Parallel and Distributed Optimization Algorithm

Part II

Ruoyu Li[1]

[1]Department of Computer Science and Engineering
University of Texas at Arlington

March 26, 2015

Outline

1. Introduction
2. The HogWild! Algorithm
3. Convergence Rate Analysis
4. Experiments

Paper

Authors: Recht, Benjamin, et al.
Title: "Hogwild: A lock-free approach to parallelizing stochastic gradient descent."
Conference: Advances in Neural Information Processing Systems. 2011.

Motivations

- Stochastic gradient descent algorithm is feasible to data-intensive projects; small memory footprint, robust to noise, rapid learning rate.
- Previous parallel stochastic gradient descent methods all requires memory-locking. randomly selected coordinates to update at each iteration. It is difficult to parallel it because it all need synchronizations before update the gradients.
- Focus on multi-core workstation to handle terabytes size data. low latency, high throughout speed of shared memory, high bandwidth for disk read and write.

Contributions

- HOGWILD! Algorithm for parallel SGD without memory locking.
- Prove that the possible overwriting risk without locking memory rarely occurs when the data access is sparse. and it cannot impact much on optimization results.
- We prove the rate convergence of the proposed parallel algorithm. robust $1/k$ converge rate when dynamic step-size is implemented by backoff in the initial constant step-size.

Sparse Separable Cost function

Our goal is to minimize the cost function as below:

$$f(x) = \sum_{e \in E} f_e(x_e) \tag{1}$$

Observation on sparsity

$f(x)$ is completely separable and compared with n, the number of coordinates, and $|E|$, the number of samples, the x_e is only a small portion of the set of coordinates, and $f_e(x)$ only work on small group of coordinates.

Sparse SVM, suppose our goal is to fit a support vector machine to some labeled data $E = \{(z_1, y_1), \cdots, (z_{|E|}, y_{|E|})\}$.

$$\min_x \sum_{\alpha \in E} \max(1 - y_\alpha x^T z_\alpha, 0) + \lambda \|x\|_2^2 \tag{2}$$

If we have a prior that z is very sparse, so it implies that resulted x is also sparse. The elements of x associated with non-zero elements of z is possible to be nonzero. So we have revised separable cost function:

$$\min_x \sum_{\alpha \in E} \Big(\max(1 - y_\alpha x^T z_\alpha, 0) + \lambda \sum_{u \in e_\alpha} \frac{x_u^2}{d_u} \Big) \tag{3}$$

d_u denotes the number of training examples which are non-zero in component u (u $=$ 1,2,$\cdots$,n)

A popular heuristic recovers the estimate of Z as a product LR^* of factors obtained from the following:

$$\min_{L,R} \sum_{(u,v)\in E} (L_u R_v^* - Z_{uv})^2 + \frac{\mu}{2}\|L\|_F^2 + \frac{\mu}{2}\|R\|_F^2 \tag{4}$$

To put this problem into separable sparse form, we write it as:

$$\min_{L,R} \sum_{(u,v)\in E} \{(L_u R_v^* - Z_{uv})^2 + \frac{\mu}{2|E_{u-}|}\|L_u\|_F^2 + \frac{\mu}{2|E_{-v}|}\|R_v\|_F^2\} \tag{5}$$

In the graph cut problem, we are given a sparse, nonnegative matrix W which indexes similarity between entities. Our goal is to find a partition of the index set $\{1, \cdots, n\}$ that best conforms to this similarity matrix.

$$\min_x \sum_{(u,v)\in E} w_{u,v}\|x_u - x_v\|_1, \quad s.t. \quad x_v \in S_D \quad for \quad v = 1, 2, \cdots, n, \tag{6}$$

where $S_D = \{\zeta \in \mathcal{R}^D : \zeta_v \leq 0, \sum_{v=1}^{D} \zeta_v = 1\}$.

Statistics

We formalize this notation by defining following statistics:

$$\Omega = \max_{e\in E} |e| \tag{7}$$

$$\nabla = \frac{\max_{1\leq v\leq n} |\{e \in E : v \in e\}|}{|E|} \tag{8}$$

$$\rho = \frac{\max_{e\in E} |\{\hat{e} \in E : \hat{e}\cap \neq 0\}|}{|E|} \tag{9}$$

Assumptions

- We assume that the shared memory is accessible to p processors.
- The decision variable x stored in the memory is accessible to all processors.
- We assume that the update is atomic:

$$x_v \leftarrow x_v + a \tag{10}$$

which means it does not need a locking structure when implementing it on practical system, since each time it updates one component.

Algorithm

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**

Figure: HOGWILD! Algorithm

Description of HogWild!

- Each processor follows the algorithm,
- b_v is the basis element in $\mathcal{R}^n$, with v ranging from 1 to n. b_v is 1 on the vth component and 0 otherwise.
- $G_e(x) \in \mathcal{R}^n$ denotes a gradient or subgradient of function f_e multiplied by $|E|$.

$$|E|^{-1}[G_e(x)] \in \partial f_e(x). \tag{11}$$

  Here, $G_e(x)$ is equal to zero to components in $\neg e$.
- Since e is uniformly sampled from E, we have :

$$\mathbf{E}[G_e(x_e)] \in \partial f(x) \tag{12}$$

- In Algorithm 1, each processor samples an term e from E, computes the gradient of f_e at x_e, and then write

$$x_v \leftarrow x_v - \gamma b_v^T G_e(x) \tag{13}$$

Update of x_j

- x_j means the state of decision variable x after jth updates.
- The gradient used for updating x_j may be calculated based on $x_{k(j)}$ denotes the value of decision variables read many clock cycles earlier than j. So x_j is generally updated with a stale gradient.
- in this case, we assume each update, the step-size used is a constant, which selection of value affects a lot on the convergence analysis.
- The proposed algorithm has the same convergence rate as the serial SGD under certain conditions. Since we implement it in parallel manner, the whole speed is linearly accelerated in term of the number of processors(core/thread).

With replacement Procedure

Each processor samples e from the E each iteration randomly and compute the subgradient of f_e based on current value of x_e. Then it chooses an v uniformly at random and updates

$$x_v \leftarrow x_v - \gamma|e|b_v^T G_e(x) \tag{14}$$

This update could be also written as

$$x \leftarrow x - \gamma|e|\mathcal{P}_v^T G_e(x) \tag{15}$$

where $\mathcal{P}$ is the Euclidean projection to coordinate v which is $b_v b_v^T$.

The problem of With replacement Procedure

- This with replacement scheme assumes that a gradient $G_e(x)$ is computed and the only one of its components is used to update the decision variable.
- Such a scheme is computationally wasteful due to that the rest of gradient components also carry information for decreasing the cost.

We, here, use the so-called without replacement to all processors.

- This processor computes the gradient $G_e(x)$;
- And we update all component in selected e in their respective queues.

Some property to claim

Before we prove the convergence rate of the parallel stochastic gradient descent algorithm, we need to claim some properties of the cost function and some quantities and bounds.

- We assume each component of the separable function f, f_e is convex function.
- Assume that f is twice differentiable with Lipschits constant L:

$$\|\delta f(x') - \delta f(x)\| \leq L\|x' - x\|, \forall x', x \in X \tag{16}$$

- We also assume that the function f is strong convex, so it satisfied:

$$f(x') \geq f(x) + (x' - x)^T \delta f(x) + \frac{c}{2}\|x' - x\|^2, \forall x', x \in X \tag{17}$$

- We also assume the norm of gradient is bounded:

$$\|G_e(x_e)\|_2 \leq M \tag{18}$$

Some property to claim- continued

Beside, we also need to claim a important inequality:

$$\gamma c < 1 \tag{19}$$

This could be easily proved by $\gamma = 1/L$ and :

$$f(x') \leq f(x) + (x' - x)^T \delta f(x) + \frac{L}{2}\|x' - x\|^2, \forall x', x \in X \tag{20}$$

, where L is the Lipschits constant of f.

The proposition 4.1- the choice of step-size and convergence rate

we first give the conclusion and then analyze it.

Proposition 4.1 *Suppose in Algorithm 1 that the lag between when a gradient is computed and when it is used in step j — namely, $j - k(j)$ — is always less than or equal to τ, and γ is defined to be*

$$\gamma = \frac{\vartheta \epsilon c}{2LM^2\Omega\left(1 + 6\rho\tau + 4\tau^2\Omega\Delta^{1/2}\right)}. \tag{4.5}$$

for some $\epsilon > 0$ and $\vartheta \in (0,1)$. Define $D_0 := \|x_0 - x_\star\|^2$ and let k be an integer satisfying

$$k \geq \frac{2LM^2\Omega\left(1 + 6\tau\rho + 6\tau^2\Omega\Delta^{1/2}\right)\log(LD_0/\epsilon)}{c^2\vartheta\epsilon}. \tag{4.6}$$

Then after k component updates of x, we have $\mathbb{E}[f(x_k) - f_\star] \leq \epsilon$.

Some Observations

- if the bound on the lag $j - k(j)$ is chosen to be zero, which means that the gradient computation and update must be synchronized, then the convergence rate becomes exactly the same as seral SGD.
- a similar rate could be obtained, if we choose $\tau = o(n^{1/4})$ as ρ and δ are typically both $o(1/n)$.
- note that the robustness means that the rate is more consistent to noise. If we have a error-deblurred c_r for instead of c, $c_r < c$. The proof and conclusion still holds.
- Up to the log term, we have displayed the $1/k$ convergence rate of proposed SGD.
- We can eliminate the log term by decreasing the step-size after a large number of iterations.

Eliminate the log term

Two-phases

- We first run algorithm for a fixed step-size *gamma* $< 1/c$ for many updates, K.
- Wait for all processor to coalesce(the only synchronization), and reduce the *gamma* by a fixed factor *beta* $\in (0, 1)$ and run for $\beta^{-1}K$ updates.
- the benefit of the scheme is that the step-size we use is always less than $1/c$, helps to avoid possible exponential slow-down caused by large step-size at the beginning.

Eliminate the log term-continued

Error term

Define the error term a_j after jth update:

$$\frac{1}{2}\mathbf{E}[\|x_j - x_*\|^2] \tag{21}$$

We can prove that it satisfied following recursion:

$$a_{k+1} \leq (1 - c_r\gamma)(a_k - a_\infty(\gamma)) + a_\infty(\gamma) \tag{22}$$

where a_∞ is some non-negative function of γ:

$$a_\infty \leq \gamma B \tag{23}$$

Eliminate the log term-continued

Based on Eq21, unwrapping it , we have the expression for error after k round of update:

$$a_k \leq (1 - c_r\gamma)^k (a_0 - a_\infty(\gamma)) + a_\infty(\gamma) \tag{24}$$

Sufficient condition for $a_k \leq \epsilon$ is that

$$a_\infty(\gamma) \leq \epsilon/2 \tag{25}$$

$$(1 - c_r\gamma)^k a_0 \leq \epsilon/2 \tag{26}$$

are both satisfied. Eq26 could be written as:

$$k \geq \frac{log(2a_0/\epsilon)}{\gamma c_r} \tag{27}$$

Eliminate the log term-continued

Backoff scheme

- We first run the algorithm with constant initial step-size and converge to the ball about x_* with radius less than $2B/c_r$. Then we will converge to the x_* by reducing the step-size.
- to find out the number of iterates that required to converge to the surface of the ball of $2B/c_r$, we define $\epsilon = 2B/c_r$ and substitute it into Eq(27). Set the initial step-size $\gamma = \frac{\theta}{c_r}$, where $\theta \in (0, 1)$, we have :

$$k \geq \theta^{-1} log(\frac{a_0 c_r}{\theta B}) \tag{28}$$

, which means as long as algorithm run with step-size $\frac{\theta}{c_r}$ for an least $\theta^{-1} log(\frac{a_0 c_r}{\theta B})$ updates, the error $\epsilon \leq 2B/c_r$

Eliminate the log term-continued

Backoff scheme

- Assume for the following epoch, the initial error $a_0 < \frac{2\theta B}{c_r}$.
- At each epoch, let us reduce the error by a factor $0 < beta < 1$.
- It takes $log_\beta(a_0/\epsilon)$ epochs to reduce the error from a_0 to ϵ.

So the total iterations need for this phase is :

$$\sum_{k=1}^{log_\beta(a_0/\epsilon)} \frac{log(2/\beta)}{\theta\beta^k} = \frac{log(2/\beta)}{\theta} \sum_{k=1}^{log_\beta(a_0/\epsilon)} \beta^{-k} \tag{29}$$

$$= \frac{log(2/\beta)}{\theta} \frac{\beta^{-1}(a_0/\epsilon) - 1}{\beta^{-1} - 1} \tag{30}$$

$$\leq \frac{a_0}{\theta\epsilon} \frac{log(2/\beta)}{1-\beta} \tag{31}$$

$$\leq \frac{2B}{c_r\epsilon} \frac{log(2/\beta)}{1-\beta} \tag{32}$$

Combining the iterations of the two phases:

$$k \geq \theta^{-1} log(\frac{a_0 c_r}{\theta B}) + \frac{2B}{c_r \epsilon} \frac{log(2/\beta)}{1-\beta} \tag{33}$$

This is sufficient to guarantee that when above inequality satisfied, the error after k updates $a_k < \epsilon$. Rearranging terms, we have the relation of error and iterates.

$$\epsilon \leq \frac{2log(2/\beta)}{1-\beta} \frac{B}{c_r} \frac{1}{k - \theta^{-1} log(\frac{a_0 c_r}{\theta B})} \tag{34}$$

Competitors

- Round-Robin algorithm. recoded and use optimized locking and signaling schemes, which lead to more delay in the update and time consumption.
- AIG. Same procedure as HOGWILD!, but it lock the variables of the selected e group of coordinates in before and after the for loop (update coordinate in e one by one).

Setups

- Code is c++, run on 6-core workstation with 24GB RAM and 72 TB/7200RPM ROM, Linux OS.
- Less than 2GB memory use for all experiments.
- $\beta = 0.9$, just show the result with largest γ that converges.

Part of Results

type	data set	size (GB)	ρ	Δ	Hogwild! time (s)	Hogwild! train error	Hogwild! test error	Round Robin time (s)	Round Robin train error	Round Robin 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: Comparison of wall clock time across of Hogwild! and RR. Each algorithm is run for 20 epochs and parallelized over 10 cores.

Part of Results

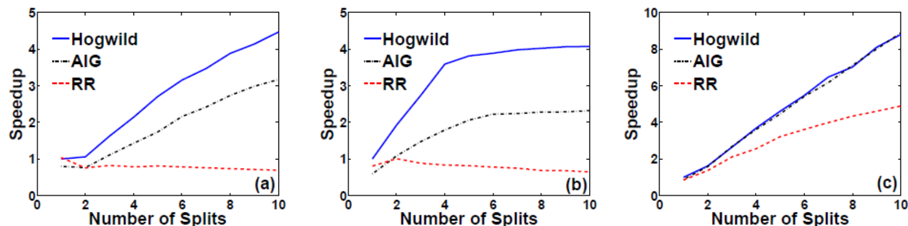

Figure: otal CPU time versus number of threads for (a) RCV1, (b) Abdomen, and (c) DBLife.

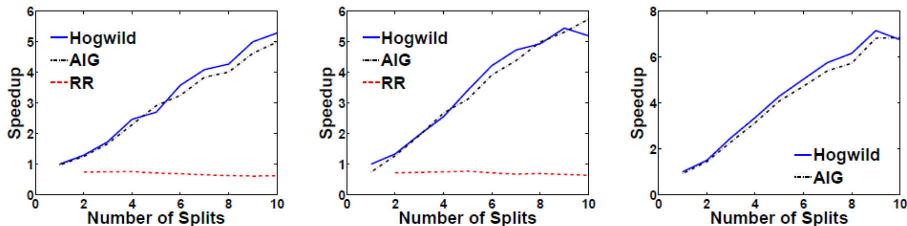

Figure: Total CPU time versus number of threads for the matrix completion problems (a)Netix Prize, (b) KDD Cup 2011, and (c) the synthetic Jumbo experiment.

Part of Results

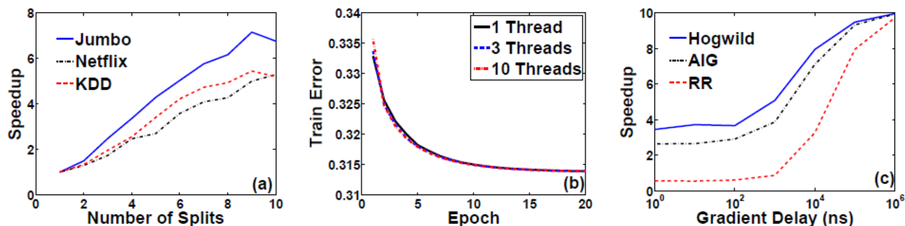

Figure: Speedup for the three matrix completion problems with Hogwild!. In all three cases, massive speedup is achieved via parallelism. (b) The training error at the end of each epoch of SVM training on RCV1 for the averaging algorithm [30]. (c) Speedup achieved over serial method for various levels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

Summary

- Our algorithm take advantages of sparsity in problem and enable a quasi-linear speedups on a variety of applications.
- Our implementation even outperformed our theoretical analysis. And for large *rho* scenario,e.g. RCV1 SVM problem, we still obtained good speedups.
- For future work, it is interesting to enumerate the structure that allows for parallel gradient computation with no collisions at all.