

The cache issues about Hogwild! SGD

- 簡介

Support vector machine 是一種經典的機器學習演算法，能夠被應用於多種學習類型。本專案以廣為使用的 libsvm 為基礎架構，提供基本的 binary SVC 以及 epsilon SVR 兩類演算法，將其 solver 置換成 lock-free Hogwild! SGD 以及 HogBatch SGD，並利用 OpenMP 實現平行化。

另外，我們也利用 linux 的效能分析工具 perf 對原論文沒有多加著墨的 cache issue 做了許多分析，並根據分析的結果，對演算法的實現進行一些優化，減少執行過程中的 cache misses 次數。

最後，我也透過實驗證實了 Hogwild! 演算法確實只適用於 sparse dataset 上。一旦違反了這項前提，除了在收斂性上會付出代價，cache misses 的次數也會大幅提昇。

- Support vector machine 實作

程式的基本架構類似於 libsvm，但在 solver 的部份，libsvm 利用 SMO 演算法來求解 dual SVM 問題，而我則是利用 Hogwild! SGD 直接求解 primal SVM 問題。由於兩種方法在本質上的不同，程式的輸入參數也有所改變，詳細使用方法可見 README file。

SVM 標準的 loss function 如下：

$$\min_{\mathbf{w}} \frac{\lambda}{2} \|\mathbf{w}\|^2 + \frac{1}{m} \sum_{(\mathbf{x}, y) \in S} \ell(\mathbf{w}; (\mathbf{x}, y))$$

根據問題的需求，可選用不同的 function ℓ ，只要能夠計算 subgradient，即可利用 SGD 求解。

該專案實作的第一個類型是 binary SVC，採用的 hinge loss 及其 subgradient 分別如下：

$$\ell(\mathbf{w}; (\mathbf{x}, y)) = \max \{0, 1 - y \mathbf{w}^T \mathbf{x}\}$$

$$\nabla \ell(\mathbf{w}; (\mathbf{x}, y)) = \begin{cases} -y \mathbf{x}, & \text{if } y \mathbf{w}^T \mathbf{x} < 1 \\ 0, & \text{otherwise} \end{cases}$$

第二個類型則是 epsilon SVR，其 loss 及其 subgradient 分別如下：

$$\ell(\mathbf{w}; (\mathbf{x}, y)) = \max \{0, |y - \mathbf{w}^T \mathbf{x}| - \epsilon\}$$

$$\nabla l(\mathbf{w}; (\mathbf{x}, y)) = \begin{cases} \mathbf{x}, & \text{if } \mathbf{w}^T \mathbf{x} - y > \epsilon \\ -\mathbf{x}, & \text{if } y - \mathbf{w}^T \mathbf{x} > \epsilon \\ \mathbf{0}, & \text{otherwise} \end{cases}$$

前者用於二元分類問題，後者用於迴歸問題。

關於資料格式的部份，libsvm 所使用的 LIBSVM data format 即為 sparse representation，因此只需按照論文設計的方式處理好 regularization term，即可符合演算法對資料表示法的要求。

Hogwild! 的虛擬碼如下：

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  $G_e(x_e)$ 
4:   for  $v \in e$  do  $x_v \leftarrow x_v - \gamma G_{ev}(x_e)$ 
5: end loop
```

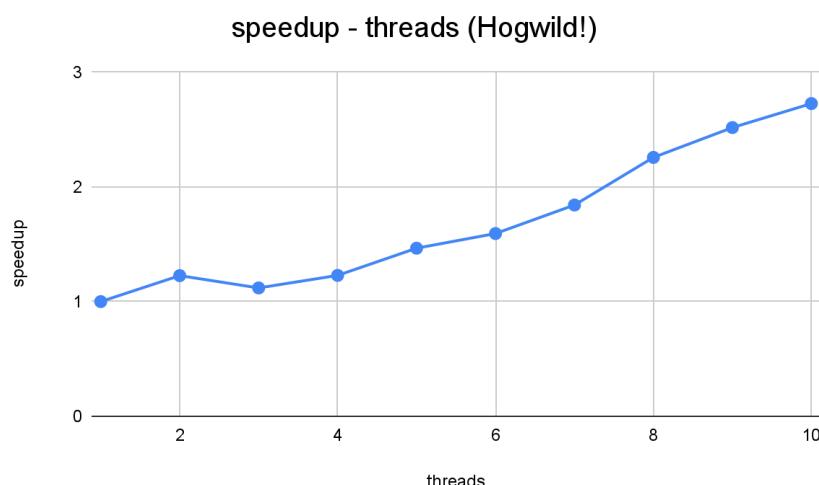
本專案利用 OpenMP 對 outer loop 實現平行化。

- **效能分析及優化**

我們使用的運算資源為 Intel Xeon E5-2620 processor，一次至多能使用 10 個執行緒進行運算。

在 sparsity 的評估上，我們以「sample vector 中，non-zero elements 所佔的平均比例」作為衡量標準，因為該定義比較符合效能分析的需求。

Hogwild! 原論文中宣稱該方法在 sparse dataset 上能夠達到 linear speedup，因此我們在 RCV1-binary dataset 上實測該演算法的執行效率，該 dataset 的 non-zero elements 佔比僅 0.0756。我們讓演算法執行 1000000 個 iterations，並只針對 training 的部份做計時。同一 thread number 的實驗會執行 10 次再取平均時間。詳細數據可見於最後實驗數據的部份。將 thread number 以及 speedup 的關係作圖如下：



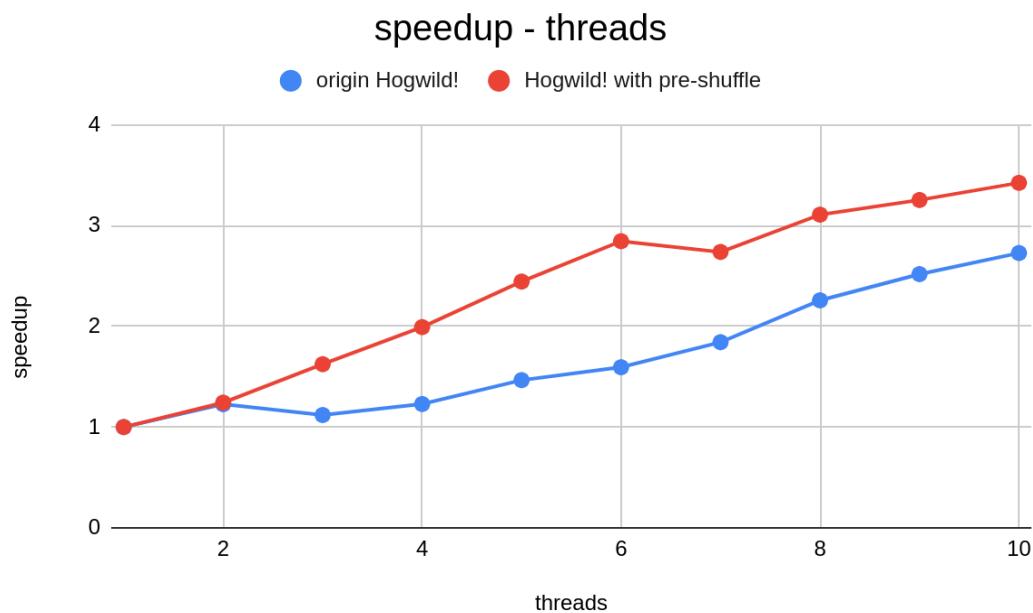
雖然 speedup 確實有明顯增長的趨勢，但斜率並不大，使用所有 threads 時，僅獲得約 2.7 倍加速。

由於該方法並沒有使用任何 lock 機制，因此最有可能破壞平行化效果的因素恐怕是 cache misses 造成的 overhead。我們利用 linux 內建的 perf 來分析 cache misses 次數，發現當 thread 數目超過 3 時，cache references 及 cache misses 的次數都會大幅提昇。

觀察 Hogwild! 演算法，可以發現兩個容易導致 cache misses 的主要原因。

第一個原因是資料存取時沒有 spatial locality 的性質。演算法在每回合是隨機挑選一個 sample feature 來計算 gradient，而非按照順序來存取每一筆 sample feature，而隨機挑選的 sample 有很高的機率不在 L1 cache 中，導致 processor 必須回到 L3 cache，甚至是 main memory 中才能取得該筆 sample，因此無法透過 cache prefetching 來減少 cache misses。

改善這個問題的方法相當簡單，我們每個 epoch 對 samples 做一次 shuffling，即可產生隨機的效果，接著只需要循序存取每一筆 sample 即可。使用 t 個執行緒執行程式時，我們將所有 samples 均分為 t 等分，每一個執行緒只會存取自己分配到的 samples，如此一來，每個 core 的 cache 都能發揮 prefetching 的效果。利用改進後的程式來進行前面的實驗，可以發現 cache misses 次數有顯著減少，且執行時間縮短，speedup 的效果也有所提昇，最多可達到 3.4 倍加速。



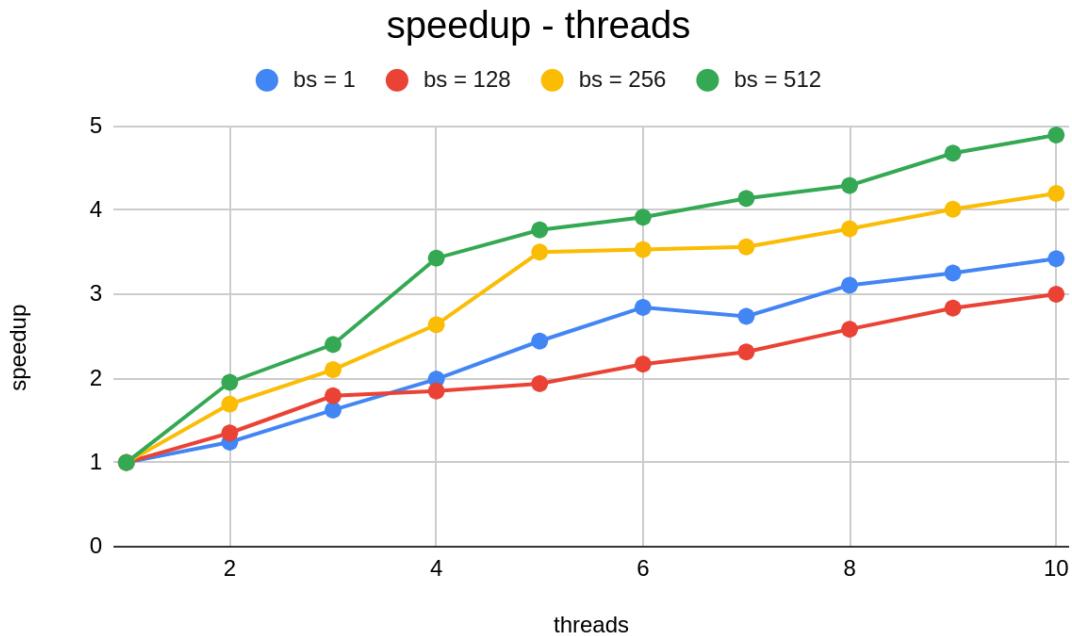
第二個可能影響 cache misses 次數的原因在於 cache coherence 及 false sharing 的問題。threads 在每個回合都會讀取以及寫入存放在共享記憶體的 weight vector，因此每個 core 的 L1 cache 都會保有一份 weight vector 的 copy。但當任何一個 thread 對 weight vector 的某個變數做寫入時，所有其他 cache 中存放該變數的 cache line 都會被 invalidated，因此當其他 thread 要對該變數做讀取或寫入時，為了維持 cache coherence，勢必要將更新後的資料先寫回 main memory，再從 main memory 中將資料拉回各個 core 的 L1 cache。當使用的 thread 數目越多時，update 會變得越頻繁，這一類的 core-to-core

communication 也會增加，因此能夠解釋實驗中出現的 cache misses 增加的情形。而這個問題在不稀疏的 dataset 上會顯得更嚴重。

為了改善這個問題，我們採用 "High Performance Parallel Stochastic Gradient Descent in Shared Memory" 提出的 HogBatch 演算法。這篇論文將 HogBatch 用來處理 logistic regression 問題，並取得了不錯的實驗結果，我認為該方法同樣適用於 SVM。該演算法的核心想法是，每個 thread 在一個回合中針對多筆 sample features (稱為一個 batch) 計算 gradient，並將結果累加於該 thread 的私有變數中，算完所有的 gradient 後，再將存放於私有變數的 gradient 總和更新到共享的 weight vector 中。如此一來，可以讓多數的寫入操作都發生在 private variable，有效減少 weight vector 的存取頻率，減輕維持 cache coherence 的成本。這個演算法的重要參數是 batch size，當 batch size 越大，weight vector 的存取頻率就越低，但 weight vector update 的延遲會變得較嚴重，可能影響到收斂率。HogBatch 的虛擬碼如下：

```
parallel for (i = 1; i < N; i += batch_size)
    batch_gradient = 0
    for (index = i; index < i + batch_size; index += 1)
        batch_gradient += grad(model, x(index))
    w = w + batch_gradient
```

我們實作 HogBatch 演算法，並分別對 batch size = 64, 128 及 256 來進行實驗，將 speedup 作圖如下：



從附錄及圖表的數據不難發現，batch size 增加能夠使 cache misses 次數減少，並且讓 speedup 效果更好。在 batch size = 512 的情況下，最多能獲得接近5倍的 speedup。

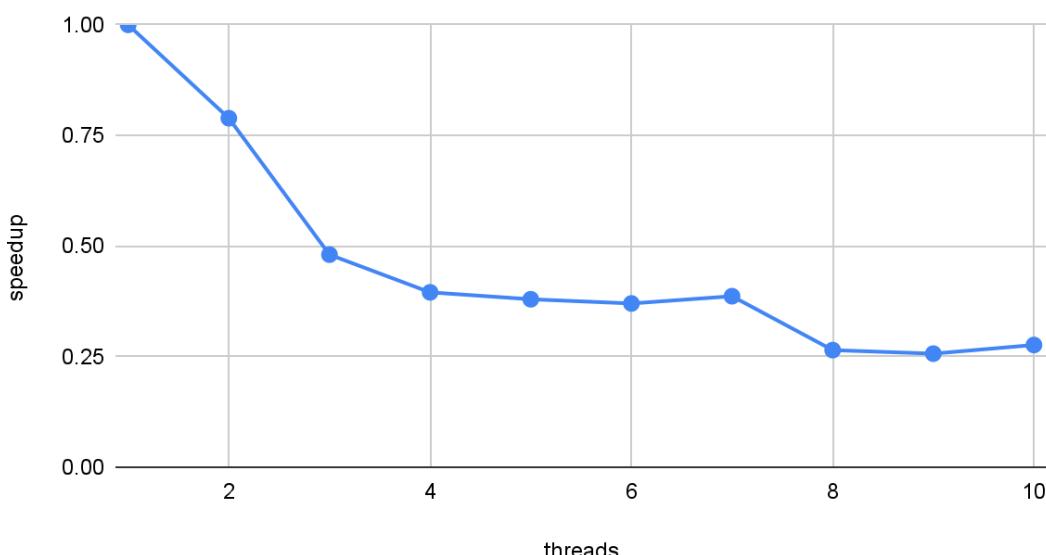
另外，從收斂性的角度出發，在使用 10 個 threads 的情形下，不論選擇 Hogwild! 或是 batch size 分別為 128, 256, 512 的 HogBatch，大約都能在 9000 ~ 9500 個 iterations 讓 training accuracy 到達 95%。因此根據該組實驗的結果，HogBatch 對於收斂性的影響並不嚴重。

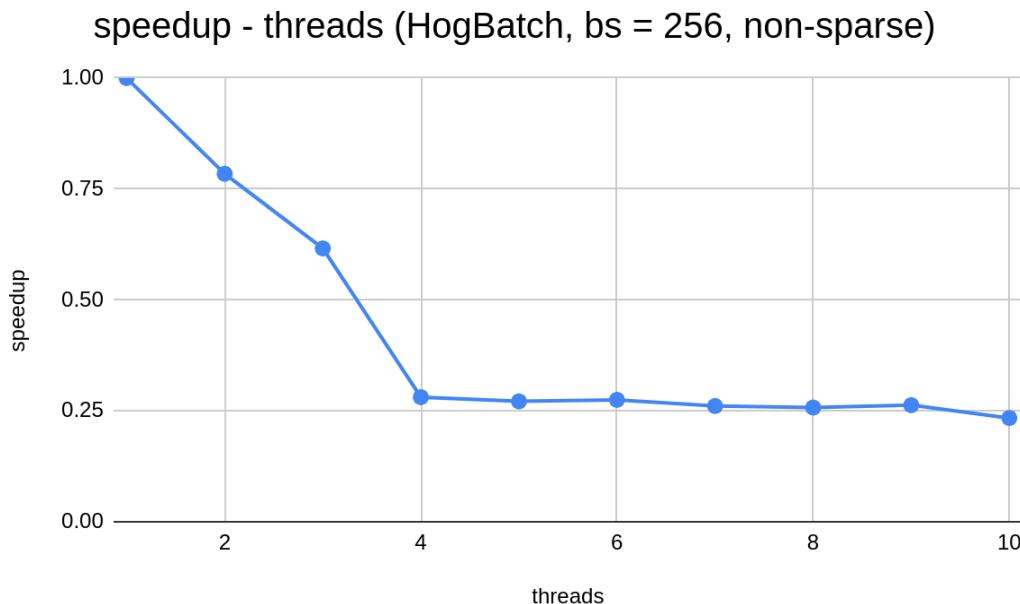
最後，我們考量非稀疏的 dataset。這裡選用 scale 過的 abalone regression dataset 做實驗，採用 epsilon SVR 方法，同樣執行 1000000 個 iterations。該 dataset 有 4177 筆 samples，feature dimension 僅為 8，但 non-zero elements 佔比高達 0.964。

可以發現，不論使用 Hogwild! 或是 HogBatch，都無法獲得任何平行化的好處。透過觀察不難發現，在 non-sparse dataset 上，cache misses 次數和 thread 數目呈現較明顯的正相關，而且儘管 dataset size 較小，cache misses 的數量級卻沒有比使用 RCV1-binary 時來得小。

利用 perf 觀察程式執行熱點，發現超過半數的執行時間都花費在寫入 weight vector。因為在 non-sparse 的情況下，每一回合都會對 weight vector 的大多數 components 做寫入，因此幾乎每一次存取 weight vector 都會發生 cache miss。因此單從 cache issue 的角度來看，在 shared memory 的架構之下，Hogwild 及其衍生的平行化方法幾乎沒有辦法在 non-sparse dataset 上獲得任何的加速。

speedup - threads (Hogwild!, non-sparse)





- 結論

藉由在 SVM 軟體上實作 Hogwild! solver, 我們對平行化加速 SGD 的可行性做了基本驗證。

並且, 我們針對該演算法實現過程中面臨到的cache issue 做了較多的探討, 並據此對演算法做小幅的優化。包含利用 pre-shuffle 來代替 random sampling, 以發揮 cache prefetching 的設計優點, 以及利用 HogBatch 的作法來減少寫入共享記憶體的頻率, 進而減少 core-to-core communication。

最後, 我們也驗證了 Hogwild 使用於 non-sparse dataset 的限制, 並針對其原理做了簡單的分析。

- Github 連結

<https://github.com/bird-cat/Hogwild-experiments>

● 實驗數據

表一 (Hogwild! on RCV1-binary)

| threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|--------------------|-----------|-----------|------------|------------|------------|------------|------------|------------|------------|------------|
| execution time(us) | 1396053 | 1138287 | 1247242 | 1136457 | 952448 | 876031 | 757728 | 618389 | 554480 | 511904 |
| speedup | 1 | 1.22645 | 1.11931 | 1.22842 | 1.4657 | 1.5936 | 1.8424 | 2.2575 | 2.5177 | 2.7271 |
| cache misses | 9,627,199 | 8,573,289 | 43,064,008 | 60,683,145 | 83,637,554 | 90,671,553 | 87,848,356 | 23,169,942 | 11,321,933 | 90,114,178 |

表二 (Hogwild! with pre-shuffle on RCV1-binary)

| threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|--------------------|-----------|-----------|------------|------------|------------|------------|------------|-----------|-----------|-----------|
| execution time(us) | 1315992 | 1059021 | 810391 | 660652 | 538473 | 462609 | 480612 | 423397 | 404493 | 384357 |
| speedup | 1 | 1.24264 | 1.6238 | 1.9919 | 2.4439 | 2.8447 | 2.7381 | 3.1081 | 3.2534 | 3.4238 |
| cache misses | 6,052,413 | 8,057,296 | 30,102,400 | 50,958,848 | 40,948,716 | 32,736,797 | 41,046,089 | 8,070,621 | 5,938,758 | 5,080,817 |

表三 (HogBatch with bs = 128 on RCV1-binary)

| threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|--------------------|-----------|-----------|------------|------------|------------|------------|------------|-----------|-----------|-----------|
| execution time(us) | 1526237 | 1129067 | 851013 | 825037 | 787490 | 703150 | 659358 | 590430 | 538067 | 508569 |
| speedup | 1 | 1.35176 | 1.7934 | 1.8499 | 1.9381 | 2.1705 | 2.3147 | 2.5849 | 2.8365 | 3.0010 |
| cache misses | 9,570,695 | 7,742,960 | 52,815,882 | 13,968,944 | 44,562,246 | 57,465,216 | 41,802,454 | 5,739,960 | 7,512,131 | 8,639,034 |

表四 (HogBatch with bs = 256 on RCV1-binary)

| threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|--------------------|-----------|-----------|------------|-----------|-----------|------------|-----------|-----------|-----------|-----------|
| execution time(us) | 1143821 | 674978 | 543524 | 433502 | 326576 | 323818 | 320898 | 302655 | 285096 | 272338 |
| speedup | 1 | 1.6946 | 2.1044 | 2.6385 | 3.5024 | 3.5322 | 3.5644 | 3.7792 | 4.0120 | 4.2000 |
| cache misses | 9,499,732 | 5,705,927 | 14,753,995 | 7,711,169 | 7,244,099 | 12,891,137 | 4,200,065 | 4,669,744 | 6,407,821 | 5,656,146 |

表五 (HogBatch with bs = 512 on RCV1-binary)

| threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|--------------------|-----------|-----------|------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| execution time(us) | 1096191 | 561004 | 456079 | 319563 | 291024 | 279858 | 264774 | 255296 | 234308 | 223967 |
| speedup | 1 | 1.9539 | 2.4035 | 3.4302 | 3.7666 | 3.9169 | 4.1401 | 4.2938 | 4.6784 | 4.8944 |
| cache misses | 6,953,468 | 5,491,216 | 13,434,106 | 7,182,921 | 4,258,928 | 3,713,378 | 3,092,467 | 3,721,642 | 3,687,286 | 3,141,684 |

表六 (Hogwild! on abalone_scale)

| threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|--------------------|--------|--------|--------|--------|--------|--------|--------|---------|---------|---------|
| execution time(us) | 313345 | 397214 | 652189 | 792886 | 825477 | 846695 | 810809 | 1184035 | 1221058 | 1134420 |
| speedup | 1 | 0.7888 | 0.4804 | 0.3951 | 0.3795 | 0.3700 | 0.3864 | 0.26464 | 0.25661 | 0.27621 |
| cache misses | 568882 | 512189 | 955262 | 926476 | 801351 | 596964 | 16702 | 76218 | 60399 | |

表七 (HogBatch with bs = 256 on abalone_scale)

| threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|--------------------|--------|--------|-----------|------------|------------|------------|------------|------------|------------|------------|
| execution time(us) | 687754 | 877007 | 1116572 | 2450565 | 2535450 | 2503914 | 2638193 | 2672394 | 2618137 | 2942718 |
| speedup | 1 | 0.7842 | 0.61595 | 0.28065 | 0.27125 | 0.27467 | 0.26069 | 0.25735 | 0.26268 | 0.23371 |
| cache misses | 13,536 | 10,413 | 2,304,249 | 50,634,806 | 51,238,362 | 48,742,729 | 52,186,397 | 53,093,987 | 52,826,899 | 59,767,735 |

● References

1. Feng Niu, Benjamin Recht, Christopher Ré and J. Wright Stephen, "HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent", *Advances in Neural Information Processing Systems 24*, pp. 693-701, 2011
2. Scott Sallinen, Nadathur Satish, Mikhail Smelyanskiy, Samantika S Sury, and Christopher Ré . 2016. High performance parallel stochastic gradient descent in shared memory Parallel and Distributed Processing Symposium, 2016 IEEE International. IEEE, 873--882.