HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent

Feng Niu, Benjamin Recht, Christopher Ré and Stephen J. Wright
Computer Sciences Department, University of Wisconsin-Madison
1210 W Dayton St, Madison, WI 53706

June 2011

Motivation

- 1 lock impedes parallel
- 2 take advantage of multicore-system

Sparsity

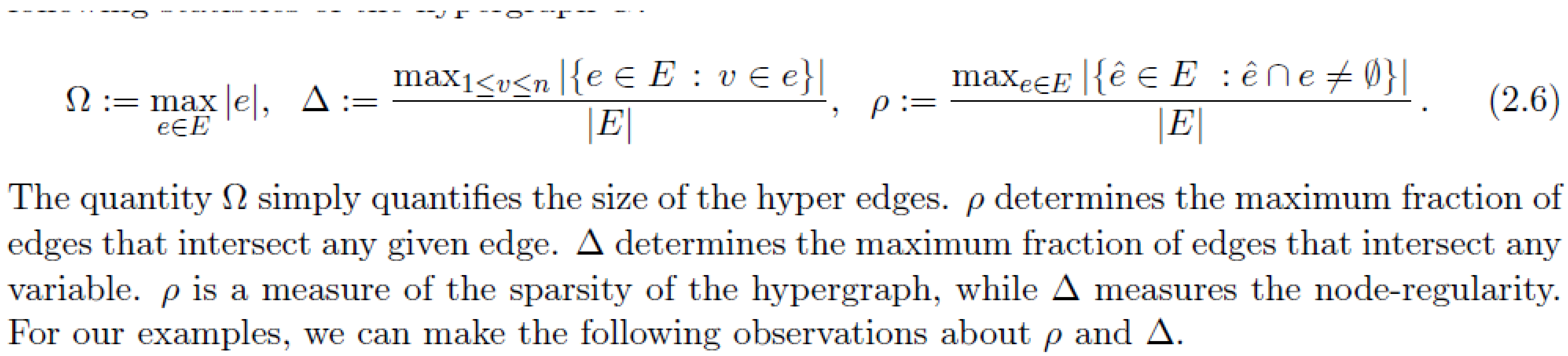
- 1 sparse data access

- 2 sparse data rewrite

Problem

- 1 Sparse SVM
- 2 Matrix Completion
- 3 Graph Cuts

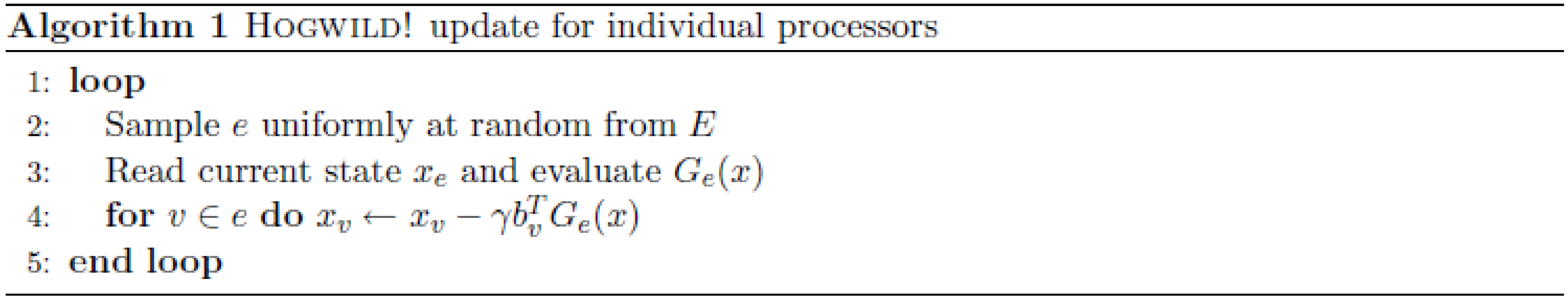
Sparsity Criterion



$$\Omega := \max_{e \in E} |e|, \quad \Delta := \frac{\max_{1 \le v \le n} |\{e \in E \, : \, v \in e\}|}{|E|}, \quad \rho := \frac{\max_{e \in E} |\{\hat{e} \in E \ : \hat{e} \cap e \neq \emptyset\}|}{|E|}. \tag{2.6}$$

The quantity Ω simply quantifies the size of the hyper edges. ρ determines the maximum fraction of edges that intersect any given edge. Δ determines the maximum fraction of edges that intersect any variable. ρ is a measure of the sparsity of the hypergraph, while Δ measures the node-regularity. For our examples, we can make the following observations about ρ and Δ.

HOGWILD!



Algorithm 1 HOGWILD! update for individual processors

```
1: loop
2:     Sample $e$ uniformly at random from $E$
3:     Read current state $x_e$ and evaluate $G_e(x)$
4:     for $v \in e$ do $x_v \leftarrow x_v - \gamma b_v^T G_e(x)$
5: end loop
```

HOGWILD!

- 1 every processor only modify the variables related itself
- 2 every processor have no idea about other processor behavior
- 3 running in parallel w/o locks -> linear speedup

HOGWILD! Theoretical Analysis

- 1 “without-replacement” vs “replacement”
- 2 constant step size ->$K \geq O(\frac{\log\left(\frac{1}{\varepsilon}\right)}{\varepsilon})$
- 3 back-off scheme -> $K \geq O(\frac{1}{\varepsilon})$
- 4 robust to Hyper Parameter

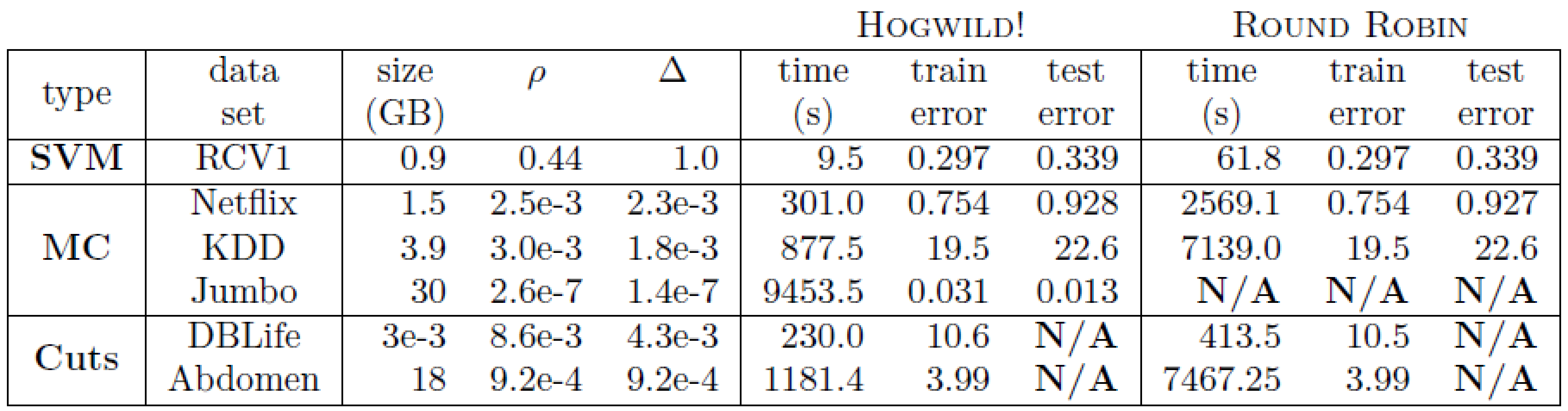
Round Robin

- 1 processors are ordered
- 2 update operations are ordered
- 3 the time required to lock memory for writing is dwarfed by the gradient computation time (gradient computation is slow!)

AIG

- HOGWILD！ with lock

Experiments



					HOGWILD!			ROUND ROBIN		
type	data set	size (GB)	ρ	Δ	time (s)	train error	test error	time (s)	train error	test error
SVM	RCV1	0.9	0.44	1.0	9.5	0.297	0.339	61.8	0.297	0.339
MC	Netflix	1.5	2.5e-3	2.3e-3	301.0	0.754	0.928	2569.1	0.754	0.927
	KDD	3.9	3.0e-3	1.8e-3	877.5	19.5	22.6	7139.0	19.5	22.6
	Jumbo	30	2.6e-7	1.4e-7	9453.5	0.031	0.013	**N/A**	**N/A**	**N/A**
Cuts	DBLife	3e-3	8.6e-3	4.3e-3	230.0	10.6	**N/A**	413.5	10.5	**N/A**
	Abdomen	18	9.2e-4	9.2e-4	1181.4	3.99	**N/A**	7467.25	3.99	**N/A**

Figure 2: Comparison of wall clock time across of HOGWILD! and RR. Each algorithm is run for 20 epochs and parallelized over 10 cores.

Experiments

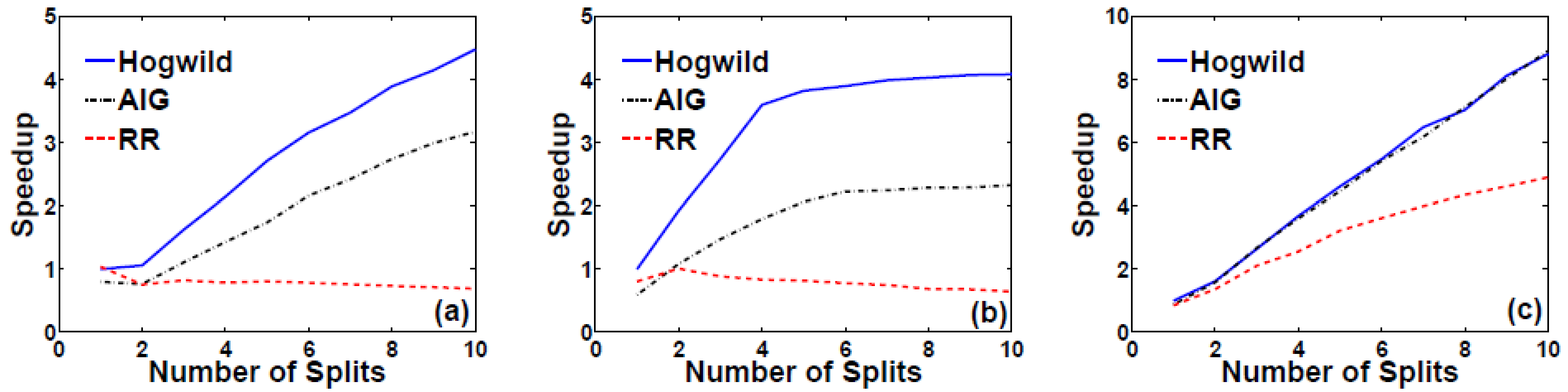


Figure 3: Total CPU time versus number of threads for (a) RCV1, (b) Abdomen, and (c) DBLife.

Experiments

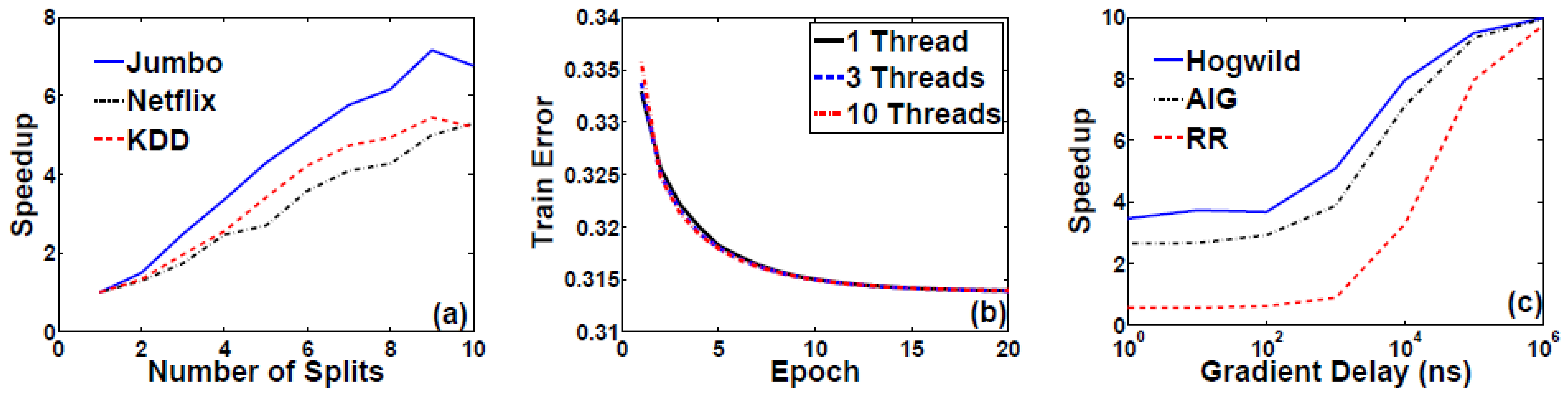


Figure 5: (a) Speedup for the three matrix completion problems with HOGWILD!. In all three cases, massive speedup is achieved via parallelism. (b) The training error at the end of each epoch of SVM training on RCV1 for the averaging algorithm [30]. (c) Speedup achieved over serial method for various levels of delays (measured in nanoseconds).

Conclusion

- 1 takes advantage of sparsity in machine learning problems
- 2 parallel speedup when the gradients are computationally intensive
- 3 can be generalized to problems where variables occur frequently

Towards Self-Tuning Parameter Servers

ABSTRACT

Recent years, many applications have been driven advances by the use of Machine Learning (ML). Nowadays, it is common to see industrial-strength machine learning jobs that involve millions of model parameters, terabytes of training data, and weeks of training. Good efficiency, i.e., fast completion time of running a specific ML job, therefore, is a key feature of a successful ML system. While the completion time of a long-running ML job is determined by the time required to reach model convergence, practically that is also largely influenced by the values of various system settings. In this paper, we contribute techniques towards building *self-tuning parameter servers*. Parameter Server (PS) is a popular system architecture for large-scale machine learning systems; and

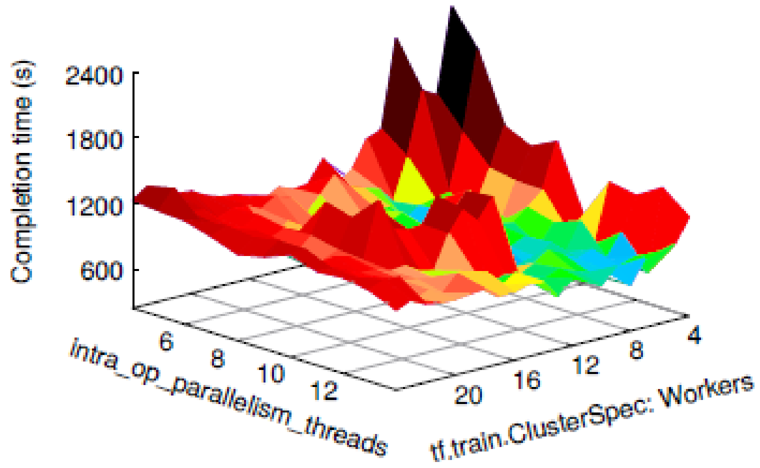


Figure 1: A 2D response surface of a TensorFlow job

Pipeline

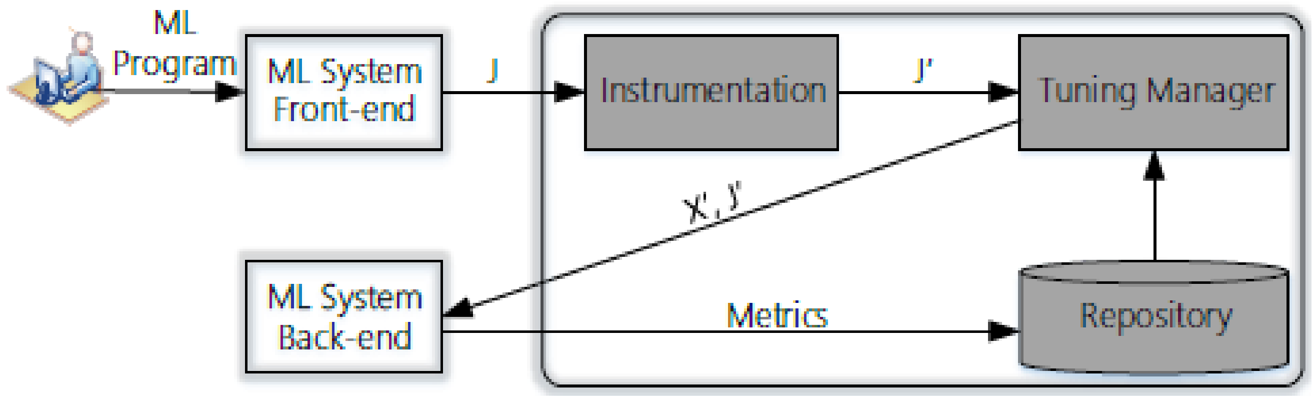


Figure 6: Online Optimization Framework

Parameter Server

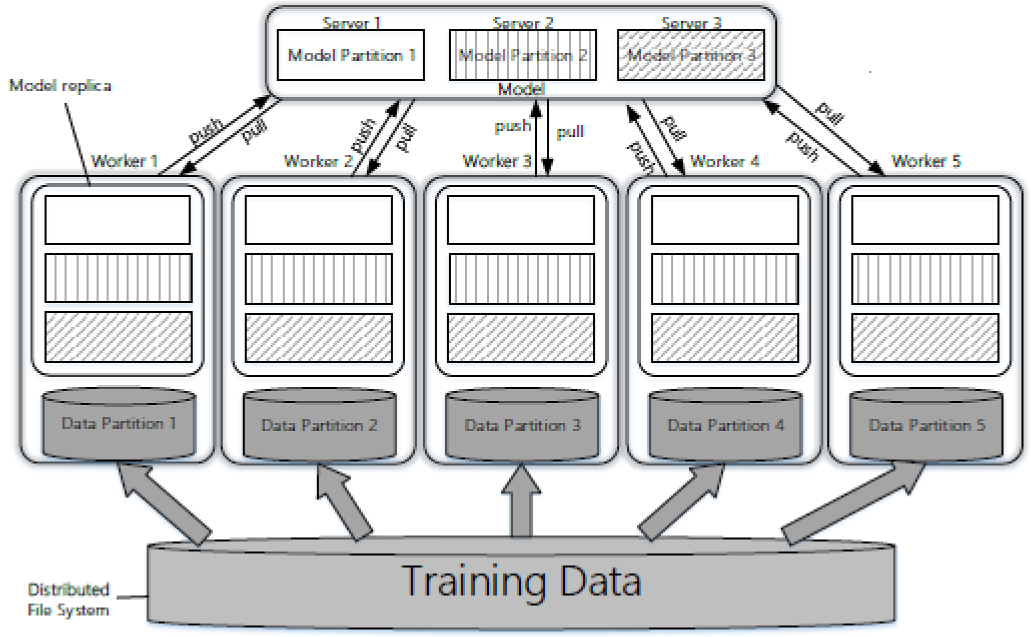


Figure 3: Parameter server architecture example: servers collectively maintain the full model and each worker gets a copy of the full model

Parameter Server

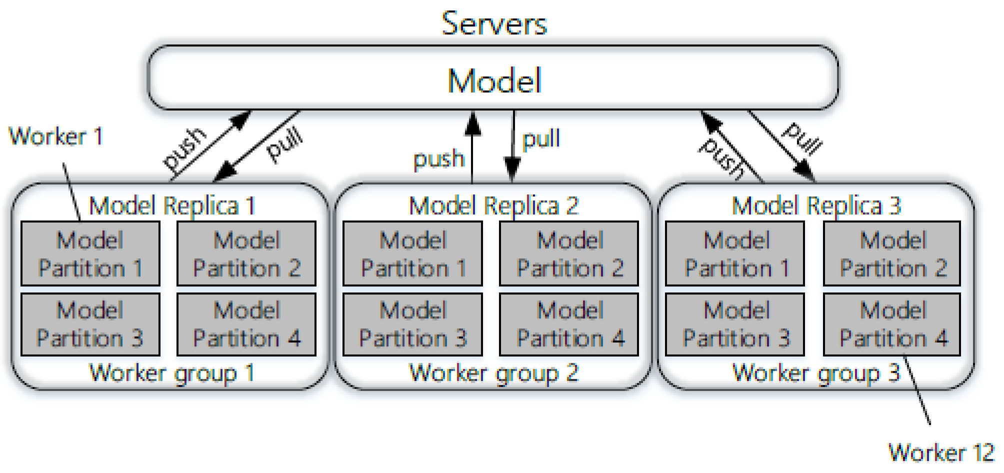
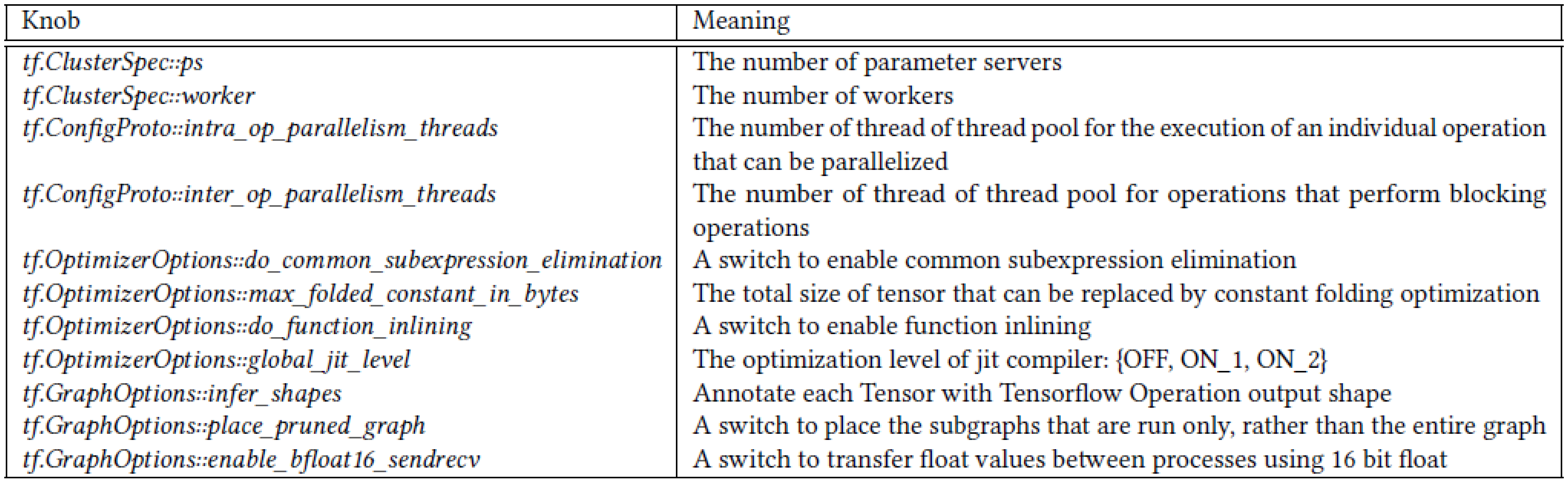


Figure 4: Parameter server architecture example: workers form groups and each worker group collectively maintains one copy of the full model

Tuning Parameter



Knob	Meaning
tf.ClusterSpec::ps	The number of parameter servers
tf.ClusterSpec::worker	The number of workers
tf.ConfigProto::intra_op_parallelism_threads	The number of thread of thread pool for the execution of an individual operation that can be parallelized
tf.ConfigProto::inter_op_parallelism_threads	The number of thread of thread pool for operations that perform blocking operations
tf.OptimizerOptions::do_common_subexpression_elimination	A switch to enable common subexpression elimination
tf.OptimizerOptions::max_folded_constant_in_bytes	The total size of tensor that can be replaced by constant folding optimization
tf.OptimizerOptions::do_function_inlining	A switch to enable function inlining
tf.OptimizerOptions::global_jit_level	The optimization level of jit compiler: {OFF, ON_1, ON_2}
tf.GraphOptions::infer_shapes	Annotate each Tensor with Tensorflow Operation output shape
tf.GraphOptions::place_pruned_graph	A switch to place the subgraphs that are run only, rather than the entire graph
tf.GraphOptions::enable_bfloat16_sendrecv	A switch to transfer float values between processes using 16 bit float

Table 1: System knobs tuned in TensorFlowOnline

Parallelism in Distributed Learning

- 1 Data Parallelism, range/hash partitioning
- 2 Model Parallelism, local keeps a full/part copy of model
- 3 Hardware Parallelism, CPU/GPU. Thread affinity

Initialization Phase

- 1 bring in a small set of representative settings
- 2 iteration, loss, execution time

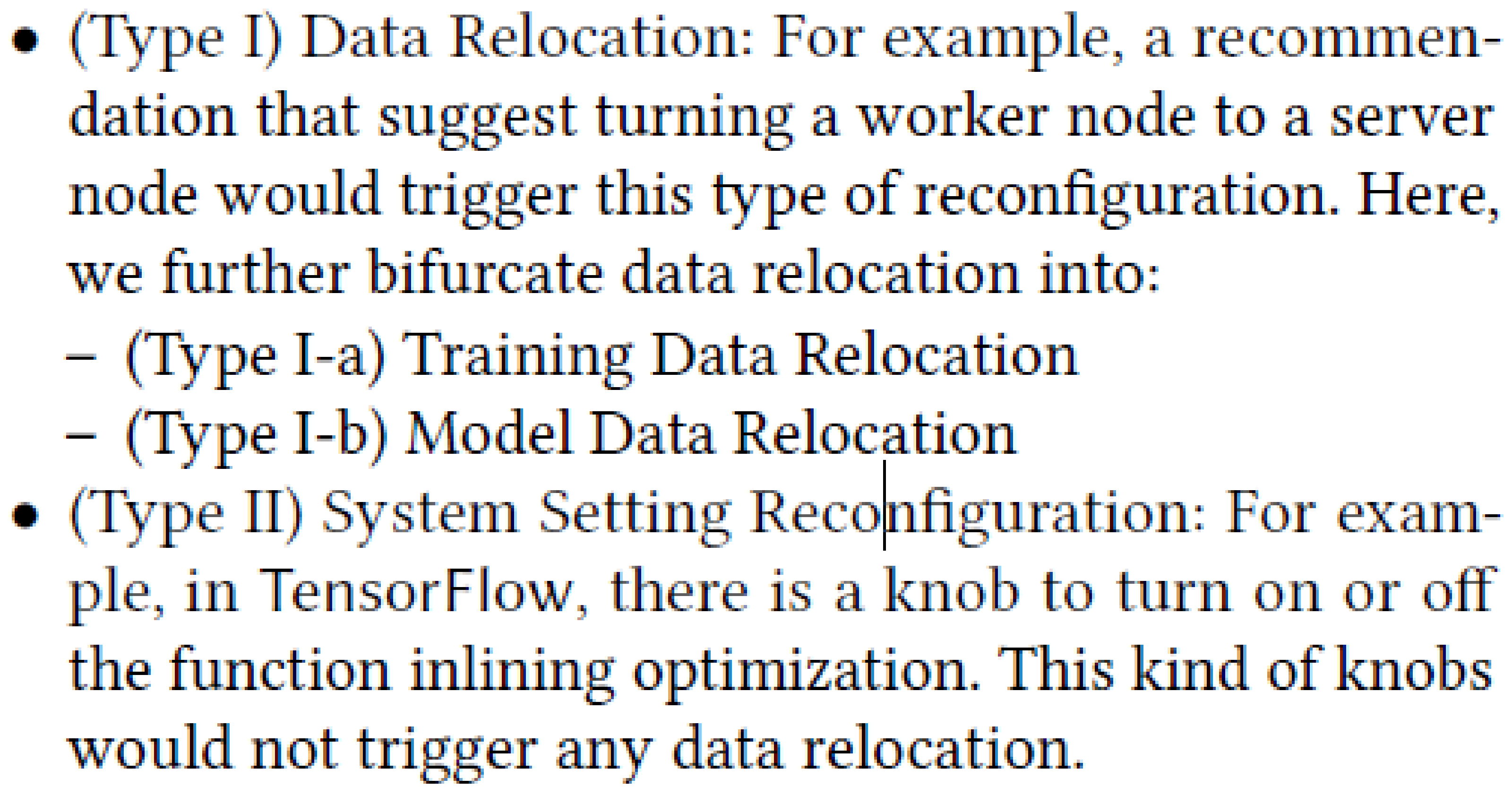
From Bounds to Legitimate Estimation

$$r_i^j = \frac{\mathcal{H}_i}{\epsilon} \log \frac{d_i}{\epsilon} \tag{4}$$

gradient based algorithms under a *synchronous, serial,* and *offline* setting. For example, they deduce the total number of iterations as $z = \mathcal{H}/\epsilon$, with only one hidden constant.

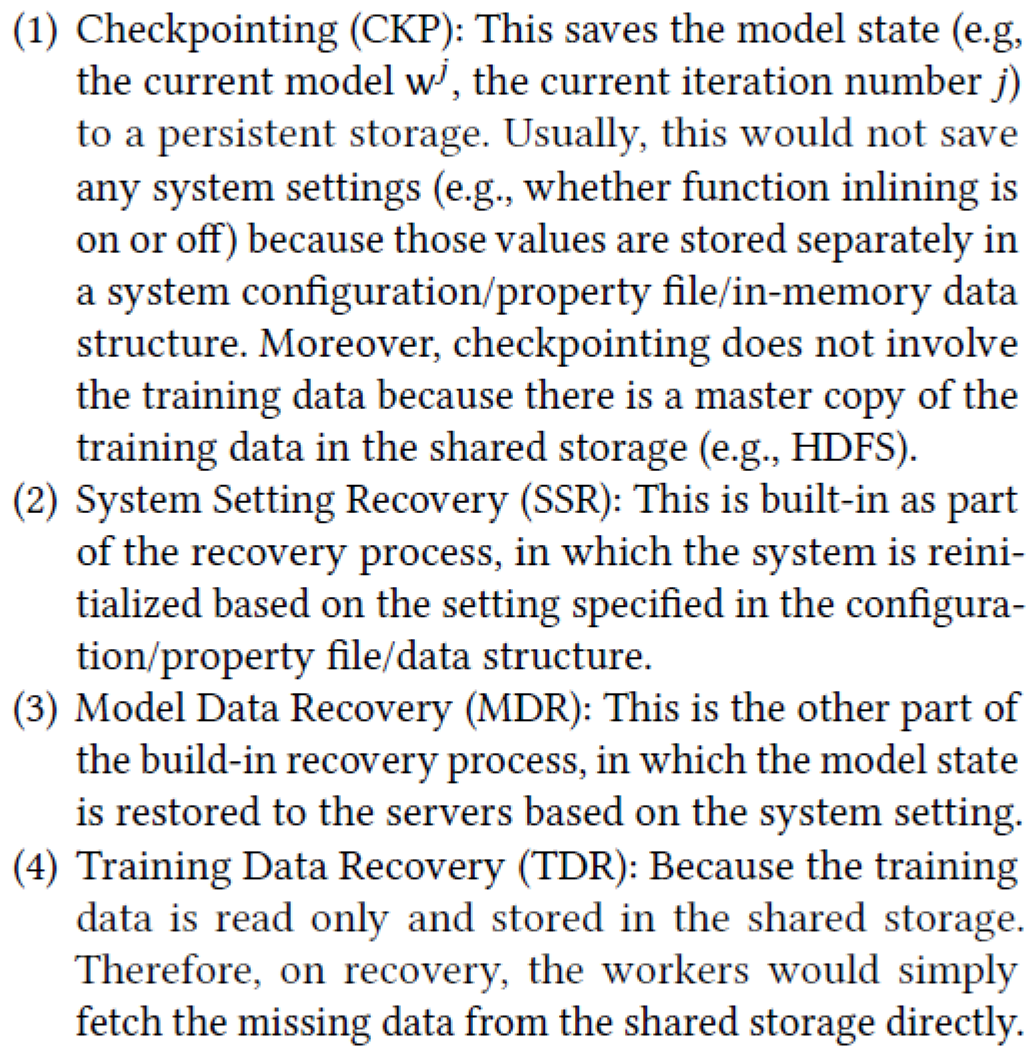
$$\min\{2l_i^{j_0}, \max\{l_i^{j_0+1}, l_i^{j_0+2}, \cdots, l_i^{j_0+a}\}\} \tag{5}$$

Online Reconfiguration



- (Type I) Data Relocation: For example, a recommendation that suggest turning a worker node to a server node would trigger this type of reconfiguration. Here, we further bifurcate data relocation into:
  - (Type I-a) Training Data Relocation
  - (Type I-b) Model Data Relocation
- (Type II) System Setting Reconfiguration: For example, in TensorFlow, there is a knob to turn on or off the function inlining optimization. This kind of knobs would not trigger any data relocation.

Online Reconfiguration



(1) Checkpointing (CKP): This saves the model state (e.g, the current model w^j, the current iteration number j) to a persistent storage. Usually, this would not save any system settings (e.g., whether function inlining is on or off) because those values are stored separately in a system configuration/property file/in-memory data structure. Moreover, checkpointing does not involve the training data because there is a master copy of the training data in the shared storage (e.g., HDFS).

(2) System Setting Recovery (SSR): This is built-in as part of the recovery process, in which the system is reinitialized based on the setting specified in the configuration/property file/data structure.

(3) Model Data Recovery (MDR): This is the other part of the build-in recovery process, in which the model state is restored to the servers based on the system setting.

(4) Training Data Recovery (TDR): Because the training data is read only and stored in the shared storage. Therefore, on recovery, the workers would simply fetch the missing data from the shared storage directly.

On-Demand-Model-Relocation (ODMR)

- 1 Originally, send model parameters/gradient to old destination
- 2 Currently, send it directly to assigned destination

- Fewer user-defined hyper-parameters
- More efficient training speed
- Online efficient reconfiguration
- Online progress estimation
- Training dependent optimization

- In DL training, iteration/batch size has been set
- Calculating the left iterations by loss is not applicable for some problems, e.g. GAN
- Compared with the experiment that offline greedy select.
- Not applicable for dynamic model (time per iteration is not stable)
- In multi-user share server, pre-allocated resource could lead to a waste and dynamic-allocated resource may lead to contradiction