General Interest

In-Hardware Moving Compute to Data Model to Accelerate Thread Synchronization on Large Multicores

Masab Ahmad
University of Connecticut

Halit Dogan
University of Connecticut

José A. Joao
Arm Research

Omer Khan
University of Connecticut

***Abstract*—In this article, the moving computation to data model (MC2D) is proposed to accelerate thread synchronization by pinning shared data to dedicated cores, and utilize in-hardware core-to-core messaging to communicate critical code execution. The MC2D model optimizes shared data locality by eliminating unnecessary data movement, and alleviates contended synchronization using nonblocking communication between threads. This article evaluates task-parallel algorithms under their synchronization-centric classification to demonstrate that the effectiveness of the MC2D model to exploit performance correlates with the number and frequency of synchronizations. The evaluation on Tilera TILE-Gx72 multicore shows that the MC2D model delivers highest performance scaling gains for ordered and unordered algorithms that expose significant synchronizations due to task and data level dependencies. The MC2D model is also shown to deliver at par performance with the traditional atomic operations based model for highly data parallel algorithms from the unordered category.**

Digital Object Identifier 10.1109/MM.2019.2955079
Date of publication 22 November 2019; date of current version 14 January 2020.

In this article, the moving compute to data (MC2D) model is proposed to accelerate thread synchronizations in large cache coherent multicores.[1–3] The MC2D model pins shared data to dedicated core(s), and utilizes in-hardware core-to-core messaging to invoke critical code sections at those cores. Consequently, data locality is optimized by preventing unnecessary shared data movement between cores. The objective of this article is to study fundamental questions regarding practical adoption of the MC2D synchronization model. What characteristics of a parallelized algorithm are best suited for the MC2D model? Does the MC2D model port efficiently to highly data parallel algorithms? These aspects are evaluated for the MC2D model on a real multicore machine *Tilera's Tile-Gx72* that enables both hardware cache coherence and in-hardware core-to-core messaging. Moreover, the MC2D model is compared against spin-lock and atomic instructions based synchronization models for a representative set of task-parallel algorithms.

This article creates a synchronization-centric characterization of task-parallel algorithms to guide the hypothesis that the MC2D model works best when an algorithm exhibits high synchronizations.

Task parallelism is a popular strategy for multicore processors to exploit fine-grain parallelism. This article creates a synchronization-centric characterization of task-parallel algorithms to guide the hypothesis that the MC2D model works best when an algorithm exhibits high synchronizations. Work efficiency is a fundamental metric used to evaluate the efficacy of an algorithm. However, exploiting task-parallelism while maximizing work efficiency is a hard problem, since it requires *ordered* task execution. Frameworks, such as kinetic dependence graphs (KDG)[4] enforce ordered task-parallel execution by introducing significant synchronizations to globally order tasks among threads. The Galois[5] framework creates *relax-ordered* task-parallel operators that introduce locally ordered task processing per core. Although this approach reduces the need for global task synchronizations, it introduces races in task-level data dependencies that add redundant computations in the algorithms for convergence. Both *ordered* and *relax-ordered* algorithms expose significant synchronizations that open opportunities for the MC2D model to improve performance scalability.

Many task parallel algorithms also exist that do not enforce task ordering and work-inefficiencies in their task-parallel implementations. This *unordered* task execution category, however, may still implement synchronizations due to data dependencies between tasks. Depending on the number and frequency of these synchronizations, the MC2D model seamlessly adapts and delivers competitive performance. In highly parallel form, the only synchronizations present in some algorithms are enforced when all threads observe barrier synchronization to transition from one phase to the next. Even for completely data-parallel algorithms, the MC2D model is shown to be competitive with traditional spin-lock and atomic operations based synchronization model. Various task parallel algorithms from the domains of graph processing, machine learning, database, and data analysis are characterized as *ordered*, *relax-ordered*, *unordered* with task-level data dependencies, and *unordered* with thread-level ordering. The evaluation on *Tilera's Tile-Gx72* multicore shows that the MC2D model performs best under high synchronizations while it matches the performance of state-of-the-art atomic model for highly data-parallel algorithms.

THREAD SYNCHRONIZATION MODELS

Thread synchronization under traditional shared memory is done using an atomic memory operation in hardware by locking a cache line in the private cache of the requesting core (*near atomic*), or as remote read–modify–write operation at the home last-level cache location for the cache line (*far atomic*). As the core count increases, the atomic instructions suffer from the cost of expensive data sharing as cache lines ping pong between the communicating threads. When applicable, the atomic instructions can be directly utilized to implement synchronization. However, they are limited to specific operations and data sizes, thus limiting their applicability to a wide range of critical section implementations. In this case, the atomic operation is utilized to build the

widely applicable spin-lock-based synchronization model that protects an arbitrary critical code section. In addition to cache line ping-pong, spin-lock implementation also suffers from instruction retries under contended thread synchronizations.

The MC2D model removes atomic operations from each critical code section. Consequently, the critical code sections are serialized on a dedicated *service core*. The shared data structures associated with the critical sections are pinned and updated by the dedicated service core. The actual application threads (executing on *worker cores*) send in-hardware critical section execution requests to the service core. The service core receives the request messages one at a time, and execute the critical section sequentially to maintain atomicity of operations. Depending on the algorithmic requirements, the worker thread may wait for a reply message from the service thread (blocking communication), or just continue to perform other work as soon as it sends the message (nonblocking communication). The serialization of critical code sections at a single service core suppresses the exploitable parallelism in such computations. Therefore, multiple cores are assigned as service cores by dividing nonoverlapping shared data among them. In this case, the worker threads send their critical section requests to the corresponding service thread based on the mapping logic of shared data. The MC2D model exploits shared data locality, and eliminates both retries and ping-pong of shared data by pinning it at a dedicated service core. However, the computations in the worker and service cores must be load balanced for near-optimal performance.

The MC2D model is shown to improve performance scalability over the traditional atomic and spin-lock synchronization models, as core counts increase.[2,3] However, several questions regarding the practical adoption of the MC2D model remain unanswered. What characteristics of a parallelized algorithm are best suited for the MC2D model? Does the MC2D model port efficiently to a broad category of parallel algorithms? Based on the widely popular task-parallel execution model, this article presents a detailed algorithm-centric instrumentation and characterization of the MC2D model.

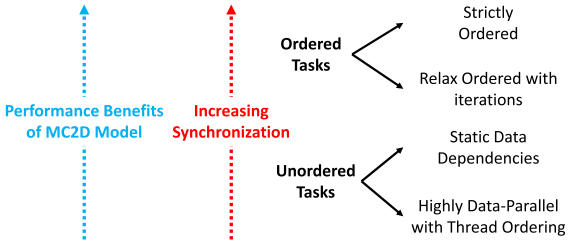

Figure 1. Workload ordering and dependence categorization for synchronization inference.

ALGORITHM-CENTRIC CLASSIFICATION OF THREAD SYNCHRONIZATIONS

Task parallelism simplifies parallel programming[I] and has gained popularity due to its integration in modern concurrency frameworks.[4,5] The programmer specifies tasks to expose parallelism, where each task is a unit of computation that executes in parallel with other tasks. However, system aspects, such as load balancing and thread synchronizations are managed by library (or framework) constructs. This machine independent execution model allows task parallel algorithms to scale at large core counts. All task parallel algorithms operate under some sort of a task ordering execution model. A task's execution order may depend on another task, or a group of tasks may require all threads to synchronize for their interdependencies to propagate. Moreover, tasks may modify shared data with read and/or write dependencies. All these inter and intra task dependencies lead to thread synchronizations. In the most parallel form, all tasks execute with no inter or intra dependencies. To contextualize the relevance of increasing synchronization for task-parallel algorithms, Figure 1 presents an algorithm-centric classification.

Ordered Tasks category strictly enforces an execution order of tasks among the cores. This strategy operates with the work efficiency of the sequential algorithm counterpart. However, extracting parallelism among tasks while enforcing a global task execution order is a hard problem. KDG[4] is a task-parallel execution framework that supports strictly ordered tasks. It implements a parallel queue (e.g., a priority queue) per core, and enforces globally ordered push and

[I] https://software.intel.com/tbb.

extract of tasks using an order list that is visible to all cores. Before dequeuing a task, a *safe source test* checks whether that task has another dependent task in another core. If so, the dequeue operation waits until the inter task dependencies resolve. Otherwise, independent tasks are allowed to concurrently execute in their respective cores. Note that during a task's execution, its shared data updates may also require thread synchronizations. The KDG framework utilizes the above strategy to enable task-parallelism while ensuring work-efficient algorithmic execution. An alternative strategy adopted for ordered algorithms is to tradeoff work-efficiency for parallelism. For example, the Galois framework[5] implements per-core priority queues while ignoring the global intertask execution order. This *relax-ordered* category mitigates global thread synchronizations by not performing the *safe source test*. However, the shared data values are monotonically (and synchronously) updated to ensure task data dependencies are enforced. This leads to multiple iterations for the algorithms to converge, which increases the redundant work performed by the cores. Adopting the ordered and relax-ordered algorithms maximizes the need for synchronizations among threads.

Unordered Tasks category encompasses task-parallel algorithms that enforce no local or global ordering among tasks. Consequently, their work efficient implementations result in good performance scalability on parallel machines. However, these algorithms may still implement synchronizations due to shared data dependencies among tasks. On the other hand, when no such shared data dependencies exist, most (if not all) algorithms require multiple phases of task-parallel computations. These phases execute independent tasks in parallel, but their data outputs must synchronously propagate from one layer to another. This thread-level ordering is enforced using primitives, such as barrier synchronization. Thread synchronizations in unordered algorithms increase as the data dependencies between tasks, or the frequency of barrier synchronization increases.

Figure 1 shows the classification of ordered through unordered task-parallel algorithms, and the impact of thread synchronizations on their performance scalability. The MC2D model is hypothesized to work best as algorithms experience increasing thread synchronizations. The following section describes how various categories of task-parallel algorithms can be ported to the MC2D model.

TASK PARALLELISM UNDER THE MC2D MODEL

The programming model of a task-parallel shared-memory application is not changed for the MC2D model. The only difference is that critical section requests are moved to a separate routine that is processed by the service threads. The critical section code in each (worker) thread is replaced with a request message to invoke execution by the corresponding service thread. The barrier synchronization is handled in a similar manner, but instead of a dedicated service thread, one of the worker threads handles the barrier. This process is automated by identifying all synchronization points in the code. Similar to RCL,[6] refactoring tools can be easily utilized to automatically transform existing applications. However, this work performs manual transformations.

Figure 2 presents an abstract framework construct (similar to KDG) outlining data structures and codes that exhibit thread synchronizations for ordered task-parallel algorithms. The pseudocode on the left represents a generic code that a thread executes under the atomic synchronization model. Task orderings are maintained using a per-core *taskQueue* and a global *orderList*. A thread first peeks into the *taskQueue*, and invokes the *safe-source test* to check (using shared data reads) if any other core has the same task with a different priority. The task is allowed to proceed to execution only when it is either globally independent, or has the highest global priority order. The task is first removed from the local *taskQueue*, and synchronously removed from the global *orderList*. During execution, and depending on the algorithm, all data dependencies among the tasks being executed in all cores are resolved using atomic critical code sections. Moreover, new task (s) are produced and pushed into the task queues and order lists. Again, the global *orderList* is updated synchronously. Under a relax-ordered algorithm, the *safe-source test* is not performed, and thus the *orderList* is also not implemented. However, the *taskQueue* is maintained per core to

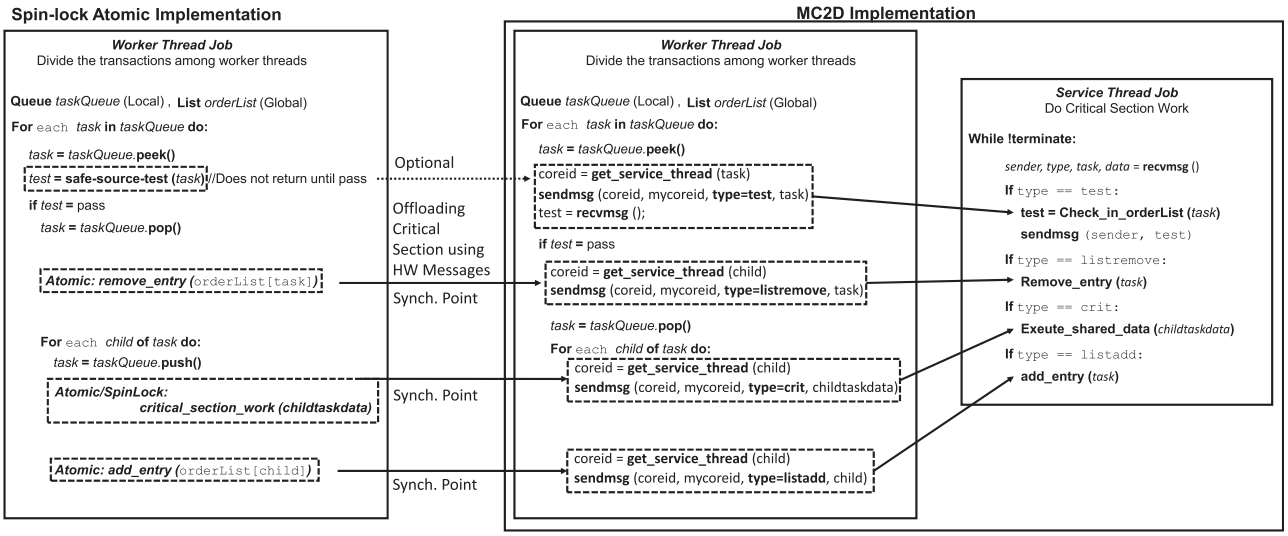

Figure 2. Generic framework construct outlining data structures and pseudocode requiring synchronizations for ordered task-parallel algorithms.

enforce locally ordered execution of tasks. These algorithms resolve the inter task dependencies in a monotonic manner to converge to their final solution. This is done by re-executing certain tasks when their data dependencies have not converged, thus increasing redundant work.

Although the pseudocode is shown for ordered algorithms, it is easily portable to unordered algorithms. Both *taskQueue* and *orderList* can be replaced with a simple per-core data structure, such as an array to schedule tasks for execution. The synchronizations in unordered algorithms only arise due to certain data dependencies among tasks, which are implemented using atomic critical code sections. Finally, an unordered algorithm may not even implement synchronization among tasks, and only synchronize threads from one phase of concurrent tasks to another phase. This results in barrier synchronization after all tasks within a given layer complete and propagate outputs to the next layer.

Each synchronization point discussed in the context of abstract constructs for ordered, relax-ordered, and unordered algorithms is instrumented for conversion to the MC2D model. Figure 2 shows an ordered algorithm's synchronization points as arrows from atomic to the MC2D implementation. The *safe-source test* is an optional conversion point since it only reads shared data values that can be done using traditional load instructions or the MC2D model. In MC2D case, data locality is optimal since shared data is pinned on service core(s), and in-hardware send and receive messages (using blocking communication) are utilized. The nonoverlapping regions of the *orderList* are pinned among the service core(s) based on a heuristic[2] that utilizes the profiled percentage of shared work to determine the right ratio of worker and service cores in the processor. The objective of the heuristic is to optimize load balancing of work done among all cores to maximize parallelism. All *orderList* update requests from each worker core are offloaded to the corresponding service core using (nonblocking) in-hardware messages. The MC2D model avoids expensive data movements for shared data, and thus exploits data locality at the service cores. Similar strategy is used for all shared data structures for each child task being processed by a parent task. This is shown as offloading the critical code section(s) from worker to service cores using in-hardware send and receive messages.

For relax-ordered and unordered algorithms, the *safe-source test* and the global task ordering (i.e., the *orderList*) are removed. However, the synchronization points are expected to be limited to one for the critical code section(s) within the task computations, and another at the completion of all tasks within a layer of computations (not shown in the figure). In summary, the MC2D model pins shared data at the service core(s), and exploits data locality to accelerate

thread synchronizations. Even for highly data-parallel algorithms that only require thread ordering on barrier synchronizations, the MC2D model is expected to perform on par with the traditional atomic synchronization model.

METHODOLOGY

Tilera's *Tile-Gx72* processor is used to evaluate various thread synchronization models against the MC2D model. The processor consists of 72 cores executing at 1 GHz, and includes a double data rate (DDR) main memory with 16 GB capacity. Each core implements a two-level cache hierarchy, where level-two shared cache is physically distributed among cores and interconnected using two-dimensional mesh networks. Directory-based hardware cache coherence enables data accesses between cores. The machine also enables core-to-core explicit messaging using its user-defined network (UDN). Four in-hardware UDN queues are integrated into each core to send and receive messages using a high level API library, Tilera Multicore Components (TMC).[2] All benchmarks are compiled by employing a modified version of GCC 4.4.7. The evaluation is performed using up to 64 cores.

Thread Synchronizations in *Tile-Gx72*

The MC2D model is implemented using the in-hardware messaging support. Each synchronization point in a shared-memory application is ported as outlined in the section "Task Parallelism Under the MC2D Model." All communication that does not use explicit messages is carried out under traditional hardware cache coherence load/store accesses. The following traditional synchronization models are also utilized for comparisons to the MC2D model.

Spin-Lock and Atomic Models: Tilera offers various atomic operations for efficient thread synchronization on shared data. Some of the operations are as follows: *cmpexch*, *fetchadd*, *fetchaddgez*, *exch*, to name a few. Compare–and–exchange (*cmpexch*) is utilized to build the widely applicable spin-lock synchronization model that can protect any arbitrary critical code section. When applicable, an atomic operation is directly used to implement the atomic model.

MC2D_shmem Model: MC2D_shmem is a shared-memory-only version of the MC2D model, which uses a shared software buffer per thread to enable messaging between worker and service cores. Although MC2D_shmem benefits from improved locality for shared data, it suffers from bouncing of the shared buffer between worker and service threads, which limits performance scaling.[3]

Benchmarks

Various task-parallel algorithms from diverse application domains of graph processing, machine learning, database, and data analysis are analyzed to show the applicability and portability of the MC2D model. Several graph algorithms, namely KCore, SSSP, A*, BFS, MST, and Color are considered as representative *ordered* and *relax-ordered* algorithms. These algorithms are ported from state-of-the-art ordered and relax-ordered parallelism works.[4,5,7] Their task-parallel implementations are implemented as outlined in Figure 2.

Several *unordered* algorithms are also considered. The triangle counting (TC)[8] graph algorithm, YCSB database workload,[9] and the SGD machine learning[10] are representative *unordered* algorithms that process tasks with thread synchronizations for task-level data dependencies. For example, YCSB processes transaction requests in parallel, but uses synchronized timestamp ordering to keep track of write accesses by using per-row write history tables. At commit for a request within a transaction, YCSB synchronously checks if reads of the current transaction overlap with 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.

Several highly parallel *unordered* algorithms are also considered that implement thread-level ordering across layers of task-parallel computations. These include PageRank, Community, and Conn. Comp.[8] graph algorithms, and deep neural networks, SqueezeNet[11] and GTRSB.[12] For example, SqueezeNet implements multiple neural computations per layer, where each layer processes its tasks in parallel across cores. Barrier synchronizations are implemented to propagate output neural values from layer to layer.

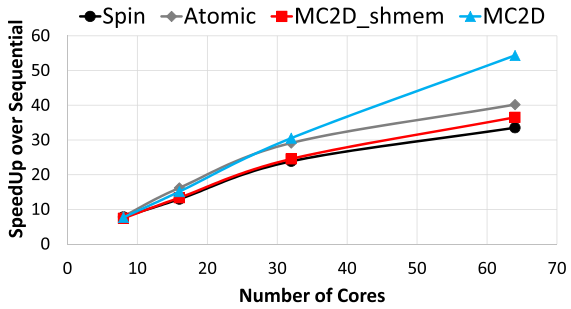

Figure 3. Average speedup of spin, atomic, MC2D_shmem, and MC2D as compared to sequential, at different core count.

For graph algorithms, three real-world graphs are used to explore input diversity. These are CAL from DIMACS,[II] LiveJournal from the Network Journal Repository,[III] and CAGE14 from SuiteSparse Matrix Collection.[IV] From CAL to CAGE14, the graph size and density increases while the diameter decreases. For GTRSB and SqueezeNet, an image is processed for inference from the ImageNet Repository.[V] In SGD, the real-sim[VI] input is used, which evaluates 20 958 features. YCSB implements access to database entries using Zipfian distribution. It includes a parameter called *theta* to control the contention level. Setting *theta* to 0.6 means that 10% of the database is accessed by 40% of all transactions. The *theta* value is varied from 0.6 to 0.9 with the increment of 0.05, then the average completion time is calculated using these *theta* values for performance comparison.

Evaluation Metrics

All evaluated algorithms are implemented using spin-lock, atomic, and both software-only and in-hardware MC2D models using the capabilities of the *Tile-Gx72* processor. All models utilize Pthreads library to spawn threads. Completion time is used as the evaluation metric, where all algorithms are run to completion, and only the parallel region is measured for performance analysis. The completion time of the worst case thread is broken down as nonsynchronization and synchronization components. For the spin-lock and atomic models, synchronization is measured as the time spent in the atomic operation, as well as time spent in the critical code section. The remaining thread local computation accounts for the nonsynchronization time. However, for the MC2D model, the synchronization time accounts for the time spent in completing each *send* and *receive* message. The tmc_udn_send_n() routine is used to send request messages to the service threads, where the message is placed into a core specific hardware queue, and then the send instruction completes. This nonblocking nature of send messages allows the MC2D model to offload critical section work, and the worker thread can overlap computation with synchronization. However, the tmc_udn0_receive() routine implements receive messages in a blocking manner. Hence, the time taken by critical code section, and message traversal is accounted when the receive completes. From a worker thread perspective, the time taken to complete all critical section work is implicitly accounted via receive messages.

EVALUATION

The MC2D model is anticipated to mitigate synchronization bottleneck as the number of cores increases per chip. Therefore, the spin, atomic, MC2D_shmem, and in-hardware MC2D models are evaluated at 8, 16, 32, and 64 threads by pinning a single thread per core. The average speedup is measured for all benchmarks over a sequential implementation optimized for single thread performance. Figure 3 shows the average speedup for each thread synchronization model as the core count increases. The in-hardware MC2D model demonstrates superior performance scaling. The atomic model keeps up with the MC2D model until low core counts (less than 32), but the performance gap rapidly increases to 33% at 64 cores. The spin and MC2D_shmem models both show diminishing performance scaling since they both suffer from increasing overheads of cache line ping-pongs and instruction retries. MC2D_shmem is slightly better than spin due to its locality optimizations for shared data, but the shared buffer used to communicate between worker and service cores still ping-pongs. The in-hardware MC2D model delivers

[II] http://users.diag.uniroma1.it/challenge9/download.shtml.
[III] http://networkrepository.com/livejournal.php.
[IV] https://sparse.tamu.edu/vanHeukelum/cage14.
[V] http://www.image-net.org/.
[VI] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

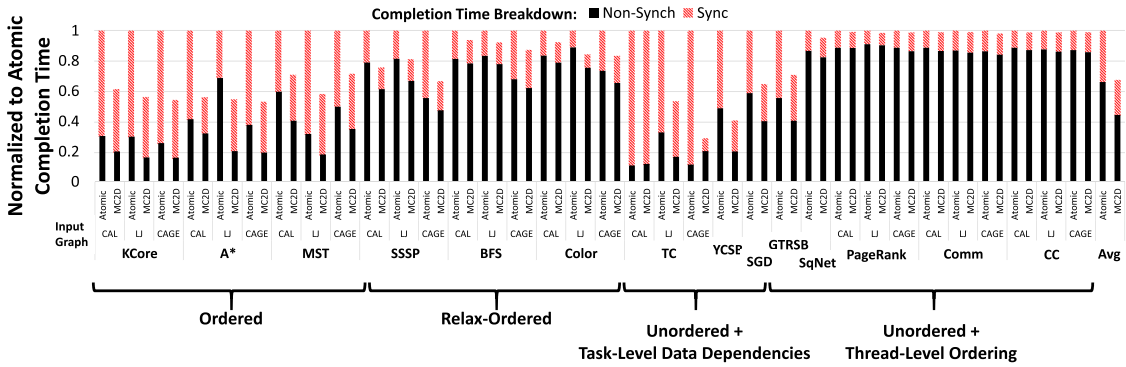

Figure 4. Normalized completion time breakdown results of MC2D over the atomic model.

near-optimal access to all shared data pinned at the service cores, and even surpasses the atomic model when the on-chip network becomes a bottleneck at high core counts. Therefore, the remaining evaluation focuses on the MC2D and the atomic model comparisons.

Figure 4 shows a per benchmark and input evaluation of MC2D against the atomic model, where the *y*-axis shows the completion time normalized to the atomic model. Furthermore, the completion time is broken down into nonsynchronization and synchronization components. Consequently, performance gain is calculated as the percentage decrease in completion time for the MC2D model relative to the atomic model. The ordered benchmarks are classified into *ordered* and *relax-ordered* implementations based on their performance under the atomic model. *Ordered* benchmarks with the MC2D model consistently deliver 30%–65% decrease in completion time compared to the atomic model. The nonsynchronization time improves due to the MC2D model offloading the critical code sections to service cores, thus reducing the code executed on the worker cores. The synchronization time is observed as a significant component of the completion time (more than 50% on average) for the *ordered* benchmarks. For example, the *safe source test* needs to wait on synchronization to process the next task in each thread, which increases its significance in terms of accelerating synchronization. It improves due to the MC2D model taking advantage of core-level shared data locality, and avoid unnecessary cache line ping-pongs between cores. Moreover, the nonblocking nature of the MC2D model allows it to overlap computation with communication. Therefore, significant improvements are observed in synchronization times for the MC2D model over the atomic model. The *relax-ordered* benchmarks observe smaller benefits from the MC2D model. Relaxed task ordering reduces synchronizations needed to order tasks globally across cores. However, more work is done in each core to converge these algorithms to their solutions. This results in a higher nonsynchronization component for these benchmarks. The MC2D model still improves performance by accelerating synchronizations that resolve shared data dependencies among tasks, as well as barrier synchronizations across algorithmic iterations. SSSP has relatively large critical code sections compared to BFS and Color benchmarks. Therefore, SSSP observes higher performance benefits with the MC2D model.

Figure 4 also shows the evaluation for *unordered* benchmarks, which are separated into two synchronization categories. The *unordered* benchmarks with task-level data dependencies exhibit significant synchronizations that must be handled within a task's execution. TC consists of tasks that are dominated by synchronization work. Hence, as graph density increases from CAL to CAGE, stress on synchronizations also increases since each parent task synchronously updates an increasing number of child/new tasks. Therefore, most cores are assigned as service cores in TC. The shared data locality also increases at the service cores as graph density increases due to increasing locality in edges. CAL does not observe any benefits in both components because the graph is sparse (on average 1.2 child/new tasks per parent) and exhibits random edge

connectivity. Therefore, under MC2D model, worker cores tend to not have enough computations to overlap communication. Moreover, the synchronization updates to random edges do not offer shared data locality benefits under the MC2D model. On the other hand, CAGE is a dense graph, and exposes data locality on edges. The worker cores now have sufficient computations to overlap communication, and the service cores demonstrate improved shared data locality. Therefore, the MC2D model improves the synchronization component significantly, but it comes at the cost of increased nonsynchronization time due to reduced parallelism (i.e., fewer worker cores). The nonsynchronization time in atomic model is better than the MC2D model because it has more cores available to perform the thread-local computations.

This article evaluates the applicability of the MC2D model to accelerate synchronizations in task parallel algorithms. The evaluation shows that improving shared data locality enables the MC2D model to deliver an average of 33% performance gains over the atomic model.

In YCSB, the critical code sections for each task are much larger, hence the importance of accelerating synchronizations increases as thread contention increases with *theta* values. The reported YCSB result is an average of *theta* values from 0.6 to 0.9. SGD also improves with the MC2D model, where evaluated outputs that require atomic writes on the minimization function are pinned to service cores.

The *unordered* with thread-level ordering benchmarks generally perform a significant amount of thread-parallel work. These benchmarks use barrier synchronization as all threads propagate their shared values from one layer to the next layer of computation. Therefore, synchronization costs for these workloads is low as depicted by the completion time breakdown, and hence these benchmarks show little benefits from accelerating synchronization. However, the two machine learning benchmarks, GTRSB and SqNet show performance improvement. The GTRSB benchmark performs much less work between barrier synchronizations as compared to SqNet. Therefore, it shows more gains from accelerating barrier synchronization using the MC2D model as compared to the atomic model.

On average, the MC2D model outperforms the atomic model by 33%. When *unordered* benchmarks with thread-level ordering are discounted, this average decrease in completion time increases to 48%.

CONCLUSION

This article evaluates the applicability of the MC2D model to accelerate synchronizations in task parallel algorithms. The evaluation shows that improving shared data locality enables the MC2D model to deliver an average of 33% performance gains over the atomic model. These benefits directly correlate with the number and frequency of synchronizations that are observed in both *ordered*, as well as *unordered* algorithms with task-level data dependencies. This work also shows that the MC2D model unlocks parallelism for highly parallel algorithms from the *unordered* category, and delivers at par performance with the atomic model.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Grant CNS-1718481. This research was also partially supported by the Semiconductor Research Corporation (SRC), and NXP Semiconductors. The authors wish to thank C. Hughes of Intel and B. Kahne of NXP for their continued support and feedback.

REFERENCES

1. 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 *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 254–264.
2. H. Dogan, M. Ahmad, J. Joao, and O. Khan, "Accelerating synchronization in graph analytics using moving compute to data model on tilera TILE-Gx72," in *Proc. IEEE 36th Int. Conf. Comput. Design*, 2018, pp. 496–505.

3. H. Dogan, M. Ahmad, B. Kahne, and O. Khan, "Accelerating synchronization using moving compute to data model at 1,000-core multicore scale," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 1, Feb. 2019, Art. no. 4.
4. M. A. Hassaan, D. D. Nguyen, and K. K. Pingali, "Kinetic dependence graphs," in *Proc. 20th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2015, pp. 457–471.
5. D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. 24th ACM Symp. Oper. Syst. Principles*, 2013, pp. 456–471.
6. J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications," in *Proc. USENIX Conf. Annu. Techn. Conf.*, 2012, USENIX Association, Berkeley, CA, USA, pp. 6–6.
7. L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proc. 29th ACM Symp. Parallelism Algorithms Archit.*, 2017, pp. 293–304.
8. M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *Proc. IEEE Int. Symp. Workload Characterization*, 2015, pp. 44–55.
9. X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," in *Proc. VLDB Endowment*, Nov. 2014, vol. 8, pp. 209–220.
10. M. A. Zinkevich, M. Weimer, A. Smola, and L. Li, "Parallelized stochastic gradient descent," in *Proc. 23rd Int. Conf. Neural Inf. Process. Syst.*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds., Vol. 2. Red Hook, NY, USA: Curran Associates Inc., 2010, pp. 2595–2603.
11. F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and $<$1 MB model size," 2016, *arXiv:1602.07360*.
12. P. Sermanet and Y. LeCun, "Traffic sign recognition with multi-scale convolutional networks," in *Proc. Int. Joint Conf. Neural Netw.*, 2011, pp. 2809–2813.

Masab Ahmad is currently a Senior Silicon Design Engineer with AMD Research. He received the Ph.D. degree in computer engineering from the University of Connecticut, Storrs, CT, USA. His research interests include parallel computing, computer architecture, and workload characterization. He is a member of IEEE. Contact him at masab.ahmad@uconn.edu.

Halit Dogan is currently a Software Architect with Intel, Santa Clara, CA, USA. He received the Ph.D. degree in computer engineering from the University of Connecticut, Storrs, CT, USA. His research interests include improving intra and inter node communication in high performance computing. Contact him at halitdoganeem@gmail.com

José A. Joao is currently a Staff Research Engineer with the Architecture Group, Arm Research, Austin, TX, USA. He received the Ph.D. and MS degrees in computer engineering from the University of Texas, Austin, TX, USA, where he was supervised by Professor Yale Patt. He also holds an Electronics Engineering degree from Universidad Nacional de la Patagonia San Juan Bosco, Argentina, where he was an Assistant Professor. His current research interests are high-performance energy-efficient scalable system architectures for HPC and server workloads. He is a member of the IEEE and the ACM. Contact him at jjoao@ieee.org.

Omer Khan is currently an Associate Professor with the Department of Electrical and Computer Engineering, University of Connecticut, Storrs, CT, USA. He received the Ph.D. degree in electrical and computer engineering from the University of Massachusetts Amherst, Amherst, MA, USA. Prior to joining UConn, he was a Postdoctoral Research Scientist with the Massachusetts Institute of Technology, Cambridge, MA, USA. His research interests include developing cross-layer methods to improve the performance scalability and security of multicore processor architectures. He is a member of IEEE and the ACM. Contact him at khan@uconn.edu.