Accelerating Graph and Machine Learning Workloads Using a Shared Memory Multicore Architecture with Auxiliary Support for in-Hardware Explicit Messaging

Halit Dogan*, Farrukh Hijaz*, Masab Ahmad*, Brian Kahne†, Peter Wilson†, Omer Khan*
†NXP Semiconductors, Austin, TX, USA
*University of Connecticut, Storrs, CT, USA

***Abstract*—Shared Memory stands out as a *sine qua non* for parallel programming of many commercial and emerging multicore processors. It optimizes patterns of communication that benefit common programming styles. As parallel programming is now mainstream, those common programming styles are challenged with emerging applications that communicate often and involve large amount of data. Such applications include graph analytics and machine learning, and this paper focuses on these domains. We retain the shared memory model and introduce a set of lightweight in–hardware explicit messaging instructions in the instruction set architecture (ISA). A set of auxiliary communication models are proposed that utilize explicit messages to accelerate synchronization primitives, and efficiently move computation towards data. The results on a 256–core simulated multicore demonstrate that the proposed communication models improve performance and dynamic energy by an average of 4× and 42% respectively over traditional shared memory.**

I. Introduction

Power dissipation and hardware complexity have halted the drive to higher core frequencies and ever-increasing single thread performance. However, transistor density has continued to grow, and machines with many low-power cores have become the means to obtain higher performance. Leveraging parallelism using multicores in mobile and datacenter environments is already underway, and future massive computational models are expected to have multicores with thousands of cores on-chip [1].

This rise in concurrency has greatly facilitated usage of parallelism to solve real world problems, such as synapse deductions from brain graphs. The input data comes from abstract level paradigms, such as road networks [2], social networks [3], and neuroscience [4], which is then processed using machine learning and graph analytic algorithms. Harnessing parallelism in these algorithms remain a challenging task since inter-core communication, memory coherence, and data dependencies all limit performance scalability. Synchronization is a primary bottleneck that steals performance gains at high thread counts [5]. It is a consequence of deploying concurrent threads to perform computations on data, and extract fine-grain parallelism. This leads to mutual exclusion requirements that handle race conditions occurring on shared data structures [6]. In parallel machine learning and graph analytic algorithms, synchronization is unavoidable since there is always shared data between threads/processes that needs to fulfill memory consistency requirements [7]. Hence, the main challenge boils down to efficiently dealing with shared data structures and communication.

Hardware based explicit messages have been explored in the past in the context of shared memory multiprocessors [8], [9] to efficiently handle shared data structures and communication. RAW [10], ADM [11], Active Messages [12], ACS [13], HAQu [14], CAF [15] and commercial Tilera [16] multicore processors have explored the idea of explicit messaging on top of the shared memory paradigm. Even novel accelerator architectures, such as Tesseract [17] have recently shown the benefits of message passing style architecture.

Building on the previous efforts, we implement explicit messaging seamlessly in hardware with lightweight costs, and expose this powerful paradigm to software to obtain performance scaling and energy benefits for graph and machine learning algorithms. The key idea is to retain the shared memory paradigm, and develop auxiliary explicit message based communication models on top of it. These communication models enable efficient management of data locality, overlapping of communication with computation, and movement of computation to data to reduce unwarranted data movement. Since graph workloads often require fine–grain synchronization of critical code sections, they benefit from the proposed communication models. Even though machine learning workloads are highly parallel and would not necessarily require critical code sections to synchronize threads, they benefit from efficient communication models that enable fine–grain parallelization strategies to improve data access reuse within threads. The contributions of this paper include:

1) Implementation of emerging machine learning and graph analytic workloads using the proposed explicit messaging based communication models.
2) Demonstrate the effectiveness of explicit messaging based communication models to extract fine-grain communication in graph and machine learning workloads that improve performance scaling over shared memory by an average of 4×, in some cases by up to 11.9×, and improve dynamic energy by an average of 42% over shared memory.

II. Architecture and Protocol

The baseline system is a tiled multicore architecture with an electrical 2-D mesh interconnection network. Each tile consists of a compute pipeline, private L1 instruction and data

caches, a physically distributed shared last-level cache with an integrated directory for cache coherence, and a network router for inter-core communication. Some tiles are also connected to a memory controller to access off-chip memory.

A. Explicit Messaging ISA Extensions

The proposed architecture utilizes a RISC style ISA with weak memory consistency model. It is extended with four explicit messaging instructions.

Send is a non-blocking instruction which requires a destination address along with the data to be sent.

Recv is a potentially blocking instruction. When a new message is received, it is buffered in a receive queue until handled via a receive instruction. A core's pipeline is stalled if it gets to a *recv* instruction but has not yet received the message.

***Sendr* (send with rendezvous)** is a blocking instruction. It works similar to a *send* instruction with the exception that it always blocks the compute pipeline until an explicit reply is received from the destination thread.

***Resumer* (resume rendezvous)** is a non-blocking instruction which works similar to the *send* instruction. It is used to respond to a *sendr* instruction from a sender.

B. Explicit Messaging Protocol

Explicit messaging between communicating threads is broken up into two protocols; 1) send-recv, and 2) sendr-recv-resumer. Each step in the life of a message is described next.

1) Send–Recv Protocol: When a *send* instruction is executed, a message is constructed by reading source registers setup by preceding software. The source registers contain address of the destination thread, and data to be sent to it. A mapping table is looked up to determine the destination $CoreID$, and the message to be sent is constructed by the core pipeline. If the send queue is full, the pipeline stalls. Otherwise, the message is inserted into the send queue and the compute pipeline commits the instruction.

The decision to inject the message into the network is based on the value of a *"capacity counter"*. There is one *capacity counter* per thread. The value of this counter is set through a special instruction in the sender's thread at the beginning of program execution. If the value of the counter is > 0, the message is injected into the network and the *capacity counter* is decremented. On the other hand, if the value is '0', the message is blocked in the send queue. This counter keeps track of the in-flight messages from the sender's point of view, and enables a per-thread flow-control protocol.

The network routes the message and delivers it to the destination tile's router. If the destination receive queue has available space, the message is inserted into the queue. Otherwise, the message stalls in the network. Note that the on-chip network implements flow-control that can stall a source core from injecting more messages. Since a received message is always consumed at the receiving core, the machine is always guaranteed to make forward progress.

A message at the head of the receive queue wakes up the corresponding destination thread, which upon execution of the *recv* instruction consumes the message by copying the data into appropriate registers already setup by preceding software. The *recv* is a blocking instruction, hence it stalls the pipeline until it can pull the data from the receive queue. If the pipeline has not yet received the *recv* instruction, the message sits in the receive queue waiting to be consumed. As soon as the message is de-allocated from the receive queue, an *ACK* is generated and injected into the network that routes it back to the source core where it increments the *capacity counter*. Note that the applications are not allowed to impose any *ordering* requirement for message arrival, and software must guarantee the processing of all arriving messages at the receive queue. This enables the architecture to implement a first-in-first-out single receive queue per core. The *ACK* message enables an efficient mechanism to track the completion of *send* instructions, as well as manage flow-control for messages between communicating threads.

2) SendR–Recv–ResumeR Protocol: When a *sendr* instruction is executed, the pipeline is blocked until it receives an explicit reply from the destination thread with *resumer*. Therefore, it is paired with a *recv* instruction at the destination, which is consequently followed by a *resumer* instruction to send an explicit reply back to the sender thread. The *resumer* is a non-blocking instruction, which is used to convey certain result back to the *sendr* thread, or just signal it to resume operation. When the sender receives the reply via *resumer* from the destination thread, the *sendr* instruction completes and the pipeline is allowed to proceed.

C. Deadlock Freedom

1) Application Level Deadlock Freedom: An easy way to deadlock the proposed architecture is by improper use of explicit message instructions. For instance, if a thread expects to receive a message from another thread and the second thread never sends the message, it will lead to an application level deadlock.

A deadlock situation can also occur if the order of receiving messages is required for functionality. For example, *thread c* expects to receive a message from *thread a* before a message from *thread b*. However, this cannot be ensured since message arrival from on-chip network cannot be ordered between different sender threads. If *thread b's* message gets to the destination first, it will result in a deadlock. This deadlock scenario can be resolved by making sure that the architecture implements multiple independent receive queues to buffer the messages. This will ensure that the messages are pulled in from the network and inserted into the appropriate receive queue. To generalize this scenario, *the number of independent receive queues must be equal to the number of concurrent communicating threads if the order of receiving messages is necessary for program functionality*. This requirement can lead to significant implementation overhead, hence for design

simplicity the proposed architecture does not impose any ordering requirement for arrival of messages, as discussed in Section II-B1.

2) 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 can be used to avoid overflowing the finite sized receive queues.

3) Network Level Deadlock Freedom: Let us state certain assumptions about the on-chip network that are required for the shared memory baseline. First, the network guarantees point–to–point ordered delivery of messages without deadlocks. Second, the routing algorithm is deadlock–free or can always eventually recover from a deadlock.

A deadlock scenario can arise if the same physical network/virtual channel is utilized for both request messages (*i.e., send*, *sendr*) and reply messages (*i.e., ACK*, *resumer*). Consider a scenario where *thread a* and *thread b* are sending an explicit message to each other. If the *ACK* replies flow on the same channel as the request messages, a network deadlock occurs due to circular dependency between threads. The network deadlock is removed by providing an additional network channel for the request messages. The existing cache coherence reply channel for the reply messages can be reused for explicit messaging replies. This is a valid assumption since it has been shown that cache coherence reply channel can carry replies for both requesting core's requests, as well as directory forwarded requests [18].

D. 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.

E. Send and Receive Queue Overheads

At each tile a 4-entry (1 entry = 4 words) send queue is implemented, which equates to an overhead of $4 \times 4 \times 8B = 128B$ per tile. Each receive queue is sized at 48 entries, resulting in an overhead of $48 \times 4 \times 8B = 1.5KB$ per tile. The size of receive queue is empirically derived using the design space study discussed in Section V-E.

III. Programmable Communication Models

The traditional shared memory synchronization and proposed explicit messaging based communication models are described in this section. Moreover, a representative graph and a machine learning algorithm are described as application illustrations of the various communication models.

First, a few terms are defined. **1) Worker thread** is a thread that performs the main application work. **2) Service thread** is a thread that provides different services (such as management of locks or execution of critical section code) for the *worker threads* to complete application work.

A. Traditional Shared Memory Synchronization

The synchronization primitives are implemented using traditional RISC load–link and store–conditional instructions [19]. This enables ease in directly porting shared memory applications to the proposed architecture. These instructions are implemented with hardware support, and without any involvement of the operating system. The shared memory synchronization based implementations are referred as **Sh_Mem** in subsequent sections.

B. Accelerating Synchronization

Synchronization is accelerated using explicit communication and dedicating one or more *service threads* to manage locks. Explicit messages are used to communicate between worker and the service threads. For example, a *"lock_acquire"* request generates a blocking send message through a *sendr* instruction. If the requested lock is free, the corresponding *service thread* marks it as locked and sends an explicit reply to the requester using the *resumer* instruction. The requesting core completes the *sendr* instruction and enters the critical section code. On the other hand, if a lock is already acquired by another thread, the *service thread* adds the requesting core into a waiter list. The service thread only replies back once the requested lock is free. This behavior ensures atomicity, as there is only one thread managing a given lock variable. The race to acquire a lock is now only in the interconnect and all requests get serialized in the receive queue.

Once a core is done executing the critical section, it releases the lock by executing *"lock_release"*. A non-blocking send message is generated through a *send* instruction. Upon receiving an unlock request, the *service thread* marks the lock as free. Similarly, a *"barrier_wait"* is implemented using blocking *sendr* instruction. In this case, a *service thread* waits for a message from all threads participating in the barrier before replying and unblocking the sender threads.

The idea of *service threads* is generalized to multiple threads. In this case, the locks are statically mapped to the *service threads* and the destination address is obtained using a simple lookup. This improves the performance of lock management, as the communication is spread across cores executing *service threads* to exploit concurrency in locks. The accelerated synchronization model and corresponding benchmark implementations are referred as **Acc** in the subsequent sections.

C. Moving Computation To Data

To avoid synchronization overheads, it is sometimes beneficial to pin and execute critical code sections at a dedicated core. This paradigm trades off synchronization with serialization costs, and is useful for contended synchronization that results in data ping-pong and loss of locality. By serializing computation at the location of data/code, unnecessary data movement costs are eliminated. Under this paradigm, instead of sending a request to acquire a lock, each *worker thread* sends a message to the *service thread*. The *service thread* receives the message and performs the critical section operations *atomically and without any interruption*.

In cases where non–blocking messages from the *worker threads* are used, they send the message and immediately move on, thereby overlapping communication with computation. Another benefit materializes when shared data is pinned at a single core, hence improving data locality and reducing coherence traffic. Multiple *service threads* provide benefits by distributing the critical section work and overlap communication with computation. This approach also provides another flexibility which proves to be highly valuable; *i.e.,* atomic execution of more than one operation. The moving computation to data and corresponding benchmark implementations are referred as **MC** in the subsequent sections.

D. Application Illustrations

The applicability of the proposed communication models is illuminated using workloads from graph analytics and machine learning domains. Graph workloads are well known for their unstructured data access and communication patterns. Moreover, machine learning workloads are also known for their massive amount of data communication requirements. All evaluated workloads are first implemented using the shared memory synchronization primitives, i.e., locks and barriers. Next, all synchronization is ported to a library that implements them using explicit messaging instructions as outlined in Section III-B. Finally, the proposed communication model that moves computation to data (Section III-C) is used to optimize workloads.

1) Graph Analytics: The graph workloads are taken from the *CRONO* benchmark suite [5]. To illustrate applicability of the proposed communication models, a hard to parallelize graph problem is considered that finds the single source shortest paths (SSSP) in a graph.

The SSSP algorithm computes the shortest path in a graph with non–negative edge weights. The algorithm starts from a user defined vertex, and traverses all vertices in the graph, updating neighboring vertices with lowest path costs from the starting vertex. A distance array is updated with the lowest path costs, depicted by the relax function in Algorithm 1.

The algorithm consists of two main loops, an outer loop that visits all the vertices, and the inner loop which visits all the neighboring vertices of a given vertex, each of which can be parallelized. The outer loop is parallelized in a controlled manner, where vertices are scheduled in pareto fronts to update distance costs [20]. Under traditional shared memory, the vertex path costs are updated using locks (Algorithm 1 lines 6–9) as threads may update distances of vertices with common neighbors. A range value specifies which vertices can or cannot be relaxed in a given iteration through the input graph, which is determined based on graph characteristics [20]. Since the updated range value must be propagated to all threads simultaneously, a master thread updates the global range value via barrier synchronization (Algorithm 1 lines 10–12).

Algorithm 1 The relax function in SSSP, using locks

```
1:  Initialize distance_array(D)
2:  (v, u) is a vertex, neighbor pair
3:  << Parallel Relax Function >>
4:  for (each neighbor, u, of v) do
5:      if D[v] + D[v,u] < D[u] then
6:          Lock (u)
7:          if D[v] + D[v,u] < D[u] then
8:              D[u] ← D[v] + D[v,u]
9:          UnLock (u)
10: barrier_wait(barrier)
11: update_range ()
12: barrier_wait(barrier)
```

The shared memory spin locks are expensive because of the instruction retries and additional memory accesses. Therefore, **Acc** is implemented to enhance lock and barrier synchronization. As discussed in Section III-B, the locks in **Acc** are pinned to service threads to prevent cache line ping–pong across the chip. Furthermore, concurrency in the lock management is achieved by utilizing multiple service threads. The SSSP algorithm for **Acc** is the same as Algorithm 1. The only difference is that shared memory locks and barriers are replaced with the primitives described in Section III-B. **Acc** eliminates the instruction and memory overheads of **Sh_Mem**, however it still has locks and related overheads.

MC eliminates locks, and atomically performs the critical section work (i.e., update distances) within a service thread

Algorithm 2 The relax function in SSSP, using messages

```
1:  Initialize distance_array(D)
2:  (v, u) is a vertex, neighbor pair
3:  << Worker Thread : Parallel Relax Function >>
4:  update_range () using sendr
5:  for (each neighbor, u, of v) do
6:      send(v, u, tid)
7:  << Service Thread : Parallel Relax Function >>
8:  recv(v, u, tid)
9:  if D[v] + D[v,u] < D[u] then
10:     D[u] ← D[v] + D[v,u]
```

that is responsible for that vertex. Each worker thread sends messages to the relevant service thread with the target distance array (Algorithm 2 line 6), after which the service thread locally performs computations and commits the distance array value (Algorithm 2 lines 8–10). Since the *send* instruction is non-blocking, the worker thread continues to do other useful work while the service thread performs the critical section work. Note that **MC** pins shared data to service threads, and worker threads avoid accessing it locally to prevent costly sharing misses.

The TRIANGLE–COUNTING and BREADTH–FIRST-SEARCH graph workloads utilize similar **Acc** and **MC** implementations. The PAGERANK, COMMUNITY–DETECTION, and CONNECTED–COMPONENTS graph benchmarks do not contain fine–grain locks. PAGERANK and CONNECTED–COMPONENTS benchmarks have only a single global lock to update a global variable which is implemented using regular shared memory spin lock for **Sh_Mem**. **Acc** implementation eliminates this lock by sending the local calculations to a service thread with a blocking *sendr* instruction. Then the service thread updates the global variable and replies to the worker threads to continue.

2) Machine Learning: Five neural network applications are evaluated as representative machine learning workloads. While four of them are Convolutional Neural Networks (CNNs) of various complexities, one is a more primitive Multilayer Perceptron (MLP). Machine learning workloads are organized in multiple layers, where these layers consist of convolution kernels, as well as fully connected networks. Performance scalability in these workloads depends mainly on the amount of work that separates parallelized layers. If the work per layer is small then barrier communication across layers results in augmented communication costs. However, if the work per layer is large then data access costs dominate performance. In the latter scenario, fine–grain parallelization strategies are implemented to illuminate the benefits of using the proposed communication models.

The basic parallelization strategy for convolutional layers is to divide all the neurons in a layer among the available threads, and synchronize the threads using a barrier after the layer's work is done. The work distribution for convolutional layers is done by tiling the neurons and dividing the tiles among threads, as depicted in Algorithm 3. This parallelization strategy is implemented using the traditional shared memory barrier described in Sec III-A, and also using the proposed accelerated barrier (**Acc**) explained in Sec III-B. Similarly, neurons in fully–connected layers are divided among available threads and synchronized with barriers.

All machine learning benchmarks have the same **Sh_Mem** and **Acc** versions with the parallelization strategy outlined above. However, because large CNNs (CNN-GTRSB and CNN-ALEXNET) involve more parallel computation between barriers, the synchronization overhead is not notable when compared to other CNN and MLP benchmarks. Therefore, to further exploit concurrency in large CNNs, the **MC** approach is adopted and described next.

Algorithm 3 Convolutional Layer of CNN

```
1:  Calculate the number of tiles
2:  nTiles = outChan * (outW/tileW) * (outH/tileH)
3:  Divide the tiles among threads
4:  start = tid * nTiles/nThreads
5:  stop = (tid+1) * nTiles/nThreads
6:  Parallel Convolutional Layer
7:  for each tile in range(start, stop) do
8:      Calculate the range of neuron locations
9:      x_min,x_max,y_min,y_max = get_loc(tile)
10:     for each ch in range(0, kernelW) do
11:         for each y in range(y_min,y_max) do
12:             for each x in range(x_min,x_max) do
13:                 Perform 2d convolution and accumulate
14:                 output[y][x] += convolution(filter,
15:                 input,ch, y, x)
```

In convolutional layers, each neuron output is calculated with the accumulation of one or more 2D convolution operations. When the number of 2D convolutions is high, the work per neuron is large. This leads to work imbalance and limits the reuse of 2D kernels. For instance, if a thread gets to work on an extra tile that contains multiple neurons, it leads to a large work imbalance because each kernel consists of a large number of 2D kernels (e.g. in the third convolutional layer of CNN-ALEXNET benchmark, each kernel has 256 2D kernels). The work imbalance can be prevented by reducing the tile size, but smaller tile means less reuse of each kernel. Hence, it is better to have fine–grain parallelization for this type of CNN to prevent imbalance and have better reuse.

Fine–grain parallelization is achieved by dispatching

Algorithm 4 Convolutional Layer of CNN, using messages

```
1:  << Worker Threads >>
2:  Divide the channels among group of threads
3:  start = tid * nChannels/nThreads
4:  stop = (tid+1) * nChannels/nThreads
5:  for each ch in range(start, stop) do
6:      for each y in range(0,outH) do
7:          for each x in range(0,outW) do
8:              Perform convolution for one channel
9:              psum = convolution(filter, input, ch, y, x)
10:             Send psum to accumulation core
11:             sendmsg(AccumCore, psum,y,x)
12: << Service Thread: Accumulation Function>>
13: num_msg = 0;
14: while num_msg < nChannels do
15:     recvmsg(&addr, &psum, &y, &x)
16:     output[y][x] += psum
17: end while
```

Architectural Parameter	Value
Number of Cores	256 @ 1 GHz
Compute Pipeline per Core	In–Order, Single–Issue
Memory Subsystem	
L1–I Cache per core	16 KB, 4–way Assoc., 1 cycle
L1–D Cache per core	32 KB, 4–way Assoc., 1 cycle
L2 Inclusive Cache per core	256 KB, 8–way Assoc. 2 cycle tag, 4 cycle data
Directory Protocol	Invalidation–based MESI ACKwise$_4$ [21]
Num. of Memory Controllers	8
DRAM Bandwidth/Latency	10 GBps per Controller/ 100ns
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	
Send queue per core	4 entries
Receive queue per core	48 entries

Table I: Architectural parameters for evaluation.

multiple threads to work on one neuron. This must be implemented using critical code sections because multiple threads have to accumulate on one neuron. The critical section work can be realized either utilizing shared memory spin locks or deploying the **MC** model. We implemented both versions, however because the overhead of spin locks is too large, it did not even scale as much as previously outlined shared memory implementation. Therefore, going forward we do not show the results for the shared memory version with locks. **MC** version is implemented as depicted in Algorithm 4. The partial sum of each neuron for each kernel channel is calculated (line 9), and sent to the service thread to be accumulated (line 11). The service thread (lines 14–17) receives the partial sums from the worker threads, and accumulates it on the output to be used in the following layers. Since the **MC** implementation executes application work with high locality, and enables aggressive overlap of communication with computation, it provides performance benefits over **Sh_Mem** and **Acc**.

IV. Evaluation Methodology

A. Compiler Support

An LLVM–based compiler was developed that supports the proposed ISA extensions. The compiler itself does not inherently understand explicit messaging. Instead, it simply wraps the explicit messaging instructions within assembly blocks, using the gcc extended asm block syntax to instruct the compiler as to what registers are inputs or outputs. This allows the compiler to allocate registers and schedule code.

B. Simulator Setup

The proposed architecture is implemented using an in–house industry–class simulator and the associated LLVM–based compiler. A futuristic 256–core tiled multicore processor with a two–level private L1, shared L2 cache hierarchy per core is evaluated. Each single-issue core is mapped spatially to either a *service* or a *worker thread* based on the corresponding communication model (cf. Section III). The right ratio of *service* and *worker threads* is empirically derived and discussed in Section V. The default architectural parameters used for evaluation are shown in Table I.

Benchmark	Input Dataset
Graph Analytics (CRONO [5])	
PAGERANK, TRIANGLE COUNTING COMMUNITY DETECTION, BFS CONNECTED–COMP, SSSP	California Road Network [2]
Machine Learning	
MLP-MNIST, CNN-MNIST [28]	MNIST [29]
CNN-GTSRB [30]	GTSRB [31]
CNN-ALEXNET[32]	ImageNet [33]
CNN-SQUEEZENET[34]	ImageNet [33]

Table II: Problem sizes for our parallel benchmarks.

C. Performance and Energy Models

All experiments are performed using the core, cache hierarchy, coherence protocol, memory system, and on–chip interconnection network models implemented within the multicore simulator. The electrical mesh interconnection network uses XY routing. A 2–cycle per hop delay is modeled; appropriate pipeline latencies associated with loading and unloading a packet onto the network are also accounted [22], [23]. In addition to the fixed per–hop latency, network contention delays are also modeled. These models are derived from the Graphite multicore simulator [24]. The performance models are extended to accurately account for explicit messaging instructions. All mechanisms and protocol overheads discussed in Section II are modeled.

The energy numbers for core energy and memory components are obtained from McPat [25] using the $22nm$ technology, and scaled to $11nm$ by using the scaling numbers from [26]. The network-on-chip per event energy numbers are acquired from DSENT [27]. For the core energy model, in addition to datapath components such as register file, ALU and store queue, the send and receive queues are also taken into account and modeled.

D. Benchmarks and Evaluation Metrics

The benchmarks and their inputs are presented in Table II. Six graph benchmarks are taken from the *CRONO* benchmark suite [5]. Five machine learning benchmarks are developed using the models and datasets referenced in Table II.

Each benchmark is run to completion, and the completion time and energy consumption of *parallel* region is measured. The parallel completion time is broken down into the following categories:

1) Compute Stalls is the time spent retiring instructions,

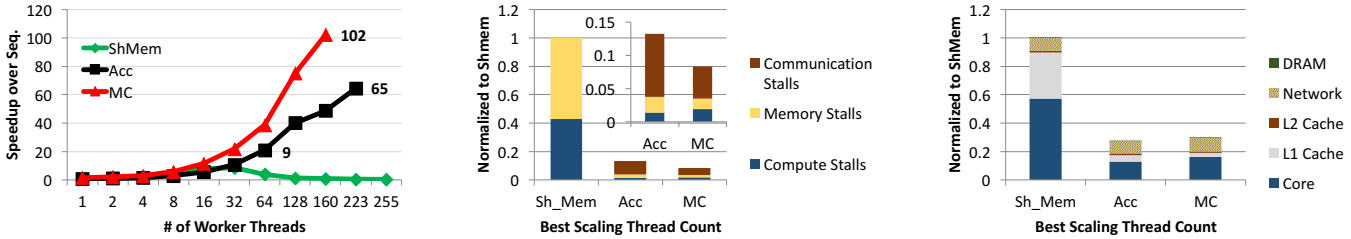

Figure 1: Completion time and energy results for SSSP under **Sh_Mem**, **Acc**, and **MC**. Left graph is normalized to sequential SSSP, while the middle and right graphs are normalized to shared memory implementation at the best thread scaling configuration.

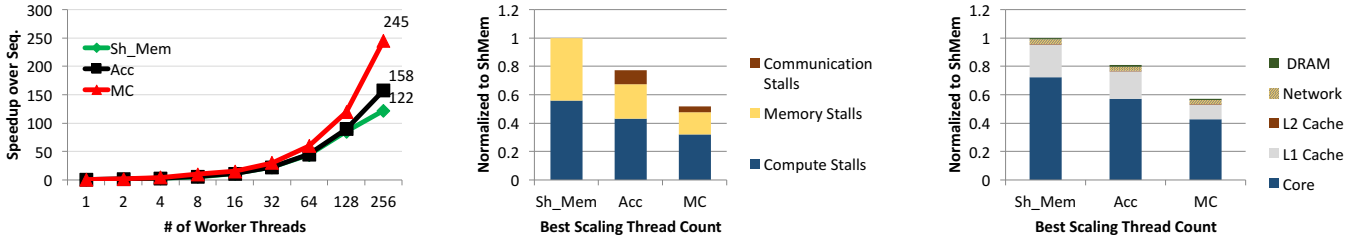

Figure 2: Completion time and energy results for CNN-ALEXNET under **Sh_Mem**, **Acc**, and **MC**. Left graph is normalized to sequential CNN-ALEXNET, while the middle and right graphs are normalized to shared memory implementation at the best thread scaling configuration.

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. Per event energy numbers are obtained from the models described in Section IV-C.

V. Results

This section first discusses representative benchmarks from machine learning (CNN-ALEXNET) and graph analytics (SSSP). Next, the performance scaling and energy consumption for remaining workloads is discussed.

A. *Graph Analytic* SSSP *Workload*

Figure 1 (left) shows the performance scaling results as a function of thread count. Shared memory implementation (**Sh_Mem**) scales to 32 threads and shows a maximum speedup of $9\times$ over sequential implementation. The speedup decreases after 32 threads because of excessive instruction retries to acquire contended locks. The instruction retries show up as large compute and memory stalls in Figure 1 (middle). The **Acc** implementation improves performance scalability up to 223 *worker threads* (using the remaining 33 cores as *service threads*) to yield a maximum speedup of $65\times$ over sequential, and $7.2\times$ over **Sh_Mem**. By pinning locks at dedicated cores and accessing them via explicit messages, the cache line ping-ponging and instruction retries are eliminated. Furthermore, multiple lock managers overlap communication and computation providing performance gains. However, **Acc** introduces notable communication stalls because of lock acquisition overheads.

MC affixes shared data at a set of cores, as described in Section III-C. As computation is moved to where data lives, gains in data locality are realized. Furthermore, **MC** improves performance over **Acc** by eliminating the locks and related overheads. As the *worker threads* offload critical section updates using non-blocking *send* messages, their chance of overlapping communication with computation greatly increases. The compute pipeline may still stall due to other reasons described in Section II. Since a send capacity counter of 4 is used per thread (cf. Section V-D), communication stalls do occur and are visible in Figure 1 (middle). Number of *service threads* are empirically set to 96 threads (remaining are *worker threads*) because it provides best performance scaling. As a result, SSSP is able to take advantage of **MC** and scale up to 256 cores to achieve a speedup of $102\times$ over sequential. This translates to a speedup of $11.9\times$ over shared memory and $1.56\times$ over **Acc**.

Figure 1 (right) also displays the dynamic energy comparison of SSSP implementations. **Acc** enhances the overall dynamic energy by eliminating instruction retries caused by spin locks. Even though **MC** improves energy compared to **Sh_Mem**, it has slightly worse energy compared to **Acc**. As seen from the figure, most of this additional energy comes from the core energy. The reason for this is that **MC** has more instructions executed because it has only test–and–set in its relaxation function, while **Acc** utilizes test–test–and-set. This can be seen from Algorithms 1 and 2. As a result, it has an increase in its core energy.

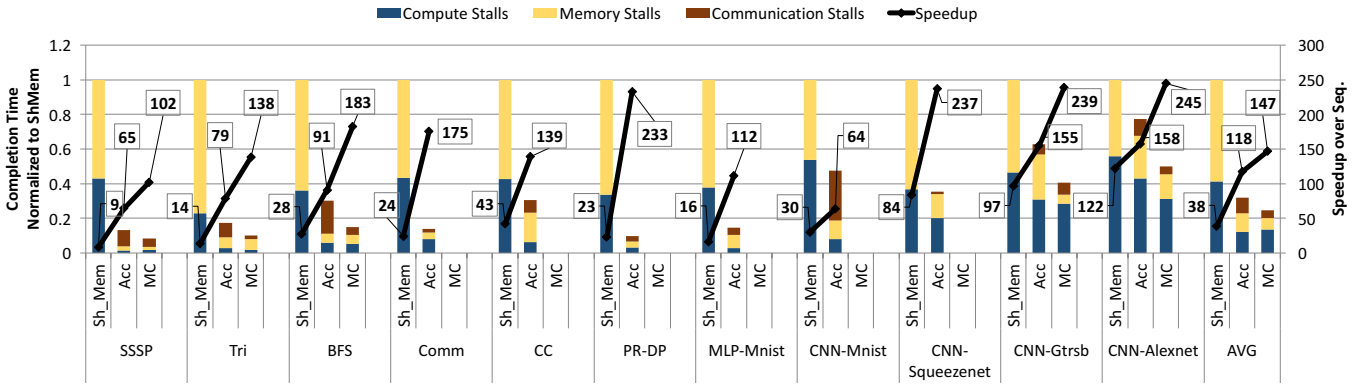

Figure 3: Completion time results for **Sh_Mem**, **Acc**, and **MC** at their respective best thread scaling point; all normalized to **Sh_Mem**.

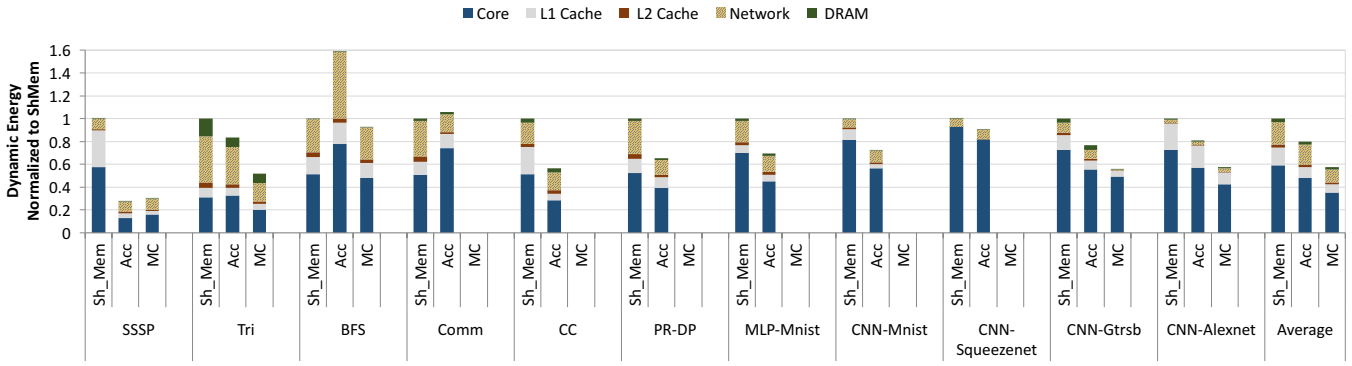

Figure 4: Dynamic energy results for **Sh_Mem**, **Acc**, and **MC** at their respective best thread scaling point; all normalized to **Sh_Mem**.

B. Machine Learning ALEXNET *Workload*

Figure 2 (left) shows the performance scaling results for all three communication models, and all of them scale up to 256 cores. However, **Sh_Mem** has a speedup of 122× over sequential at 256 cores, and **Acc** improves it by 1.3× to 158× over **Sh_Mem**. This benefit comes from reducing the instruction count and memory stalls by using the **Acc** barrier implementation. Improvement in barrier is limited because CNN-ALEXNET has a lot more computation per core in each layer. Therefore, communication stalls do not come up as notable components of the completion time, as illustrated in Figure 2 (middle).

To improve performance further, **MC** is implemented as described in Sec III-D2. The number of *service threads* are decided empirically as 16. The reason to have a small number of *service threads* is because CNN-ALEXNET is a compute and data intensive workload which needs more *worker threads*. **MC** enhances performance by 2× over the best performing **Sh_Mem** version. As **MC** enables fine–grain parallelization by allowing multiple threads to work on a single neuron, it improves the reuse of the kernel channel in L1 data cache, which helps decrease memory stalls. Additionally, it helps to reduce the number of loads by accumulating multiple channels worth of partial sums in a local variable. This way the compiler enables accumulating partial sums in core registers without loading data for every accumulation. However, in the **Sh_Mem** case, because a single thread accumulates over the global data structure instead of local variable, the compiler generates load instructions for every accumulation, which results in higher instruction count.

The dynamic energy breakdowns for all versions of CNN-ALEXNET can be observed from Figure 2 (right). It is clear that because of the instruction count and memory stall improvements of the **Acc** over **Sh_Mem**, core energy and L1 data cache energy decreases. **MC** further decreases L1 data cache energy because it eliminates extra load instructions.

C. Performance and Energy Scaling Analysis

Figures 3 and 4 show the completion time and dynamic energy breakdowns for the best scaling thread count of each benchmark. The results of **Sh_Mem**, **Acc**, and **MC** are normalized to the best scaling point of **Sh_Mem**.

1) Workloads with Fine-grain Synchronization: The benchmarks with fine-grain synchronization include BFS, TRIANGLE COUNTING, and CNN-GTRSB. The CNN-GTRSB benchmark is similar to CNN-ALEXNET, so it is not discussed in further detail. The two graph benchmarks show similar behavior as SSSP and their performance results follow SSSP discussion in Section V-A.

The dynamic energy breakdowns for the benchmarks are presented in Figure 4. **Acc** boosts dynamic energy for BFS as

compared to **Sh_Mem**. The reason for this boost is because of the increase in core and network energy. The core energy augments for BFS because its benefits from **Acc** are not as much as the other benchmarks. The reason for this is that the instruction count for **Acc** version of BFS declines by 5.8× compared to **Sh_Mem**, however it has 8× more cores. Due to this difference, an increase in the core energy for BFS is observed for **Acc**. Similarly, network energy also rises due to larger core count and less benefits from **Acc**.

2) Workloads with Coarse-grain Synchronization: The benchmarks in this category include PAGERANK, COMMUNITY DETECTION (aka COMM), CONNECTED–COMPONENTS (aka CC), MLP-MNIST, CNN-SQUEEZENET, and CNN-MNIST. These benchmarks do not have **MC** versions because they do not require fine–grain synchronization.

As seen in Figure 3, the **Acc** versions of COMM, PAGERANK and CC deliver 7.2×, 10.1× and 3.2× better performance respectively, as compared to **Sh_Mem**. Replacing shared memory barriers with accelerated barrier implementation using explicit messages boosts performance for all three benchmarks. Similar to other benchmarks, MLP-MNIST and CNN-MNIST also suffer from overheads of **Sh_Mem**. Since both benchmarks do not have much work per thread between barriers, the shared memory barrier overheads become dominant and they do not scale beyond 64 cores. **Acc** accelerates shared memory barriers and delivers 6.9× speedup for MLP-MNIST, and 2.1× speedup for CNN-MNIST over **Sh_Mem**. Compared to CNN-MNIST and MLP-MNIST, CNN-SQUEEZENET has more work and larger networks per layer. Therefore, **Sh_Mem** version scales to 128 threads and provides 84× speedup over sequential. **Acc** reduces communication stalls by improving the barrier implementation, which allows it to scale to 256 cores and deliver 237× speedup over sequential.

As shown in Figure 4, while **Acc** provides dynamic energy benefits for CC, PAGERANK, CNN-MNIST, MLP-MNIST and CNN-SQUEEZENET, dynamic energy for COMM worsens. MLP-MNIST and CNN-MNIST are small benchmarks, and as discussed previously a big portion of the completion time comes from the barrier synchronization cost for **Sh_Mem**. Hence, eliminating the aforementioned cost yields energy improvement even though **Acc** has more active cores. Even though CNN-SQUEEZENET has a lot more work than the other two neural networks, it also has many more barriers. These barriers result in increased instruction count and memory accesses for **Sh_Mem**, which then leads to a boost in dynamic energy. Therefore, reducing overheads of barrier implementation with **Acc** improves the energy for CNN-SQUEEZENET as well. PAGERANK and CC also show similar behavior despite having more work compared to the other benchmarks. COMM is the only benchmark in this section that has worse energy for **Acc**. COMM worsens energy consumption because of the same reason as BFS. While **Sh_Mem** scales to 32 threads, **Acc** scales to 256 threads, thus **Acc** has 8× more executing threads. However, the

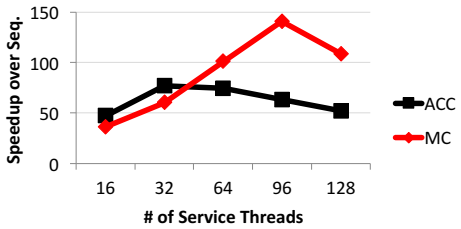

Figure 5: Average speedup over sequential by varying service thread count from 16 to 128 for *Acc* and *MC* models. The per thread send capacity counter is set to 4 for this study.

average instruction count improvement per thread compared to **Sh_Mem** is only 5.2×. Therefore, **Acc** observes an increase in dynamic energy for COMM.

3) Summary of Results: The proposed explicit messaging based communication models improve both performance and dynamic energy consumption over the traditional shared memory paradigm. The **Acc** model achieves average speedup of 3.1× over **Sh_Mem** by pinning synchronization related shared data on cores executing *service threads*, and utilize explicit messages to ship updates to them. The **MC** implementation further enhances performance by 4× over **Sh_Mem** by completely eliminating locks, and concurrently executing the critical section work on *service threads*. In addition to performance benefits, **Acc** accomplishes average of 20% dynamic energy reduction over **Sh_Mem**. However, **MC** further improves locality and achieves an average of 42% less dynamic energy compared to **Sh_Mem**.

D. Determining the Number of Worker and Service Threads

For **Acc** and **MC** communication models, a design space study is performed to empirically determine the right ratio of worker and service threads that lead to best performance. SSSP, TRIANGLE COUNTING and BFS benchmarks are used to illuminate the findings of this study. Even though CNN-ALEXNET and CNN-GTRSB utilize service threads for the **MC** model, they do not have fine-grain synchronization primitives in their **Acc** version. Therefore, they are not considered in this study.

The number of service threads are varied from 16 – 128 while keeping the send capacity counter fixed at 4. The speedup for each service threads setting is measured, and average results for both **Acc** and **MC** models are presented in Figure 5. **Acc** performance degrades when the number of service threads is increased beyond 32, while **MC** performs better with higher service thread count and speedup decreases beyond 96 service threads. Therefore, 32 service threads for **Acc** and 96 service threads for **MC** are used.

As lock acquisition is a blocking operation, the number of in–flight lock requests can be at most equal to the number of worker threads. Therefore, a small number of service threads are required in **Acc**. On the other hand, critical section work requests are not blocking, therefore worker threads have multiple in-flight critical section requests. Moreover, the worker threads in **MC** do not perform critical section work,

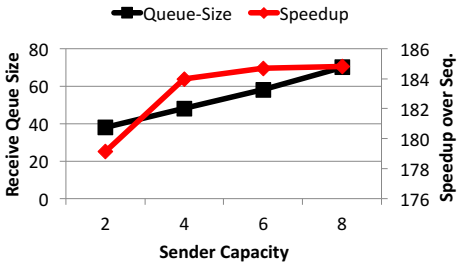

Figure 6: Receive queue size (left y-axis) and average speedup (right y-axis) observed by varying send capacity counter from 2 to 8 using the *MC* model.

which reduces the amount of work done in each worker thread. Consequently, the **MC** model requires more service threads to handle all critical section requests.

E. Determining the Receive Queue Capacity

A design space study is performed to empirically determine the per core receive queue size using the **MC** model. The service thread count is set to 96 for SSSP, TRIANGLE COUNTING and BFS, and 16 for CNN-GTRSB and CNN-ALEXNET benchmarks. Figure 6 shows the average results across all benchmarks. The per thread sender capacity counter is swept from 2 – 8. For each sender capacity counter setting, the maximum utilization among the receive queues, as well as the corresponding performance speedup is plotted. The left y-axis shows the acquired receive queue size to obtain the speedup shown in the right y-axis for the corresponding sender capacity value. Beyond a sender capacity of 4, 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, the per thread sender capacity is empirically determined and set to 4, while the per core receive queue is set to 48 messages.

VI. Related Work

The idea of hardware-level explicit messages for efficient communication has been explored in the context of multi-processors [8], [9]. The proposed architecture differs from these works by exploring the idea of explicit communication in the context of single–chip multicores, where the tradeoffs are different. Moreover, the challenge applications are the emerging graph analytic and machine learning workloads that have been shown here to benefit greatly from the proposed communication models.

The architecture closest to the proposed one is developed and commercialized by Tilera [16]. Tilera developed a multicore cache coherent processor with message passing on a separate on-chip network. The proposed architecture is different than Tilera chips in some implementation details. However, the main factors that differentiate this paper are: (1) the proposed programmable communication models are novel, i.e., accelerating communication without changing a shared memory application, and moving computation to data for fine-grain locality optimal critical code section execution without traditional synchronization primitives. (2) the graph and machine learning applications and their significant performance scaling and energy improvements using the proposed communication models are novel insights. An evaluation of the developed workloads and the communication models on Tilera processors is future work.

ADM [11] explores point–to–point asynchronous messaging for task scheduling. ADM implements the messaging on top of shared memory, however it does not utilize it for general purpose computation. Active Messages (AM) [12] recently showed the applicability of hardware message passing on top of a shared memory architecture. It demonstrates the benefits achievable by such an architecture, however it stops short of fully exploring the architecture for real workloads. Furthermore, it requires a separate hardware context per core for processing explicit messages.

HAQu [14] and CAF [15] show that the multicore processors benefit from hardware queues to enhance fine–grain synchronization. This approach is similar to our proposed **MC** communication model. However, the target applications for their work, and machine learning and graph workloads in our work are the differentiating factor. Performance scaling on large-scale multicores (100s of cores) in our work also show futuristic trends for hybrid shared memory and explicit messaging architectures.

The idea of using separate cores to handle critical section code is explored in ACS [13]. Although ACS is similar to the proposed work in a way that it ships critical section code to some specified core, the proposed architecture differs in several aspects. ACS targeted small-scale heterogeneous multicores, whereas the proposed work targets large-scale multicores where the on-chip network latency and contention plays a significant role. In this context a spatial thread distribution scheme is presented that mitigates serialization effects and improves concurrency. ACS does not remove locks, however the proposed **MC** model eliminates locks and associated unwanted data movement overheads. Finally, the proposed architecture demonstrates the advantage of novel communication models for graph and machine learning workloads that have recently gained significant attention.

Tesseract [17] shows the benefits of only message passing style architecture in processing–in–memory context. It utilizes a 3D stacked memory and a large number of simple cores located near memory. However, it lacks shared memory support. In contrast, the proposed architecture leverages similar message passing style communication, while retaining shared memory.

VII. Conclusion

This paper proposes novel communication models for accelerating the emerging machine learning and graph analytic workloads on futuristic large single-chip multicore processors with support for hardware-level explicit messaging on top of the shared memory paradigm. These workloads exhibit fine-grain communication and data sharing for large

amounts of data. The programmable communication models introduced by the proposed architecture enable such workloads to improve performance by an average of $4\times$ over the traditional shared memory paradigm. Furthermore, the proposed architecture achieves an average of 42% less dynamic energy consumption compared to shared memory.

ACKNOWLEDGMENT

This work was supported in part by Semiconductor Research Corporation (SRC). This research was also partially supported by the National Science Foundation under Grant No. CCF-1452327.

REFERENCES

[1] S. Borkar, "Thousand core chips: a technology perspective," in *ACM Design Automation Conference*, 2007.

[2] 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*, pp. 29–123, 2009.

[3] J. Mcauley and J. Leskovec, "Discovering social circles in ego networks," *ACM Transactions on Knowledge Discovery from Data*, p. 4, 2014.

[4] J. W. Lichtman, H. Pfister, and N. Shavit, "The big data challenges of connectomics," *Nature neuroscience*, pp. 1448–1454, 2014.

[5] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *IEEE Int. Symp. on Workload Characterization*, 2015, pp. 44–55.

[6] D. Nguyen, A. Lenharth, and K. Pingali, "Deterministic galois: On-demand, portable and parameterless," *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 499–512, 2014.

[7] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[8] J. Kubiatowicz and A. Agarwal, "Anatomy of a message in the alewife multiprocessor," in *Intl. Conf. on Supercomputing*, 1993, pp. 195–206.

[9] T. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Intl. Symp. on Computer Architecture*. IEEE, 1992.

[10] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua *et al.*, "Baring it all to software: Raw machines," *Computer*, pp. 86–93, 1997.

[11] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 2010.

[12] R. C. Harting and W. J. Dally, "On-chip active messages for speed, scalability, and efficiency," *IEEE Transactions on Parallel and Distributed Systems*, pp. 507–515, 2015.

[13] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[14] D. Tiwari, J. Tuck, S. Y, and S. Lee, "Haqu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor," *Int. Symp. on High Performance Computer Architecture*, 2011.

[15] Y. Wang, R. Wang, A. Herdrich, J. Tsai, and Y. Solihin, "Caf: Core to core communication acceleration framework," in *Intl. Conf. on Parallel Architectures and Compilation*, 2016.

[16] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE micro*, 2007.

[17] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Intl. Symp. on Computer Architecture*. IEEE, 2015, pp. 105–117.

[18] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang *et al.*, "Openpiton: An open source manycore research framework," *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2016.

[19] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams, "Synchronising c/c++ and power," *Conf. on Programming Language Design and Implementation*, pp. 311–322, 2012.

[20] J. Y. Yen, "An algorithm for finding shortest routes from all source nodes to a given destination in general networks," *Quarterly of Applied Mathematics*, pp. 526–530, 1970.

[21] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal, "Atac: a 1000-core cache-coherent processor with on-chip optical network," in *Intl. Conf. on Parallel architectures and compilation techniques*. ACM, 2010, pp. 477–488.

[22] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[23] S. Park, T. Krishna, C.-H. Chen, B. Daya, A. Chandrakasan, and L.-S. Peh, "Approaching the theoretical limits of a mesh noc with a 16-node chip prototype in 45nm soi," in *Design Automation Conference*, 2012.

[24] 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 *Intl. Symp. on High Perf. Computer Architecture*, 2010.

[25] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Intl. Symp. Symposium on Microarchitecture*, 2009.

[26] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron, "Scaling with design constraints: Predicting the future of big chips," *IEEE Micro*, pp. 16–29, 2011.

[27] 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*. IEEE, 2012, pp. 201–210.

[28] M. A. Nielsen, *Neural Networks and Deep Learning*. Det. Press, 2015.

[29] Y. Lecun, C. Cortes, and C. Burges, "Mnist handwritten digit database," 1992.

[30] P. Sermanet and Y. LeCun, "Traffic sign recognition with multi-scale convolutional networks," in *Intl. Conf. on Neural Networks*, 2011.

[31] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "The german traffic sign recognition benchmark: a multi-class classification competition," in *Intl. Conf. on Neural Networks*, 2011.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[33] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Conf. on Computer Vision and Pattern Recognition*, 2009.

[34] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv:1602.07360 preprint*, 2016.