC-reply with default MC and Atomic at 512 cores; all normalized to Atomic.

a test to prevent the 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 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 5.4.1. Hence, the main advantage of MC over atomic instructions for this workload 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 for the critical section requests. 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 7.2.3 shows the performance comparison of the default non-blocking MC with Atomic and MC-reply. 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 the concurrency is limited with MC as compared to Atomic. On the other hand, BFS does not show any change in the completion time. This suggests that BFS does not benefit from non–blocking send messages because the *workers* temporally send requests after significant local computations.

Graph Workloads with Coarse–grained Synchronization: For PAGERANK, CC and COMM, Atomic reduces the instruction count by replacing the lock in the barrier implementation with an atomic fetch–and–add instruction. The decrease in the instruction count depends on whether the barrier variable is contended or not. If there is load imbalance and the threads reach the barrier at different timestamps, utilizing atomic instruction does not help much in the performance. It makes updating the barrier variable more efficient, however, it still requires spinning until all the threads arrive at the barrier. On the other hand, if the 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. On the other hand, MC eliminates instruction retries and expensive shared barrier variable ping–pongs. The spinning cost is replaced with more efficient explicit communication which leads to enhanced performance compared to both Spin and Atomic. COMM is the only workload that MC does not have noteworthy improvement over Atomic. This is due to the fact that it has more load imbalance than other two workloads. Even though MC reduces the instruction count as a result of more efficient barrier, the load imbalance between the

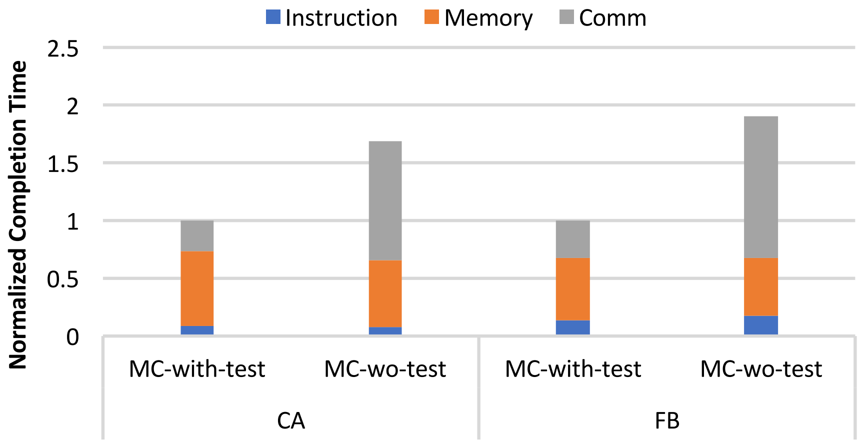


Figure 7.2.4: Performance comparison of SSSP under MC with and without test before critical section request at 512 cores; all normalized to MC.

participating threads lead to compensating communication stalls.

Machine Learning Workloads: The MC models yields significant performance improvements for the two machine learning workloads. For ALEXNET, it utilizes the fine–grain parallelization as explained in Section 5.4.2, while SQUEEZENET is realized using coarse–grain parallelization similar to the Spin and Atomic models. 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 the performance over the Spin model. However, Atomic also suffers from bouncing the shared barrier variable between cores as it is very expensive when the core count is high. The MC model further 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

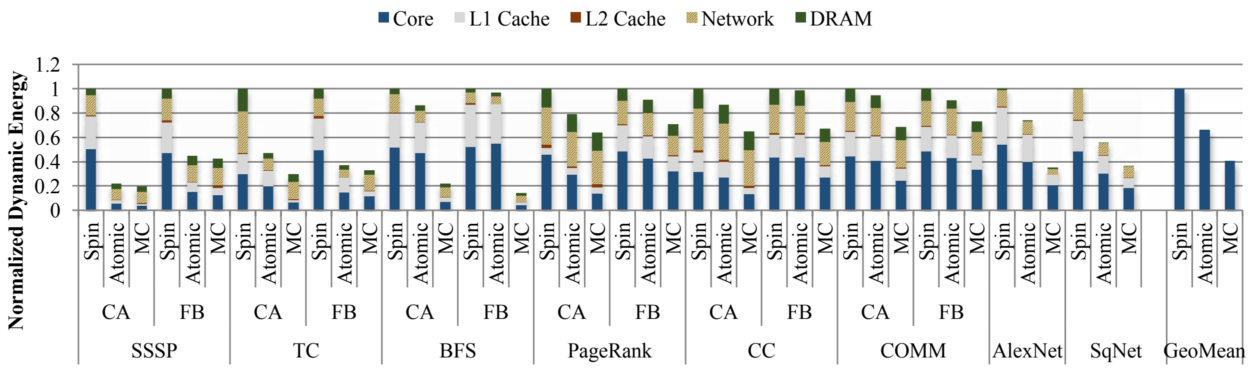


Figure 7.2.5: Dynamic energy results for Spin, Atomic, and MC at 512 cores; all normalized to Spin.

(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 the 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 superior performance. To evaluate this hypothesis, the SSSP benchmark is also implemented without the test, and its performance is compared against the default MC implementation with test. Figure 7.2.4 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.

Dynamic Energy Evaluation

Figure 7.2.5 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 7.2.2. 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 and the communication stalls in MC do not contribute to the dynamic energy as the core stays idle during this stall time.

The figure also shows the results for graph workloads with coarse–grain synchronization. As seen, the dynamic energy again follows very similar trend with the performance results. 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 the dynamic energy (as also mentioned in BFS case). On the other hand, 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. 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. On the other hand, 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.

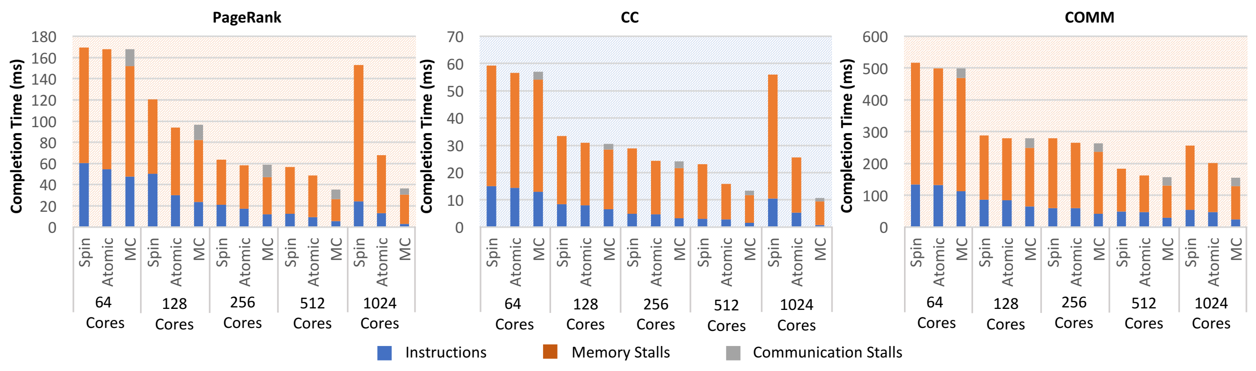


Figure 7.2.6: Core scaling results for Spin, Atomic, and MC implementations of PAGERANK, CC and COMM.

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

Figures 7.2.6, 7.2.7 and 7.2.8 show the performance scaling results of all benchmarks as the number of cores per multicore chip are increased from 64 to 1024. The total on-chip cache capacity is kept nearly constant at $\sim 22MB$, that is to say that per tile cache sizes are scaled down as more core are integrated on-chip. The results of all benchmarks are average of California Road Network and Facebook graphs and reported as raw completion times in each figure.

Figure 7.2.6 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, even the Atomic drastically slows down in its performance scaling as core counts are increased from 512 to 1024. On the other hand, the spatial MC model achieves superior scaling compared to both Spin and Atomic models as a result of its efficient barrier implementation. Even though it does not improve performance at 1024 cores compared to 512 cores for PAGERANK and COMM, the completion time does not get worse. The reason for

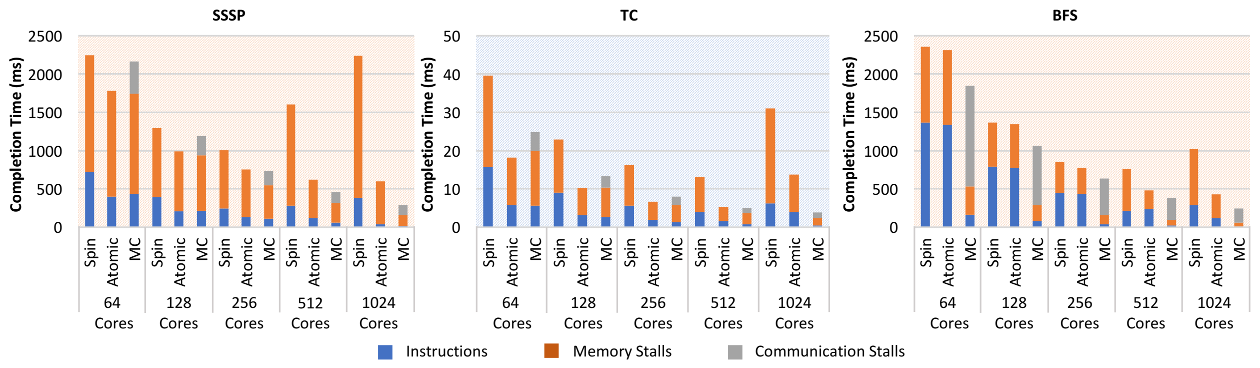
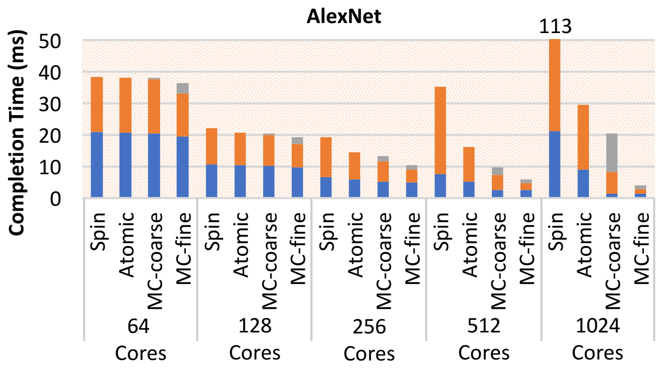


Figure 7.2.7: Core scaling results for Spin, Atomic, and MC implementations of SSSP, TC and BFS.

not scaling beyond 512 is mainly due to the fact that the memory stalls do not scale for these two benchmarks. There are two factors that contribute to the memory bottleneck. The first one is that as the network gets larger, the data access latency increases. The second reason is that PAGERANK and COMM make more memory accesses as compared to CC. As a result, the memory stalls become bottleneck beyond 512 cores for these two workloads.

Figure 7.2.7 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 for 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 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 1024 cores, and provides $2.2\times$ better performance than the Atomic model. 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. On the other hand, the MC model provides superior performance up to 1024 cores, and offers $2.4\times$ speedup over the best performing Atomic model implementation. Unlike SSSP, barrier synchronization is the biggest bottleneck in BFS since the locks are not contended,



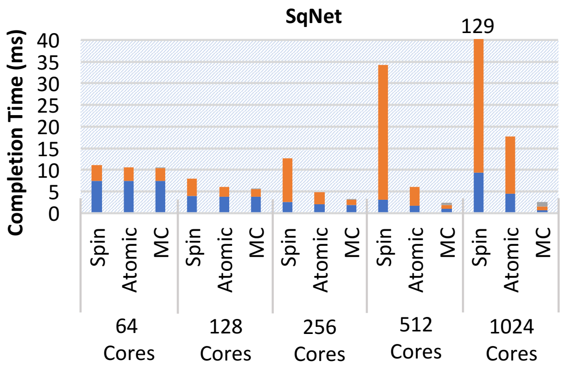


Figure 7.2.8: Core scaling results for Spin, Atomic, and MC implementations of ALEXNET and SQUEEZENET.

as discussed in Section 7.2.2. Subsequently, providing faster barrier with the MC model pushes scaling to thousand cores. Contrary to other two workloads, the Spin version of TC scales to 512 cores. However, due to overheads of per node lock acquisition, its performance is worse than both Atomic and MC models. The reason for better scaling compared to SSSP and BFS is that the Spin model benefits from concurrency when the shared data is not contended. Since these graphs are sparse, there is not much contention, and thus the Spin model scales. The Atomic model surpasses the Spin model consistently, and provides more efficient completion time but it also does not scale beyond 512 cores. Even though the MC model looses to Atomic at 64 cores for SSSP and TC as a result of better concurrency, it yields more effective completion times at 512 and 1024 cores. At higher core count, as the network latency boosts, the overlapping of communication stalls becomes even more important. Hence, the MC model leads to better scaling than Atomic, even though the Atomic model provides superior concurrency.

Figure 7.2.8 shows the scaling of both machine learning workloads, ALEXNET and SQUEEZENET. The Spin model does not even scale to 256 cores. As both these workloads are implemented using 4–way SIMD with 16–bit floating point capabilities per core, it significantly reduces both instruction counts and the memory stalls between barriers. Hence, synchronization at the end of each layer becomes important. As SQUEEZENET contains more layers, and thus more barriers, it experiences more performance degradation with the increase in core counts for the Spin model. The Atomic model achieves performance scaling up to 256 cores, however beyond that performance starts declining for both benchmarks. The MC model, on the other hand, scales to 1024 cores for SQUEEZENET with more efficient barrier synchronization. Similarly, the MC model also helps achieve superior performance for ALEXNET up to 1024 core. However, as discussed in Section 5.4.2, it has two implementations for the spatial MC model. One is naive implementation which only replaces the barrier, which is called coarse–grain MC (MC–coarse). This implementation of ALEXNET does not scale to 1024 cores as it suffers from load imbalance due to limited concurrency. By using the fine–grain strategy discussed in Section 5.4.2, both imbalance and concurrency challenges are solved without sacrificing data reuse. Hence, fine–grain MC (MC–fine) provides performance improvements up to 1024 cores, and offer $3.5\times$ speedup over the best scaling Atomic implementation.

7.2.4 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, it results in $3\times$ 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, a profiling driven heuristic is proposed to determine the near-optimal service thread count.

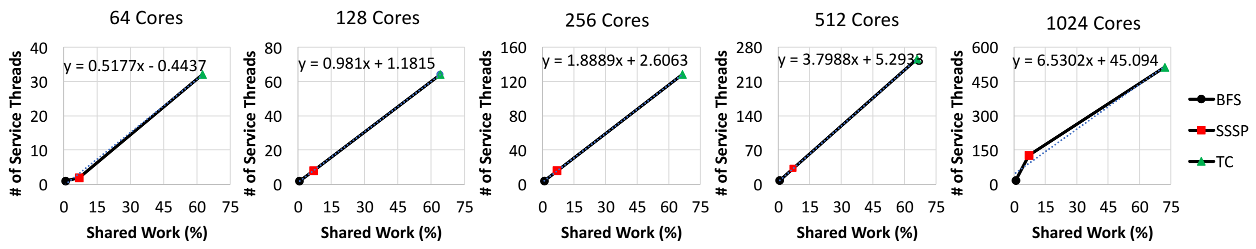


Figure 7.2.9: Correlation of service core count with shared work for SSSP, TC and BFS at different core counts.

As discussed in Section 3.2.3, 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 are profiled to obtain the percentage shared work each thread performs at 64, 128, 256, 512 and 1024 cores. Also, for each core count, a sweep study is conducted to obtain the best performing *service thread* count. Figure 7.2.9 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 the benchmarks. This suggests that by profiling the Spin version, one can easily determine the required number of *service threads*. For example, at 512 cores, SSSP has 7% of 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 3.2.3, the *worker* to *service thread* ratio is limited to at most 50%. Therefore, in some cases such as TC, the shared work is more than 50% of the total completion time. Hence, more than half of the cores are not 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.

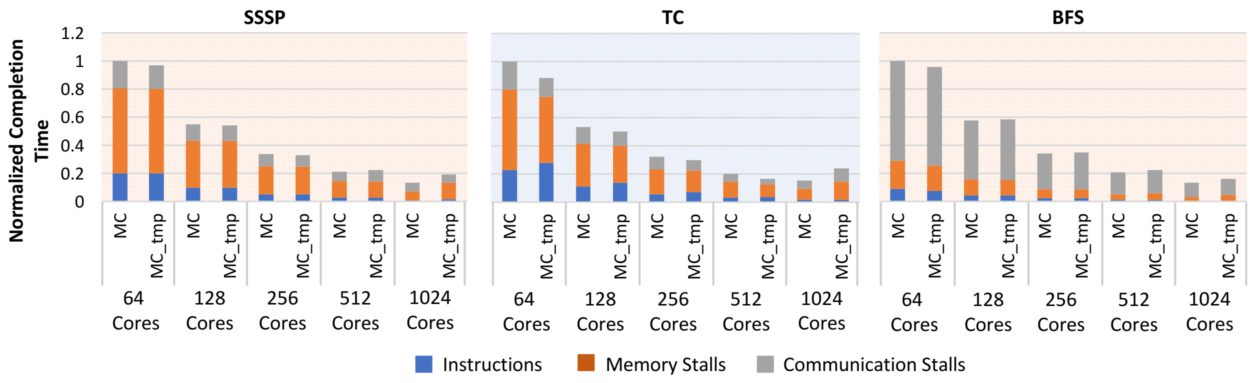


Figure 7.2.10: Normalized core scaling results of MC, and MC_tmp; all normalized to MC at 64 cores.

7.2.5 Spatial versus Temporal MC Model

This section evaluates the spatial MC model against the MC_tmp model which eliminates the need for tuning the service thread count by utilizing temporal mapping of *service* and *worker threads* in the same core, as explained in Section 3.2.3. Figure 7.2.10 demonstrates the results of spatial (default) MC and temporal MC (MC_tmp) for three graph benchmarks with fine–grain communication. The presented results are the average of California road network and the Facebook graphs, and the results are normalized to the spatial MC model at 64 cores.

The temporal implementation improves performance compared to spatial MC at lower core counts (see 64 cores in the figure). 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. Therefore, it slightly improves performance over the spatial implementation. However, as the number of cores increases,

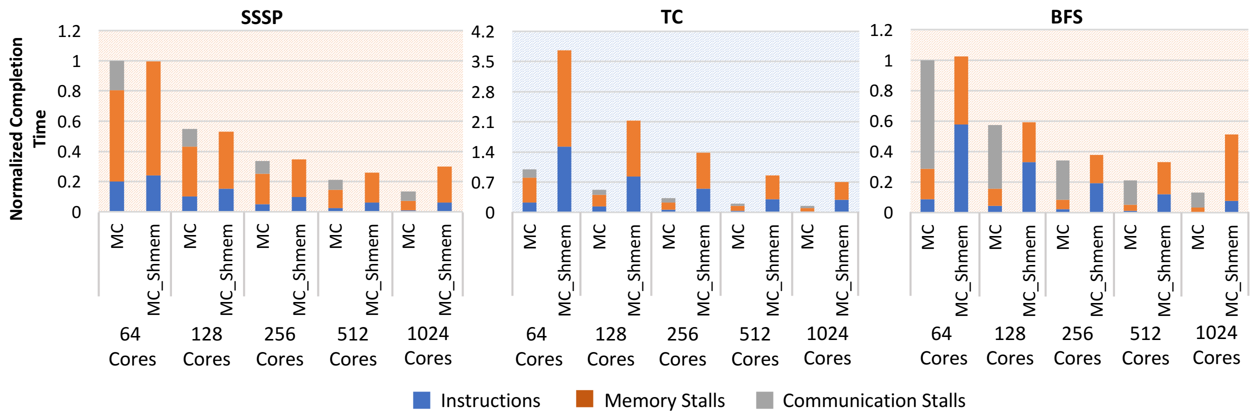


Figure 7.2.11: Normalized core scaling results of MC, and MC_shmem; all normalized to MC at 64 cores.

the benefits of temporal implementation 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 spatial approach. In addition, $2\times$ more threads participate in the barrier with MC_tmp, and this enlarges the barrier overheads specially at higher core counts where the barrier performance is important. Moreover, assigning two different tasks in the same core stress the private cache capacity, which results in higher memory stalls as suggested by the data in the figure. Consequently, employing MC_tmp 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 implementation of the MC model. In addition, it also requires two hardware contexts per core, and context switching policy logic that takes explicit messaging into account. On the other hand, the spatial MC model is lightweight and flexible in managing the various *worker* and *service threads* in the system.

7.2.6 Evaluation of MC against MC_shmem

This section discusses the results of the shared memory version of moving computation to data model. As discussed in Section 3.2.1, the spatial MC model is implemented using the shared memory cache coherence (MC_shmem) without in–hardware explicit messaging support. Figure 7.2.11 shows the normalized completion time of default MC and MC_shmem for SSSP, TC and BFS at different core counts. The presented results are the average of California road network and the Facebook graphs, and the results are normalized to MC at 64 cores.

The figure shows that the performance scaling of BFS follows almost the same trend with Atomic in Figure 7.2.7. As discussed in Section 7.2.2, BFS is not contended, hence the implementation of the critical section does not make any difference in the performance. The most of the performance benefit comes from the explicit messaging based barrier implementation. Therefore, at higher core counts, the performance of MC_shmem degrades due to the ping–pongs of the shared barrier variable. In the case of SSSP, MC_shmem provides similar performance with MC up to 256 cores. At 512 cores, the performance of MC_shmem is $1.22\times$ worse than the MC with explicit messaging. This is better than the completion time of Atomic at the same core count ($1.35\times$ 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 barriers are also becoming a limiting factor at these core counts. At 1024 cores, the ping–ponging affect becomes worse, hence more than $2\times$ performance difference is observed. TC is the only workload in which MC_shmem is always worse than MC at all core counts. Unlike other two workloads, MC version of TC prevents ping–ponging of the 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 actual shared data. In the case of Atomic, if there is no contention on the shared data, there is no bouncing, hence it can benefit from elevated concurrency on the critical section. As a result, it can provide better performance. On the other hand, as a result of constant ping–ponging of the shared buffer, the performance of MC_shmem is always worse than both Atomic and MC. The results show that on average the MC model with explicit messaging is $2.3\times$ 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.

Chapter 8

Conclusion

This thesis proposes a novel moving compute to data model for accelerating synchronization on a 1000–cores 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. This enables to efficiently pin shared data to dedicated cores, and eschews unnecessary cache line ping–ponging. In addition, by allowing non–blocking communication for workloads that do not have strict consistency requirements, the proposed model hides communication latency by overlapping it with computation and other stalls. The applicability of the proposed model is shown by implementing various workloads with different synchronization requirements from graph analytics, machine learning and database domains.

This thesis includes two sets of evaluations for the proposed synchronization model. First, *Tilera*® *Tile-Gx72*™ multicore platform is employed to implement MC model using its core–to–core messaging network, and evaluate the proposed model up to 64 cores. The results show that the proposed model provides on average 16% and 34% performance benefit over atomic instruction and spin–lock based synchronizations, respectively. The evaluations at 64 cores suggest that for the graph workloads, the most notable performance improvement of the MC approach stems from the workloads with fine–grained synchronization, and utilizing non–blocking communication for the critical section requests is the most significant contributor to the superior performance. For the database workload, on the other hand, pinning the shared data to service cores is shown to prevent expensive cache line ping–pongs under high contention, and provide enhanced performance as the core count increases.

Since the Tilera machine only contains 72 cores, a RISC-V based multicore simulation environment is deployed for further analysis at 1000–cores scale. The proposed synchronization model is evaluated against atomic instructions and the traditional spin-lock based synchronization primitives for the graph analytics and machine learning benchmarks. The experimental results show that the spatial MC model scales performance up to 1000 cores while traditional shared memory approaches do not scale beyond 512 cores. It offers an average of 60% improvement over the lock based synchronization, and 27% better performance over atomic instruction based synchronization at 512 cores. The proposed profiling based heuristic is also evaluated up to 1000 cores and shown to be effective regardless of the core count. Furthermore, the MC model achieves an average of 39% efficiency on dynamic energy as compared to the atomic instruction based synchronization. Moreover, the proposed spatial MC model is realized using shared memory cache coherence without in–hardware explicit messaging, and a comparative study is conducted to show that in–hardware explicit messaging is required for efficient implementation of the MC model. Finally, a temporal implementation of the MC model is implemented, and a scaling study is conducted. The study shows that while at lower core counts the temporal implementation provides better performance, the spatial MC model outperforms the temporal as the core count approaches to 1000–cores.

Tilera evaluations up to 64 cores, and the higher core count evaluations using the RISC-V multicore simulator show that the proposed moving computation to data approach is a promising synchronization model for future multicores at the 1000–cores scale.

Bibliography

[1] Brent Bohnenstiehl, Aaron Stillmaker, Jon J Pimentel, Timothy Andreas, Bin Liu, Anh T Tran, Emmanuel Adeagbo, and Bevan M Baas. KiloCore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits*, 52(4):891–902, 2017.

[2] J Balkind, M McKeown, Y Fu, T Nguyen, Y Zhou, A Lavrov, M Shahrad, A Fuchs, S Payne, X Liang, M Matl, and D Wentzlaff. OpenPiton: An Open Source Manycore Research Framework. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2016.

[3] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270. ACM, 1993.

[4] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[5] Travis Craig. Building FIFO and priorityqueuing spin locks from atomic swap. Technical report, Citeseer, 1993.

[6] Michael L Scott and William N Scherer. Scalable queue-based spin locks with timeout. In *ACM SIGPLAN Notices*, volume 36, pages 44–52. ACM, 2001.

[7] Robert W Wisniewski, Leonidas I Kontothanassis, and Michael L Scott. Scalable spin locks for multiprogrammed systems. Technical report, ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE, 1993.

[8] M. Ahmad, F. Hijaz, Qingchuan Shi, and O. Khan. CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores. In *Workload Characterization (IISWC), 2015 IEEE Int. Symp. on*, pages 44–55, Oct 2015.

[9] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 33–48. ACM, 2013.

[10] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia L Lawall, Gilles Muller, et al. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *USENIX annual technical conference*, pages 65–76, 2012.

[11] R. Harting and William Dally. On-Chip Active Messages for Speed, Scalability, and Efficiency. *TPDS*, 99(PrePrints), 2014.

[12] H. Dogan, F. Hijaz, M. Ahmad, B. Kahne, P. Wilson, and O. Khan. Accelerating Graph and Machine Learning Workloads Using a Shared Memory Multicore Architecture with Auxiliary Support for in-Hardware Explicit Messaging. In *IPDPS, 2017*, 2017.

[13] Halit Dogan, Brian Kahne, and Omer Khan. QUARQ: A Novel General Purpose Multicore Architecture for Cognitive Computing. In *TECHCON*, 2017.

[14] Halit Dogan, Masab Ahmad, José A Joao, and Omer Khan. Accelerating Synchronization in Graph Analytics using Moving Compute to Data Model on Tilera TILE-Gx72. 2018.

[15] Halit Dogan, Masab Ahmad, Brian Kahne, and Omer Khan. Accelerating Synchronization using Moving Compute to Data Model at 1000-core Multicore Scale. *ACM Trans. Archit. Code Optim.*, 2019.

[16] John Kubiatowicz and Anant Agarwal. The Anatomy of a Message in the Alewife Multiprocessor. In *ICS*, 1993.

[17] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *ISCA*, 1992.

[18] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proc. of the 42Nd Annual Int. Symp. on Computer Architecture*, ISCA '15. ACM.

[19] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-grain Scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ACM.

[20] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. Brown III, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, September 2007.

[21] B. C. Lam, A. D. George, and H. Lam. TSHMEM: Shared-Memory Parallel Computing on Tilera Many-Core Processors. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 325–334, May 2013.

[22] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-core Architectures. In *Proc. of the 14th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS. ACM, 2009.

[23] D. Tiwari, J. Tuck, Solihin Y, and S. Lee. HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. *2011 IEEE 17th Int. Symp. on High Performance Computer Architecture (HPCA)*, 2011.

[24] Y. Wang, R. Wang, A. Herdrich, J. Tsai, and Y. Solihin. CAF: Core to Core Communication Acceleration Framework. In *Proc. of the 2016 Int. Conf. on Parallel Architectures and Compilation*, PACT '16. ACM.

[25] Dhruba Borthakur et al. HDFS architecture guide. *Hadoop Apache Project*, 53:1–13, 2008.

[26] William Gropp. MPICH2: A new start for MPI implementations. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 7–7. Springer, 2002.

[27] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2014.

[28] Tilera Corporation. UG527-Application Libraries Reference Manual, 2014.

[29] Bradford Nichols, Dick Buttlar, Jacqueline Farrell, and Jackie Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[30] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[31] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Acm sigops operating systems review*, volume 42, pages 247–260. ACM, 2008.

[32] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM, 2011.

[33] P. Sermanet and Y. LeCun. Traffic sign recognition with multi-scale convolutional networks. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*. IEEE.

[34] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and Tell: A Neural Image Caption Generator. *CoRR*, abs/1411.4555, 2014.

[35] A. Krizhevsky, I. Sutskever, and G.E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[36] Yoav Goldberg. A Primer on Neural Network Models for Natural Language Processing. 2016.

[37] Sato Kaz, Young Cliff, and Patterson David. An in-depth look at Googles first Tensor Processing Unit (TPU). https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, May 2017.

[38] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[39] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

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

[41] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[42] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[43] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[44] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment*, 8(3):209–220, 2014.

[45] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013.

[46] R. Bayer and M. Schkolnick. Readings in Database Systems. chapter Concurrency of Operations on B-trees, pages 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[47] Ryan A. Rossi and Nesreen K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[48] J. Leskovec, K.J. Lang, A. Dasgupta, and M.W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *CoRR*, 2008.

[49] Jure Leskovec and Rok Sosivc. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[50] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. Quickly generating billion-record synthetic databases. In *Acm Sigmod Record*, volume 23, pages 243–252. ACM, 1994.

[51] G. Kurian, J. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. Kimerling, and A. Agarwal. ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network. In *PACT*, 2010.

[52] Brian Kahne. FreescaleADL: An Industrial-Strength Architectural Description Language For Programmable Cores. http://opensource.freescale.com/fsl-oss-projects/, June 2013.

[53] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA*, 2010.

[54] William J Dally and Brian Towles. *Principles and practices of interconnection networks*. 2004.

[55] Sunghyun Park, T. Krishna, C. Chen, B. Daya, A. Chandrakasan, and Li-Shiuan Peh. Approaching the theoretical limits of a mesh NoC with a 16-node chip prototype in 45nm SOI. In *DAC*, 2012.

[56] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 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, pages 469–480, New York, NY, USA, 2009. ACM.

[57] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron. Scaling with Design Constraints: Predicting the Future of Big Chips. *IEEE Micro*, 31(4):16–29, July 2011.

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

[59] J. Leskovec, K.J. Lang, A. Dasgupta, and M.W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 2009.

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