Distributed Shared Memory for Machine Learning

Amin Tootoonchian
Intel Labs

Aurojit Panda
NYU, ICSI

Aida Nematzadeh
UC Berkeley

Scott Shenker
UC Berkeley, ICSI

1 INTRODUCTION

Distributed systems communicate and coordinate through *message passing* or *shared memory*: in the former paradigm, threads communicate by exchanging messages, whereas, in the latter, they communicate by reading from and writing to directly accessible shared memory. Due to the need for large-scale machine learning, most existing frameworks (such as TensorFlow [1], PyTorch [7], and Caffe [4]) implement distributed machine learning where workers rely on shared memory for local communication and message passing (*e.g.,* using BSD sockets) for inter-node communication.

Many multiprocessing systems provide a shared memory abstraction. Recent theoretical and practical results [6, 8, 9] suggest that well-designed shared-memory implementations of algorithms such as stochastic gradient descent (SGD) can achieve impressive speed-ups when compared to more traditional implementations. Besides providing performance benefits, shared memory abstractions also simplify application development by freeing developers from reasoning about communication mechanisms (*e.g.,* buffer management, message construction, naming, timing of sends and receives). Despite these benefits, no current ML framework uses shared memory across machine boundaries, since hardware shared memory is not available in this setting.

The key challenge in building a software distributed shared memory (DSM) is achieving efficiency while providing an interface that is easy to use for the programmer. Distributed shared memory implementation commonly have three properties: *(a)* cache coherency, *(b)* location transparency, and *(c)* need for synchronization for safe concurrent access. They make scaling challenging.

We present an alternative DSM design, Tasvir, which relaxes these guarantees thus sidestepping the associated shortcomings while providing an interface that is well-aligned with the requirements of machine learning algorithms. It has the potential to simplify state sharing and improve data movement across a large pool of distributed workers. We show that on a multicore machine, it performs similarly to or, in some cases, surpasses the performance of hardware shared memory.

2 ML-FRIENDLY SHARED MEMORY

Our design provides the following interface to the user:

Versioned areas: Area is the unit of memory allocation and is contiguous. Areas are atomically updated (no torn writes), versioned, and timestamped. The latter enables threads to quantify the staleness of cached copies. At least two versions of an area are hosted on a machine: a writer and a reader version.

Ownership: Each area has a single writer which is internally marked as the owner. Area ownership could be transferred.

Membership: Threads must subscribe to areas to start receiving a continuous stream of updates.

Explicit synchronization: Threads are responsible for regularly invoking a service routine at points in the application where it is safe to do (*e.g.,* outside the critical sections). At scheduled intervals, the service routine blocks to initiate a synchronization to transfer and/or apply pending updates. Threads may force a synchronization if they want a local write to be immediately made visible.

Explicit write logging: Threads must inform the system of any writes to an area. This may be automated with a compiler pass.

Synchronization intervals: Threads may specify the interval at which internal synchronization (cross-core) and external synchronization (cross-machine) must be invoked.

This interface aligns well with recent trends in machine learning research: It embraces staleness as a first-class primitive [3, 8], simplifies implementing bounded staleness [3, 5], and avoids coherency issues [10]. It also enables an efficient DSM implementation and a wide range of performance optimizations that are otherwise hard to achieve.

Being explicit about state caching enables the system to use local memory as cache (akin to COMA [2]) without a need to update or invalidate for individual writes. Applications may decide to block and continuously synchronize until they achieve a desired level of freshness, but during normal operation no read or write is impeded because of concurrency or distribution – *i.e.,* they are all served at local memory speed. The notion of single-writer areas allows us to commit writes with no concern for conflicts.

Change tracking at the memory level enables cooperative internal synchronization and offloading external synchronization to be done asynchronously by a background thread. During an internal bulk synchronization phase, threads cooperatively parse the log and copy modified data from writer copies to reader copies. It is during this internal synchronization that area version numbers and timestamps are updated if any change is found. External synchronizations are invoked asynchronously by a background thread in-between internal ones to packetize and push differential updates to remote nodes subscribed to receive the updates. Upon receipt of remote updates, the same background thread logs the update and transfers it to the local writer copy. Such changes will be made visible upon initiation of the next internal synchronization.

Scoping updates through area memberships and synchronization periods are knobs that we expose to help applications meet a target scale, staleness, and state distribution overhead. Coarser synchronization periods result in lower update traffic and better scaling at the expense of higher staleness (and possibly worse convergence), while limiting area membership to neighboring nodes may help localize update traffic to sidestep network bandwidth bottlenecks.

Our design provides several benefits for machine learning frameworks and applications. First, it simplifies development and improves performance by taking over networking and data movement. This is possible because of numerous hardware and network optimization, *e.g.,* it speed up data movement using non-temporal reads/writes and vector instructions, and reduces the overhead of network transfers using performant network stacks[1]. Our current

[1]Our implementation currently uses DPDK, but our design also enables RDMA to be seamlessly incorporated.

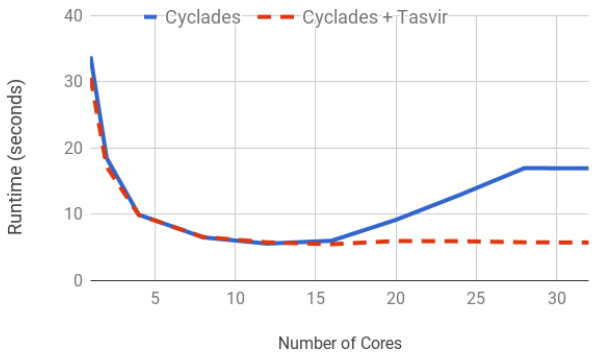

Figure 1: Runtime for Cyclades using hardware shared memory (Cyclades) and Tasvir (Cyclades + Tasvir) as a function of threads.

implementation provides intra-machine data transfer that is comparable to DRAM bandwidth, and line-rate (10/40Gbps) network transfer for cross-machine transfers. Second, compared to hardware shared memory, our design provides developers with greater flexibility in how writes from multiple workers are combined – whereas hardware shared memory traditionally only provides last-writer-wins semantics. Tasvir allows developers to implement a variety of aggregation policies including ones which average writes from different workers, add up writes from different workers, etc. Finally, Tasvir enables ML researchers and developers to scale single machine algorithms (such as HogWild!) to multiple machines without requiring significant algorithmic or development changes, thus simplifying the development and deployment of new ML applications. That is because our design maintains the same interface regardless of whether workers are placed in the cores of the same machine or distributed across machines.

We implemented Tasvir as a library and runtime daemon in around 2500 lines of `C`. In the next section, we discuss how we modified an existing stochastic gradient descent implementation to work with Tasvir instead of native shared memory.

3 SGD EXAMPLE

To give a sense of changes required to adopt Tasvir, we briefly discuss how we modified a shared-memory SGD implementation. We also use this implementation to evaluate Tasvir's overhead when compared to a native hardware shared memory implementation.

While Tasvir provides a high-level interface compared to message passing, its main programming interface (raw memory) is not convenient for many applications. Therefore, we often need to build distributed data structures on top that abstract away distribution details from the application and embed the desired policies. A distributed array data structure is a good fit for SGD. There are different ways one could build such a data structure; we describe a simple one that we built and experimented with below.

Allocation: Each thread requests creation of a Tasvir area with itself as the owner. The master thread creates a master area in addition to its worker area. Master thread updates the data structure with the addresses of slave areas as they appear. Once the master thread finishes initialization of the data structure, all threads proceed.

Versioning: Equivalent to a logical clock, a version number is maintained per worker in the data structure that reflects how shows which stage of computation the worker is at. We use this version number to implement barriers for synchronous SGD.

Barrier: The master busy-waits until all workers reach the current version, and then progresses to the next version. The worker threads block until the master thread proceeds to the next version. This barrier could be modified to implement bounded staleness [3, 5] by letting workers proceed if they are apart at most by a given bound.

Map: Receives information about the batch to be processed next as input and copies data from the master area to the worker area.

Reduce: Copies the updated data back to the master area. Policy for update averaging could be implemented here if multiple threads work on the same parameters.

Using this data structure, replacing a shared-memory SGD implementation with a Tasvir-based one is straightforward. We chose Cyclades [6] because its codebase is publicly available, implements a wide range of models and updaters, and includes datasets for evaluation and comparison. More importantly, it is a good baseline to measure Tasvir's overhead; Cyclades partitions datapoints such that updates to the same parameter do not to overlap, which results in a notable improvement over Hogwild!.

At a high-level, our modifications to Cyclades are as follows. We replaced the model data with the above described array. At the start of processing each batch, the `Barrier` method followed by the `Copy` method is invoked. At the end of processing each batch, the `Reduce` method is called to copy the updated parameters back to the master. We use another Tasvir array to distribute the computation of loss function.

We expected Tasvir-based Cyclades to perform notably worse than Cyclades because *(a)* we add a lot of local data movement, and *(b)* Cyclades should already have minimal coherency traffic (except false sharing). We experimented with a simple test: matrix completion using sparse SGD on the Movie Lens 1M dataset with a minibatch size of 2000 and learning rate of 0.02. We note that both implementations are functionally identical and yield the same result for synchronous SGD. Surprisingly, we observe that Tasvir-based Cyclades performs very closely or in some cases significantly better than the native shared-memory implementation (see Figure 1). This is despite ~1000 synchronizations per second resulting in ~3.5GB/s of data movement. Interestingly, with an increase in the number of threads Cyclades runtime almost triples whereas Tasvir maintains a similar runtime with higher thread count.

4 CONCLUDING REMARKS

While it has long been observed that shared memory can simplify programming, this approach has so far not been adopted due to a lack of hardware implementations and concerns that large-scale software DSMs might be inefficient. We have demonstrated through evaluation on a simple prototype implementation that our software DSM implementation imposes little overhead compared to a native shared-memory implementation on a single machine. We plan to evaluate performance at scale, and ultimately hope to provide an efficient yet simple to use, machine-learning-friendly shared memory implementation.

REFERENCES

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Dahlgren, F., and Torrellas, J. Cache-only memory architectures. *IEEE Computer 32* (1999), 72–79.

[3] Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J. K., Gibbons, P. B., Gibson, G. A., Ganger, G. R., and Xing, E. P. More effective distributed ml via a stale synchronous parallel parameter server. *Advances in neural information processing systems 2013* (2013), 1223–1231.

[4] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R. B., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *ACM Multimedia* (2014).

[5] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *OSDI* (2014).

[6] Pan, X., Lam, M., Tu, S., Papailiopoulos, D. S., Zhang, C., Jordan, M. I., Ramchandran, K., Ré, C., and Recht, B. Cyclades: Conflict-free asynchronous machine learning. *CoRR abs/1605.09721* (2016).

[7] PyTorch. https://github.com/pytorch/pytorch, retrieved 01/05/2017.

[8] Recht, B., Ré, C., Wright, S. J., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS* (2011).

[9] Sa, C. D., Zhang, C., Olukotun, K., and Ré, C. Taming the wild: A unified analysis of hogwild!-style algorithms. *Advances in neural information processing systems 28* (2015), 2656–2664.

[10] Zhang, C., and Ré, C. Dimmwitted: A study of main-memory statistical analytics. *Proc. VLDB Endow. 7*, 12 (Aug. 2014), 1283–1294.