Parallelizing the training of sequential and linear models

present by Wei li

December 2020

Content

- Preliminaries
- Hogwild
- SAUS
- My experiments

Introduction

- motivation: Parallelizing gradient based methods to solve linear problems
- take linear regression as an example, but not limit to LR (include a big class of problem)
- talk about two algorithm, an older Hogwild, and a newer SAUS

Linear Regression Refresher

- the model:

$$\hat{y} = f(x) = w * x$$

  where $w \in R^n$ and $x \in R^n$

- squared error loss function:

$$l(x, y) = (f(x) - y)^2 = (w * x - y)^2$$

Gradient Descend Refresher

- randomly samply k instances
- calculate gradient $\nabla_w \ell(x, y)$
- the gradient can be calculated by $G_w(x, y)$:

$$\mathbb{E}\left[G_w(x, y)\right] = \nabla_w \ell(x, y)$$

$$G_w(x, y) := -2(w \cdot x - y)x$$

- apply one gradient step:

$$w_{t+1} = w_t - \lambda G_w(x, y)$$

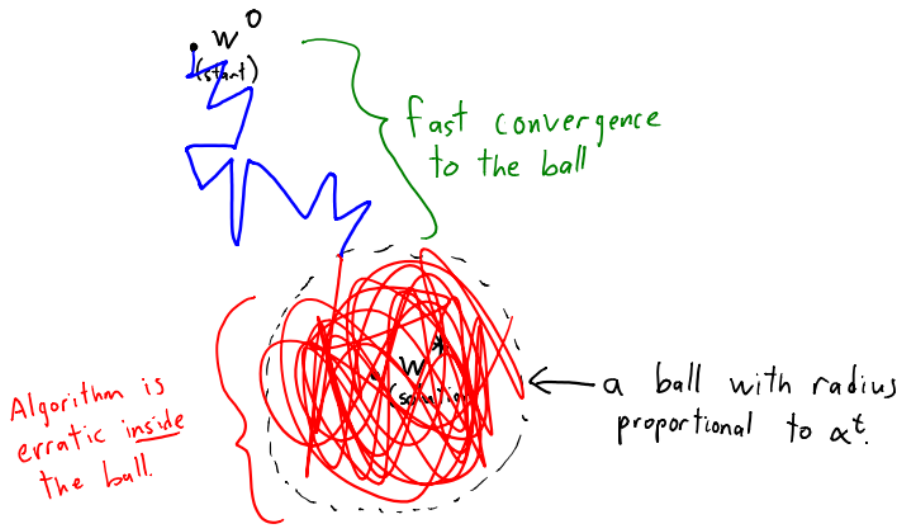
where λ is the learning rate

Gradient Descend Refresher

two issues

- 1. inherently serial: through a series of updates
- 2. too sensitive to step-size, very hard to tune

Stochastic Gradient with Constant Step Size



- the hyperparameter tuning problem gets even worse when multicores is used (divergence)

Parallel Metric

two goals

- **models' quality:** models are expected to perform as well when more cores are used – L2 distance to the real weight
- **training efficiency:** speed of training should increase at the same scale as the number of cores increase – monitor execution time for a fix number of steps

Parallelizing SGD

Setup

- shared memory model with p processors
- global weight w can be read and updated by all processors
- component-wise addition operation is atomic

$$x_v \leftarrow x_v + \alpha$$

$$\_\_atomic\_exchange(dest, \&dec\_val, \&ret\_val);$$

- updating many components at once require locking

Parallelizing SGD

a naive locking attempt

- the locking algorithm
  - Acquire a lock on current state of parameters
  - Read the parameter
  - Update the parameter with SGD step
  - Release lock
- very slow, spending most of the time acquiring locks
- we don't want locks

Hogwild[Recht et al., 2011]

HOGWILD!: Asynchronous SGD

- Hogwild idea: remove all thread lock, allow threads to overwrite each other and everything goes!

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

sparsity

- although overwrite occurs, sparsity of the gradient reduce the effect
- Sparse Separable Cost Function:

$$\min \sum_{e \in E} f_e\left(x_e\right)$$

E is the feature set, e forms a sub-vector from E
machine learning problems sub-vector x_e contains only very small number of components (is sparse compared to $|E|$)

Hogwild

Examples of sparsity

- Sparse SVM: Features are very sparse (only few non-zero components)
- Matrix Completion: Only provided entries from a sparse sampling of data (e.g., each user rates very few movies)
- Graph Cuts: Find the minimum number of cuts to separate entities (group similar items together)
- Linear Regression and Logistic Regression : control the sparsity by apply zero mask to the input data

Stochastic Atomic Update Scheme(SAUS)[Raff et al., 2018]

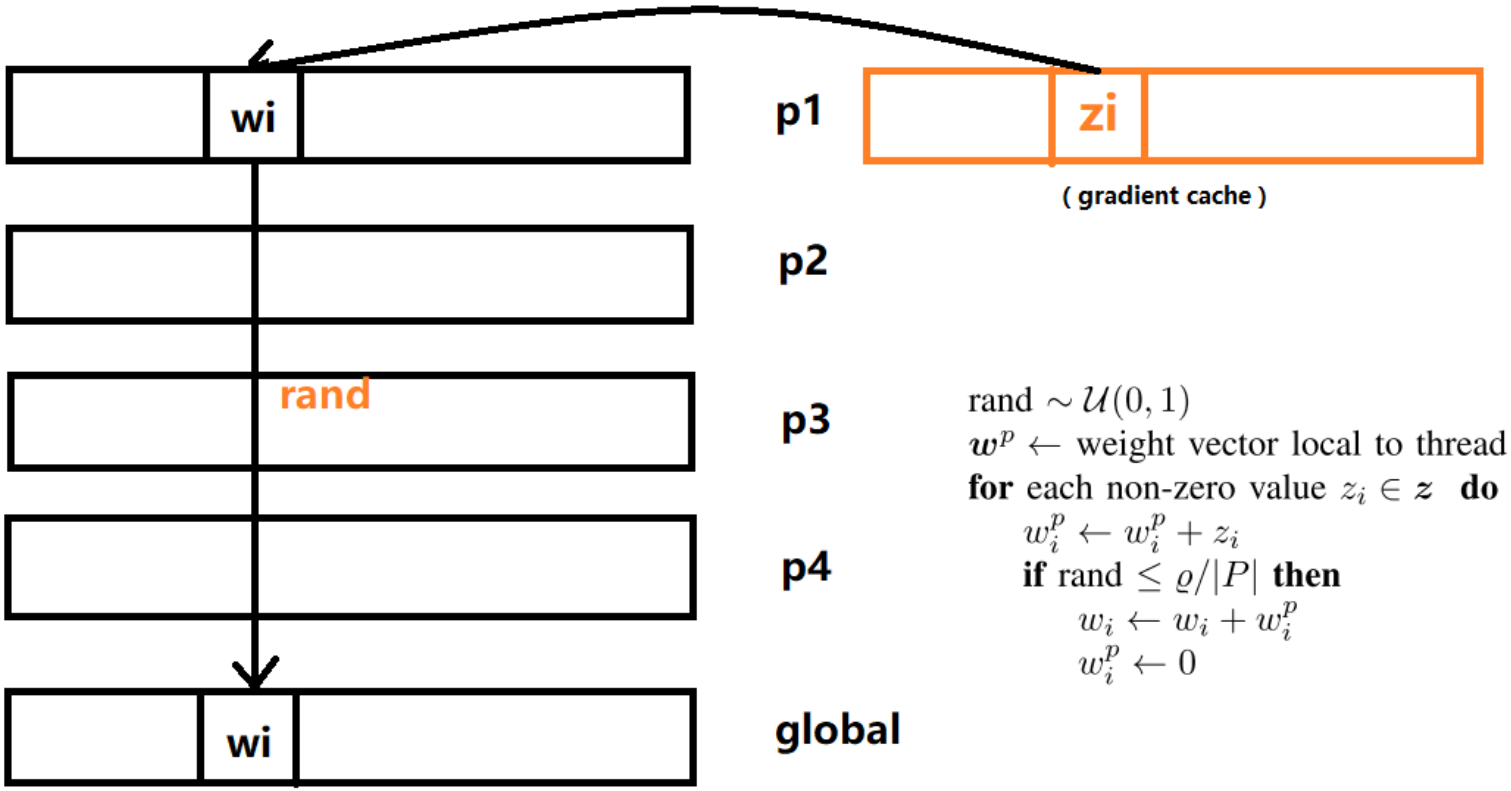
- asynchronous update scheme like Hogwild

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**

SAUS

update stage



rand $\sim \mathcal{U}(0, 1)$
$\boldsymbol{w}^p \leftarrow$ weight vector local to thread p.
for each non-zero value $z_i \in \boldsymbol{z}$ **do**
  $w_i^p \leftarrow w_i^p + z_i$
  if rand $\leq \varrho/|P|$ **then**
    $w_i \leftarrow w_i + w_i^p$ ▷
    $w_i^p \leftarrow 0$

SAUS

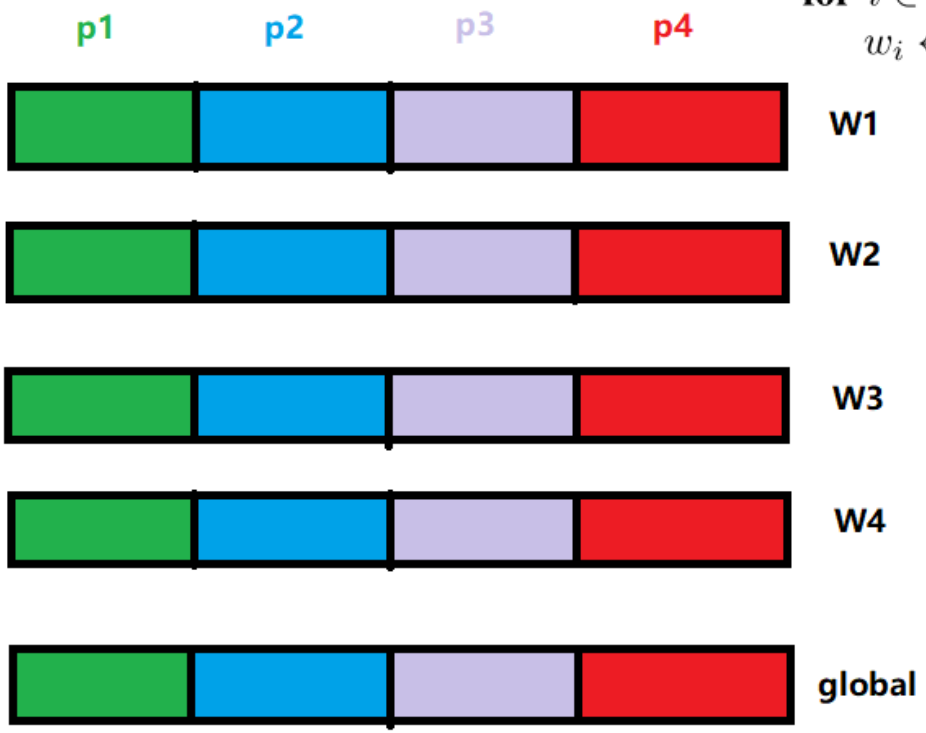
update stage

- update stochastically reduce the number of operation to the global weight vector, thus reduce the possibility of collision
- not fully transfer all information that is contained in the local weight
- on average, transfers information from frequently used features from the working threads to the global state

SAUS

accumulate stage

procedure ACCUMULATE(All thread contexts P)
  for $p \in [0, |P| - 1]$ **do** ▷ in parallel
    for $i \in [{}^{pD}/_{|P|}, {}^{(p+1)D}/_{|P|}]$ **do**
      $w_i \leftarrow w_i + w_i^p$ ▷ non-atomic, but safe

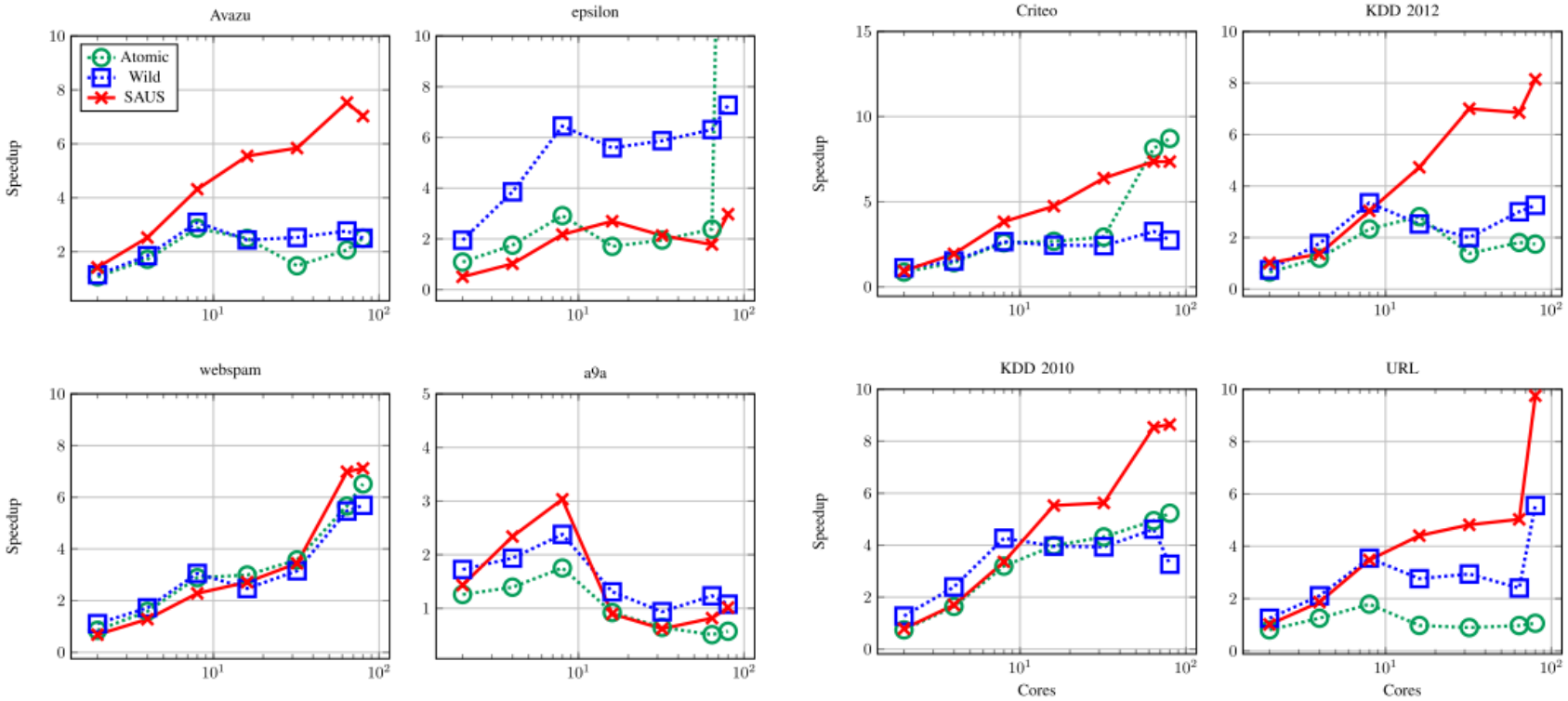


SAUS

Stochastic Dual Coordinate Ascent

- Stochastic Gradient Descent (SGD):
  - ✓ Strong theoretical guarantees.
  - ✗ Hard to tune step size (requires $\alpha \to 0$).
  - ✗ No clear stopping criterion (Stochastic Sub-Gradient method (SSG)).
  - ✗ Converges fast at first, then slow to more accurate solution.
- Stochastic Dual Coordinate Ascent (SDCA):
  - ✓ Strong theoretical guarantees that are comparable to SGD.
  - ✓ Easy to tune step size (line search).
  - ✓ Terminate when the duality gap is sufficiently small.
  - ✓ Converges to accurate solution faster than SGD.

- compared with Hogwild and Atomic method



my work

implement summary

- dataset: generate on the fly Linear/logistic regression problem
- parallelizing algorithm: hogwild and SAUS
- learning algorighm: SGD
- test the effect of different data density to locking method and Hogwild, the density includes [0.5%, 1%, 10%, 20%, 50%, 100%]
- other detail: C, thread, 8 cores and 20 cores machine

experiments and results

convergence curve

- dense data set converge better in the give number of steps, there is not big different between Hogwild and the locking scheme

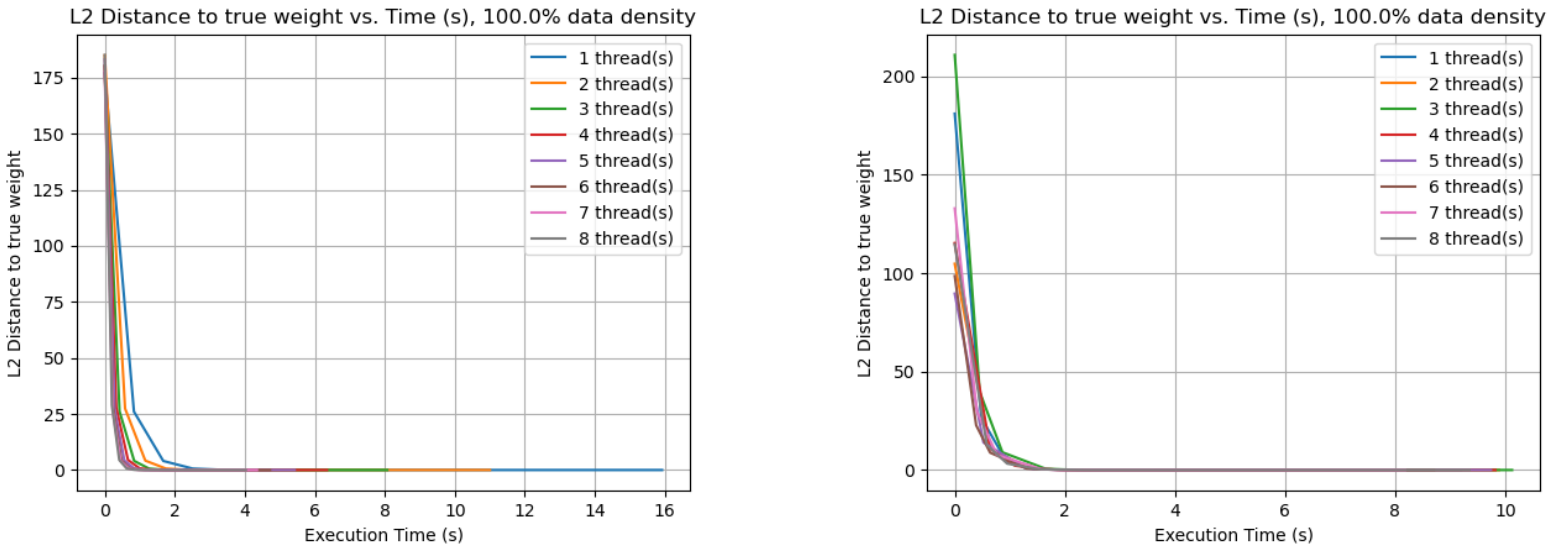


Figure: Execution time(s) vs. L2 distance to the true value, training linear regression with Hogwild (left column) and the Naive method (right column) in different data density, training was conducted in 8 cores machine

experiments and results

convergence curve

- when data set is less dense (20% of below), Hogwild has better speed when more cores are used, and Naive locking has worse speed

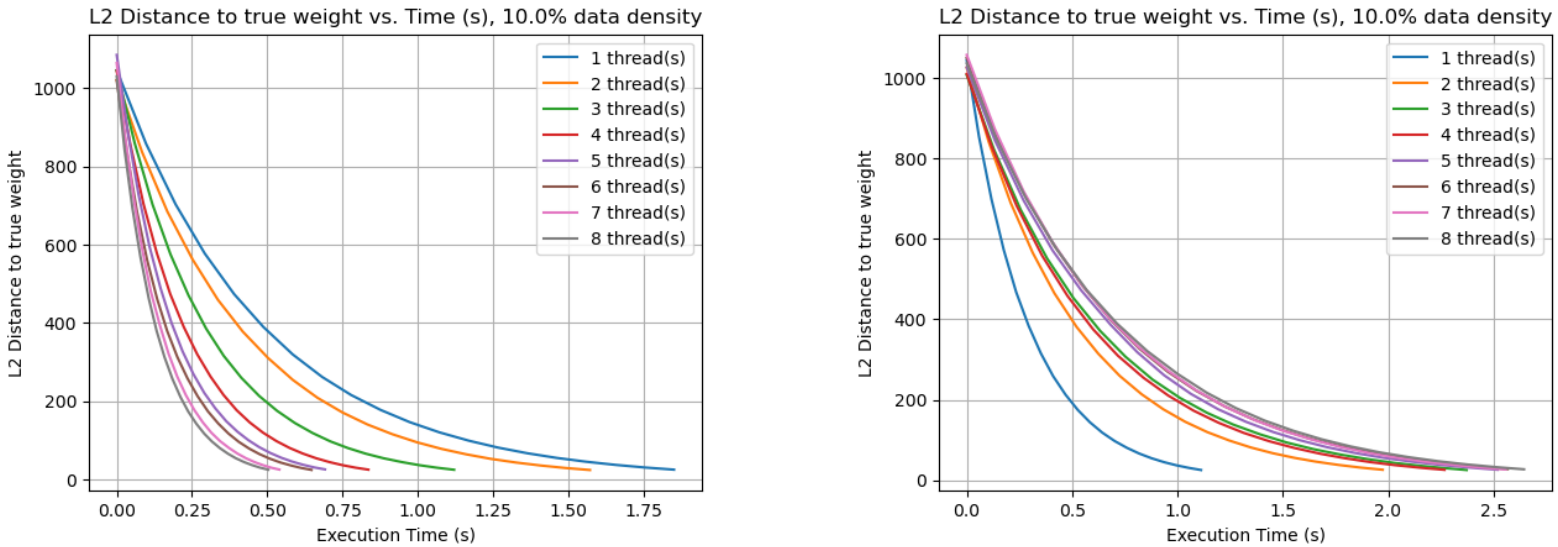


Figure: Execution time(s) vs. L2 distance to the true value, training linear regression with Hogwild (left column) and the Naive method (right column) in different data density, training was conducted in 8 cores machine

- Hogwild: the less densed data generally produce better speedup. The observation does not always true.

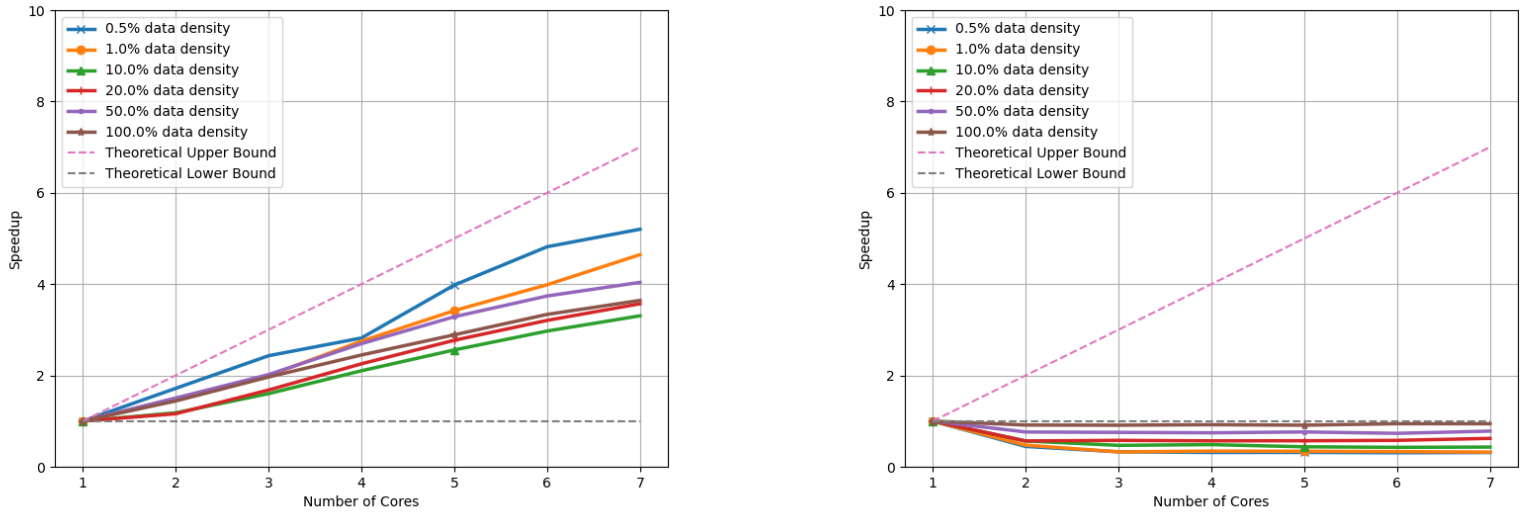


Figure: Speedup vs. num of cores, training linear regression with Hogwild (left column) and the Naive method (right column) in different data density, training was conducted in 8 cores machine

experiments and results

speedup

- the effect of density is less clear, curve tangled up

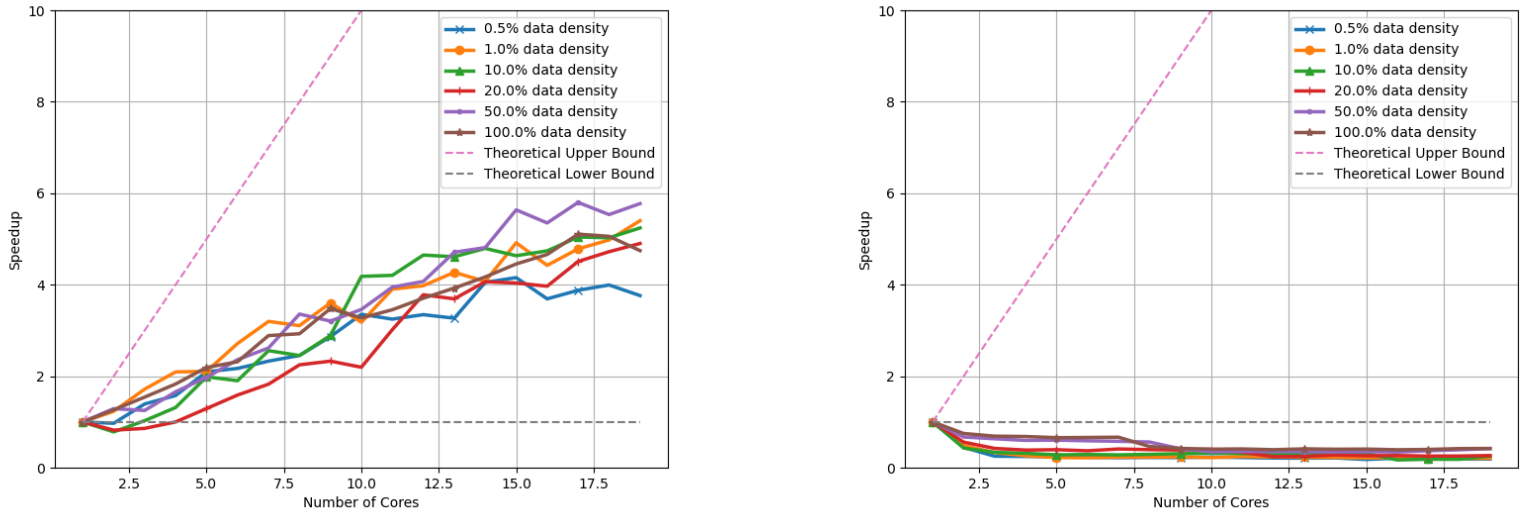


Figure: Speedup vs. num of cores, training linear regression with Hogwild (left column) and the Naive method (right column) in different data density, training was conducted in 20 cores machine

Questions

- which part of SGD has concurrency issue a) data sampling b) gradient calculation c) gradient update
- Why locks are not wanted when training linear models in parallel
- why SAUS was applied to Stochastic Dual Coordinate Ascent instead of Stochastic Gradient Descent?

The End


Raff, E., Hamilton, B. A., and Sylvester, J. (2018).
Linear models with many cores and cpus: A stochastic atomic update scheme.
2018 IEEE International Conference on Big Data (Big Data), pages 65–73.


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