Efficient and Programmable Machine Learning on Distributed Shared Memory via Static Analysis

Jinliang Wei
Carnegie Mellon University
jinlianw@cs.cmu.edu

Garth A. Gibson
Carnegie Mellon University and Vector Institute
garth@cs.cmu.edu

Eric P. Xing
Petuum Inc. and Carnegie Mellon University
epxing@cs.cmu.edu

ABSTRACT

Distributed shared memory (DSM) offers superior performance for applications that perform fine-grain reads and writes to in-memory variables, such as iterative machine learning (ML) training, but presents a great challenge for application developers due to data dependency over shared mutable states. Thus iterative machine learning training is often parallelized by replicating the same computation on different data partitions (a.k.a. data parallelism) ignoring data dependency. Since ML training can often tolerate bounded error [7], such parallelization may still produce a working solution at the cost of additional computation.

In many cases, preserving data dependency greatly reduces the computation needed to achieve the same model quality without harming computation throughput. In this paper, we show that such opportunity may be exploited with minimal programmer effort via static analysis. We present a system called Orion that statically parallelizes serial for-loop nests, which read and write distributed shared memory and schedules computation on a distributed cluster, preserving fine-grained data dependency. Orion may ignore certain dependences (given programmer permission), potentially falling back to data parallelism, when preserving all data dependency results in a serial execution. We show that a machine learning training program parallelized by Orion may get a 3.5× speedup compared to a data-parallel implementation based on parameter servers due to preserving data dependency, while enjoying a much more usable programming model.

1 INTRODUCTION

Distributed shared memory (DSM) [9, 11, 18] provides a global address space for objects distributed among a cluster of nodes' memory. By abstracting away network communication, it provides a simple programming model for application developers. More importantly it offers efficient fine-grained reads and updates when there are indices on the globally shared variables. This is particularly suitable for machine learning training as it often performs frequent sparse reads and updates over a large number of in-memry parameters. Indeed the superior performance of distributed shared memory for machine learning training has been demonstrated by various parameter server systems [5, 12, 19]. Applications on DSM systems, including parameter servers and Piccolo [17] are usually implemented as a program that runs on each worker node and the worker programs share state via DSM. Programming an efficient DSM application is still challenging for a number of reasons. The biggest challenge arises from data dependency over shared mutable states, which occurs when two instructions access the same variable and one of them is a write.

DSM systems typically do not provide concurrency control over the shared variables, and thus it's left to the application programmer to properly parallelize the serial computation to a distributed program. Different parallelization could result in vastly different convergence rate for ML training. A ML common practice referred to as data parallelism simply replicates the same computation over different data partitions, ignoring any data dependency and thus achieving high computation throughput. However, conflicting writes may occur if two workers write to the same variable and workers might read stale values as the most up-to-date values are buffered by other workers. Since iterative machine learning training tolerates bounded error, data parallelism may still produce a working solution but may take additional iterations to reach the same solution quality compared to when data dependency is preserved [7, 10, 19]. In STRADS [10], application programmers are expected to design and implement a computation scheduling strategy that maximally preserves data dependency without severely reducing computation throughput. While a proper scheduling strategy may significantly improve ML training convergence time, coming up with such a strategy imposes nontrivial burden on STRADS users.

In data-flow systems such as Dryad [8], Spark [21], Naiad [15] and TensorFlow [2], dependency is described as a Directed Acyclic Graph (DAG) where nodes are operators and edges are the operators' inputs and outputs. Extracting parallelism is simple with such an explict dependency representation. However, explicitly representing operators and their inputs and outputs in a DAG fundamentally forbids the DAG from capturing fine-grained dependencies as doing so would result in an intractably large graph. Deep learning systems such as TensorFlow [2] provide parallelized operators to utilize all cores in one GPU, but parallelizing the whole DAG across multiple GPUs still relies on users and data parallelism is often employed for simplicity.

2 SYSTEM DESIGN AND EXPERIMENTS

We propose a system Orion for ML researchers/practitioners who invent new models or algorithms and need to scale them out. Orion provides an application library and a runtime which consists of a set workers and a master for coordination. Application programmers implement a Juila [4] driver program which uses primitives provided by the application library to initiate computation to be performed by the runtime. In Orion, a dataset is represented as a Distributed Array, a.k.a. DistArray, which is an n-dimensional matrix, partitioned across multiple workers. DistArrays can store objects of arbitrary serializable type. Similar to RDDs, DistArray supports transformations (set operations such as map and groupBy) that transforms one DistArray to another. Unlike RDD, DistArray

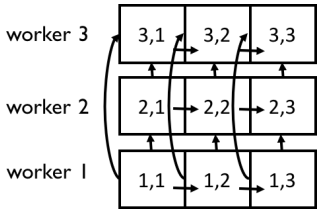

Figure 1: the 3×3 iteration space of a two level loop nest; arrows denote dependence vector; this loop is parallelized by assigning each outer loop iteration (row) to a worker.

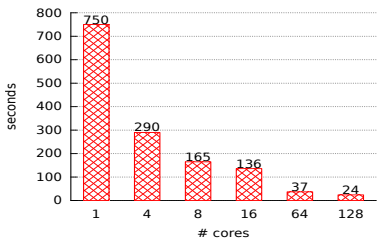

Figure 2: per-data-pass execution time of SGD matrix factorization on various number of cores

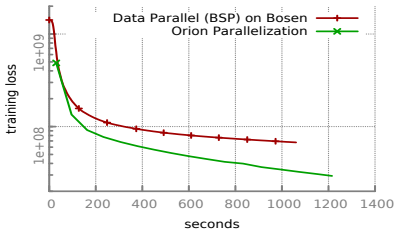

Figure 3: comparing SGD matrix factorization parallelized by Orion with a data-parallel implementation on Bösen

supports point and range query on matrix elements, and more importantly, in-place updates. When such a fine-grained operation is invoked, Orion builds an index for that DistArray if one doesn't already exist, and the index can be either dense or sparse.

The driver program may create and transform DistArrays and access its elements. It may also iterate over a DistArray in a loop. The loop body may read and write any DistArray that is created before the loop. Such loops are widely used in machine learning training programs where the training algorithm often iterates over the training dataset, model parameters or model dimensions. Since a DistArray is an n-dimensional matrix, such a loop is essentially a perfectly nested loop and the matrix is the iteration space. By default, such a for-loop executes serialy on the driver. The programmer may parallelize it among the distributed workers by applying the `@parallel_for` macro. Many language tools such as OpenMP [6] and MATLAB [1] also provide a parallel for-loop construct. The major difference is that Orion can parallelize loops with loop-carried dependences (i.e. dependency among loop iterations) while preserving data dependency for DistArray accesses. Moreover, Orion targets distributed execution while traditional parallelization algorithms target multicore shared-memory machines.

Opportunities for parallelization exist when data dependency is sparse, i.e. when computation performs sparse accesses to shared states. In many cases, such sparsity can be exposed via static analysis. When array access subscripts are linear combinations of the loop induction variables (i.e. matrix indices), dependency among loop iterations can be represented by dependence vectors [14, 20]. There exists a dependence vector from node i to node j in the iteration space if iteration j depends on iteration i.

In many ML training algorithms, even when dependency exists among loop iterations, the loop iterations may be executed in any arbitrary serial order. For example, Stochastic Gradient Descent (SGD) may process training data in any serial order, event though different execution order may produce different, but equally valid results (don't-care-nondeterminism [16]). Eliminating ordering constraints may expose additional opportunities for parallel execution. For example, in Fig. 1, if there were no ordering constraints, worker 2 may start executing iteration (2, 2) or (2, 3) while worker 1 is executing iteration (1, 1). This is been not possible with the ordering constraint as iteration (2, 2) can only be executed after (2, 1).

After the dependence vectors are computed, Orion tries to identify and parallelize cases (e.g. Fig. 1) that can take advantage of iteration reordering. If there exist two dimensions i and j in the iteration space such that for any dependence vector d, either $d[i] = 0$ or $d[j] = 0$, the loop can be parallelized by partitioning the iteration space along i and j and assigning all iterations with the same i to the same worker. As long as we ensure iterations executing at the same time have different values for both i and j, no two workers can access the same variable at the same time. To handle the more general cases, Orion employs a classic compiler parallelization algorithm [20]. This algorithm transforms the iteration space to a loop nest of m levels of outer loops and n levels of inner loops such that no dependence is carried by the inner loops. Thus for each outer loop iteration, its inner loops can be executed in parallel.

When the dependency is dense or when dependence vectors do not accurately capture the fine-grained dependency, Orion may fail to parallelize the loop, resulting in a serial execution. In this case, the program may declare a buffer for a DistArray and write to the buffer instead. The buffered updates are applied to DistArray by Orion in a best effort manner. This may eliminates some dependency, permitting parallelization.

In a distributed program, each read on DistArray may cause a remote procedure call (RPC) to fetch a value over network. Orion tries to identify DistArrays that can be pipelined among workers to eliminate random accesses over network. For DistArrays that cannot be pipelined, Orion synthesizes a function that computes the DistArray access subscripts to prefetch values in bulk.

To demonstrate Orion's effectiveness, we present an SGD matrix factorization application on Orion, which is implemented in 70 lines of Julia code, while implementations on parameter servers [5, 19] and GraphLab [13] take 300-400 lines of code. Our experiments were conducted on a cluster of up to 12 nodes, which each contains 16 physical cores (Intel E5-2698Bv3 Xeon) and 64 GiB of RAM, and used the Netflix [3] dataset (rank is set to 1000). Fig. 2 shows the per-data-pass execution time when the same algorithm is parallelized across different number of CPU cores. Orion's parallelization gains a 5.5× speedup using 16 cores on 1 machine and a 31× speedup using 128 cores on 8 machines compared to serial execution, without harming per-data-pass convergence progress. Fig. 3 compares Orion's parallelization with a data-parallel (BSP) implementation on Bösen [19] using 12 nodes. Even though Orion is 6× slower than Bösen in terms of per-data-pass execution time, it reduces the time spent to achieve the same model quality by 3.5× thanks to preserving data dependency.

REFERENCES

[1] MATLAB Parfor Documentation. https://www.mathworks.com/help/distcomp/parfor.html/.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[3] J. Bennett and S. Lanning. The netflix prize. *In KDD Cup and Workshop*, 2007.

[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[5] H. Cui, A. Tumanov, J. Wei, L. Xu, W. Dai, J. Haber-Kucharsky, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting iterative-ness for parallel ml computations. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 5:1–5:14, New York, NY, USA, 2014. ACM.

[6] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.

[7] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1223–1231. Curran Associates, Inc., 2013.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[9] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: High-performance all-software distributed shared memory. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 213–226, New York, NY, USA, 1995. ACM.

[10] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing. Strads: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 5:1–5:16, New York, NY, USA, 2016. ACM.

[11] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[12] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, Oct. 2014. USENIX Association.

[13] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[14] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 1–14, New York, NY, USA, 1991. ACM.

[15] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[16] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[17] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[18] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel Distrib. Technol.*, 4(2):63–79, June 1996.

[19] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 381–394, New York, NY, USA, 2015. ACM.

[20] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):452–472, Oct. 1991.

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.