STRADS-AP: Simplifying Distributed Machine Learning Programming

Jin Kyu Kim[1] Garth A. Gibson[1,2] Eric P. Xing[3,1]
[1]Carnegie Mellon University, [2]Vector Institute, [3]Petuum Inc.

1 INTRODUCTION

STRADS-Automatic Parallelization(AP) is a distributed framework that automatically parallelizes sequential machine learning (ML) programs to execute in a cluster. STRADS-AP offers a set of memory container data structures with fine-grained read/write access and two parallel loop operators that schedule parallel loop bodies with strong[10, 14, 17, 19, 22] or relaxed[6, 12, 18, 24] consistency. The STRADS-AP APIs allow ML programmers to write sequential code for a ML algorithm and easily achieve good parallel performance.

The STRADS-AP runtime is responsible for partitioning input data and model parameters, parallelizing slices of ML computations across a cluster, and enforcing strong consistency on shared data or synchronizing partial outputs with relaxed consistency. The STRADS-AP runtime frees ML programmers from low-level details of distributed ML programming: 1) concurrency control for shared data if any exists; and 2) synchronization of partial model parameter updates across a cluster.

STRADS-AP is implemented as a C++ template library. In this extended abstract, we present our programming model applied to two well-known ML applications, word2vec[11, 13, 20, 21] and stochastic gradient descent matrix factorization(SGDMF)[15].

2 PROGRAMMING MODEL

STRADS-AP offers a set of distributed data structures (DDSs) including dvector, dmap, and dmultimap. For data loading and parameter initialization, STRADS-AP provides data processing operators such as map, reduce, transform, join, create, and store that operate on DDSs. ML programmers declare DDSs for storing large input data and parameters and process them using these data processing operators. For ML training, STRADS-AP offers loop operators that take a user-defined function that specifies ML optimization computations as a loop body. This user-defined function is allowed to read/write individual DDS elements. ML programmers write visually simple, straightline code as a driver program using these APIs. STRADS-AP offers two parallel-for operators that support two popular parallel ML execution models, asynchronous and synchronous models.

Async_Parallel_For: takes three arguments: the start index S, the end index $S+N$, and the C++ lambda function [captures]F(const int index){ code for lambda function body}. The for-loop function executes $N+1$ lambda function instances, $F(S), F(S+1), .., F(S+N)$ concurrently. For simplicity's sake, we let F_i denote $F(i)$. Programmers are allowed to capture arbitrary variables in the current scope in the capture list[] of F and specify a parameter update routine as the function body of F. The one constraint in capturing variables is that only DDS type variables can be captured by reference; that is, non-DDS type variables must be captured by copy. At runtime, STRADS-AP partitions the index range$(S, S+1, .., S+N)$ into P non-overlapping chunks C_p, and schedules up to P workers to concurrently execute F with different index chunks. A worker W_k sequentially runs $|C_k|$ lambda function instances, F_i with $i \in C_k$ allowing(and scheduling) reads/writes to global DDS elements(DDS typed variables). If the lambda function body modifies a DDS type variable captured by reference, read/write or write/write data conflicts might happen. The STRADS-AP runtime avoids data conflicts using dependency-aware scheduling and improves statistical efficiency. This dependency-aware scheduling is effective for data conflict-sensitive algorithms(i.e. coordinate descent lasso[14, 17, 25]) where data conflict-allowing codes might sacrifice more statistical progress than is needed for the throughput increase achieved. Therefore, STRADS-AP emphasizes good statistical progress wherever possible. For demonstration purposes, consider SGDMF[15]. The core computation routine of SGDMF can be expressed in about 50 lines in STRADS-AP . The following shows the main part of STRADS-AP SGDMF.

```
typedef rate T1; // struct rate {int i, int u, float r}
typedef array<float, K> T2;
dvector<T1> &R=CreateVector<T1>(path,parser);//DDS for input data
dvector<T2> &P=CreateVector<T2>(maxi, initf);//DDS for parameters
dvector<T2> &Q=CreateVector<T2>(maxu, initf);//DDS for parameters
float alpha(.01f), lambda(.1f);
for(auto i(0);i<maxiter;i++){
  Async_Parallel_For(0,R.size()-1,[alpha,&R,&P,&Q](int i){
                rate &r=R[i];
                T2 er = r - P[r.u]*Q[r.i];
                Q[r.i]+= alpha*(er*P[r.u]-lambda*Q[r.i];
                P[r.u]+= alpha*(er*Q[r.i]-lambda*P[r.u];});
}
```

Note that the STRADS-AP runtime is responsible for addressing data conflict problems on Q and P so programmers can focus on writing ML computation routines without implementing code to resolve and avoid data conflict problems.

Sync_Parallel_For: takes four arguments: input data of DDS<T> type, size of mini-batch M, a lambda function [captures]F(const vector<T> &minibatch){lambda function body}, and a synchronization option. The STRADS-AP runtime partitions input data into L mini-batches where L is approximately $(input\_data\_size)/M$ and then schedules multiple workers to process different batches concurrently. A worker starts the lambda function $F(minibatch_i)$ with a local copy of captured variables and allows read/write only to the local copy while running F. For simplicity's sake, we let F_i denote $F(minibatch_i)$. At the end of a minibatch, a separate per-worker thread synchronizes the local copy of only those DDSs captured by reference according to the sync option. For demonstration purposes, consider the core of word2vec based on Google's open source[5]:

```
typedef vector<word> T1; typedef vector<array<float, vec_size>> T2;
dvector<T1> &inputD=CreaetVector<T1>(path,parser);// DDS for input
dvector<T2> &Syn0=CreateVector<T2>(vocsize, initrow1);// DDS
dvector<T2> &Syn1=CreateVector<T2>(vocsize, initrow1);// DDS
float alpha(.01f); binary_tree &bintree=CreateBinaryTree();
expTable &eb=MakeExpTable();//precompute the exp() table
for(auto i(0);i<maxiter;i++){
 Sync_Parallel_For(inputD,minibatchsize,
      [alpha, eb, bintree,&Syn0,&Syn1](const vector<T1> &m){
           for(auto &sentence: m){
             //for each window in setence, pick up N words
               // select N negative sample words based on bintree
               // r/w to N rows of Syn0 and Syn1 vector tables
           }
      },FLAGS_SyncOption); // specifies synchronization scheme
  } // end of for(auto i(0);i<maxiter..)
```

STRADS-AP allows programmers to select a synchronization scheme: BSP[27], SSP[6, 12, 18], or HybridSync[1]. This allows programmers to focus on writing ML computations without implementing network communication or parameter synchronization.

3 IMPLEMENTATION

The STRADS-AP runtime consists of a master driver process, multiple worker processes, and DDS server processes. An application is written as a driver program which runs on the master only. In a driver program, programmers create DDSs using the create operator, and the DDSs are automatically partitioned over DDS servers. DDS servers provide a global address abstraction to master and workers. On invocation of a parallel operator, the master schedules the workload of the operator and transparently makes RPC calls to workers passing serialized arguments[2].

Parallel-For operators extend the virtual iteration(VI) technique[7] previously used for prefetching. STRADS-AP uses VI for analyzing data dependency among lambda function instances, F_i with $i = S, S+1, .., S+N$, for Async_Parallel_For as well as prefetching for Sync_Parallel_For. To achieve an efficient implementation of VI, STRADS-AP assumes that targeted ML applications satisfy three properties: 1) parallel-for operators are repeated many times before convergence; 2) accessed memory addresses of F_i do not change over different iterations; and 3) any serial reordering of F_i executions is acceptable. All three are routinely acceptable in ML.

The STRADS-AP runtime keeps track of the invocation count of each parallel-for operator. On the first invocation of a parallel-for operator, STRADS-AP runs the operator without committing writes and gathers read/write addresses on data structures of DDS type for each F_i. Sync_Parallel_For uses this read/write address information for prefetching. For Async_Parallel_For operator, STRADS-AP builds a dependency graph based on this address information and makes parallel execution plans that do not cause conflicts. To find enough parallelism, STRADS-AP might change execution ordering of F_S,F_{S+1}...,F_{S+N}. Thus, the output of Async_Parallel_For might be different from that of sequential execution, but STRADS-AP ensures serializability[3] of loop bodies. For each Async_Parallel_For in a driver program, STRADS-AP runs VI and scheduling once and reuses execution plans over subsequent iterations. Thus, the overhead of VI and scheduling is amortized over multiple iterations.

4 PERFORMANCE & PRODUCTIVITY

In our performance evaluations shown in Table 1 and 2, we used up to 16 machines where each machine has 16 cores. First, we compared STRADS-AP SGDMF and word2vec with single machine baseline codes and MPI-based distributed codes. For a fair comparison, we implemented the same algorithm [3] as did our competitors. In Table 1, compared with single machine baselines, STRADS-AP 256 core codes converged 38.7 and 51 times faster for SGDMF and

SGDMF	cores	Netflix[2]
MPI	256	254s
STRADS-AP	256	325s
SingleThread	1	12600s
OpenMP	16	1498s

Word2Vec	cores	1Billon[4]
MPI	256	1141s
STRADS-AP	256	1331s
SingleThread	1	67754s
OpenMP	16	11772s

Table 1: Left table reports elapsed times in seconds for 60 iterations for MF with rank=1K. Right table reports the times for 10 iterations for word2vec with 1 billion data[4] with vector size=100,window=5.

#nodes	Similarity		Analogy	
	STRADS	MPI	STRADS	MPI
SingleThrd	0.564		0.403	
128	0.565	0.561	0.411	0.411
256	0.555	0.552	0.400	0.397

Implementaton	lines
W2V SingleThrd	572
W2V STRADS-AP	591
MF SingleThrd	271
MF STRADS-AP	279

Table 2: Left table reports test accuracy for 1 billion data set. We used WordSimilarity-353 test set[8], and Google's analogy test set[21]. Right table reports line counts of W2V and MF codes.

word2vec, respectively. However, the 256 core STRADS-AP SGDMF converged 27% slower than the hand optimized MPI version. We attributed this difference to the overhead of VI+Scheduling (17%) and automated data serialization (10%). STRADS-AP word2vec converged 16 percent slower than hand optimized MPI code on 256 cores. The memory copy optimization available in MPI and the data serialization overhead of STRADS-AP account for this performance gap. Second, we compared the test accuracy of word2vec. The left side in Table 2 shows that STRADS-AP word2vec on 128, 256 core configurations achieved comparable accuracy with the ideal accuracy of SingleThread. STRADS-AP and SingleThread SGDMF both converged to a similar objective value after running 60 iterations, which means STRADS-AP SGDMF achieved the ideal statistical efficiency of SingleThread thanks to STRADS-AP scheduling.

To explore programming productivity with STRADS-AP , we ran two experiments. First, we compared line counts of whole application programs including I/O and computation routines. The right side in Table 2 shows that the line counts of STRADS-AP applications are comparable to those of SingleThread baselines. Second, we conducted a preliminary user study. We ran a capstone project with a master student in CMU's Master of Computational Data Science program[26]. The student had C++ programming experince and had finished an introductory graduate ML course. The student implemented distributed implementations of word2vec with STRADS-AP and MPI[9] based on Mikolov's papers[20, 21] and Google open source[5]. It took 2 hours to implement the STRADS-AP code starting with Google's open source code. It took 2 days to achieve a working implementation of MPI-based code, but the initial MPI-based code suffered low test accuracy and computation throughput. The student was able to match the STRADS-AP code in terms of test accuracy and throughput after two weeks of performance optimizations. We believe this is a complelling example of the potential value of STRADS-AP for ML-savvy developers interested more in statistical code than in distributed system code.

Recently, high-level frameworks, such as TensorFlow[1] and PyTorch[23], have significantly simplified deep learning programming using symbolic differentiation and automatic gradient computation and update. Compared with frameworks like these, STRADS-AP offers a lower level programming abstraction and requires ML developers to write the optimization code. However, STRADS-AP enables relatively simple programming for ML developers who want to develop non-gradient algorithms, to explore ML models with non-differentiable loss functions, and to parallelize dependent computations without relaxing serializability; that is, STRADS-AP enables strong support for ML programs like these that cannot be efficiently supported by such high-level frameworks.

[1]This scheme runs lock-free asynchronous execution[24] among multi-threads within a machine, but enforces synchronization across a cluster at the end of a minibatch.
[2]C++ does not have strong reflection capability like Java/Python. To fill this void, STRADS-AP implements a standalone tool that analyzes user source code using LLVM [16] Clang ASTMatcher, identifies all type information of parallel operators' arguments, and generates RPC stub codes and a named function class for each lambda function argument of a parallel operator. This code synthesizing tool saves ML programmers significant time and effort needed for writing application-specific RPC codes.
[3]For word2vec, we conducted evaluations with Skip-gram with Hierarchial Softmax in [5] and did not include CBOW routines in line counts for both STRADS-AP and SingleThread. For word2vec SingleThread/OpenMP, we modified google word2vec to load all input data into main memory so that no disk access occurs in training time.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] James Bennett, Stan Lanning, and Netflix Netflix. 2007. The Netflix Prize. In *In KDD Cup and Workshop in conjunction with KDD*.

[3] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1986. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. 2013. *One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling*. Technical Report. Google. http://arxiv.org/abs/1312.3005

[5] Google Code. 2013. word2vec. (2013). https://code.google.com/archive/p/word2vec/

[6] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2014. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 37–48. https://www.usenix.org/conference/atc14/technical-sessions/presentation/cui

[7] Henggang Cui, Alexey Tumanov, Jinliang Wei, Lianghong Xu, Wei Dai, Jesse Haber-Kucharsky, Qirong Ho, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2014. Exploiting Iterative-ness for Parallel ML Computations. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 5, 14 pages. https://doi.org/10.1145/2670979.2670984

[8] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. 2001. Placing search in context: The concept revisited. In *Proceedings of the 10th international conference on World Wide Web*. ACM, 406–414.

[9] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard*. Technical Report. Knoxville, TN, USA.

[10] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *OSDI*, Vol. 12. 2.

[11] Saurabh Gupta and Vineet Khare. 2017. BlazingText: Scaling and Accelerating Word2Vec Using Multiple GPUs. In *Proceedings of the Machine Learning on HPC Environments (MLHPC'17)*. ACM, New York, NY, USA, Article 6, 5 pages. https://doi.org/10.1145/3146347.3146354

[12] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS*.

[13] Shihao Ji, Nadathur Satish, Sheng Li, and Pradeep Dubey. 2016. Parallelizing Word2Vec in Shared and Distributed Memory. *CoRR* abs/1604.04661 (2016).

[14] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. 2016. STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2901318.2901331

[15] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 8 (2009), 30–37.

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[17] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth Gibson, and Eric P. Xing. 2014. On Model Parallelism and Scheduling Strategies for Distributed Machine Learning. In *NIPS*.

[18] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*.

[19] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (2012), 716–727. https://doi.org/10.14778/2212351.2212354

[20] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). arXiv:1301.3781 http://arxiv.org/abs/1301.3781

[21] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'13)*. Curran Associates Inc., USA, 3111–3119. http://dl.acm.org/citation.cfm?id=2999792.2999959

[22] Xinghao Pan, Maximilian Lam, Stephen Tu, Dimitris Papailiopoulos, Ce Zhang, Michael I Jordan, Kannan Ramchandran, and Christopher Ré. 2016. Cyclades: Conflict-free Asynchronous Machine Learning. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 2568–2576. http://papers.nips.cc/paper/6604-cyclades-conflict-free-asynchronous-machine-learning.pdf

[23] PyTorch. [n. d.]. PyTorch. ([n. d.]). PyTorch.org/

[24] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*.

[25] R. Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* 58, 1 (1996), 267–288.

[26] Carnegine Mellon University. 2016. Master of Computational Data Science. (2016). https://mcds.cs.cmu.edu/learn-us-curriculum

[27] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/79173.79181