PARALLEL STOCHASTIC GRADIENT DESCENT WITH SUB-ITERATIONS ON DISTRIBUTED MEMORY SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Orhun Çağlayan
February 2022

PARALLEL STOCHASTIC GRADIENT DESCENT WITH SUB-ITERATIONS ON DISTRIBUTED MEMORY SYSTEMS
By Orhun Çağlayan
February 2022

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

M. Mustafa Özdal(Advisor)

Cevdet Aykanat(Co-Advisor)

Özcan Öztürk

Murat Manguoğlu

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

PARALLEL STOCHASTIC GRADIENT DESCENT WITH SUB-ITERATIONS ON DISTRIBUTED MEMORY SYSTEMS

Orhun Çağlayan
M.S. in Computer Engineering
Advisor: M. Mustafa Özdal
February 2022

We investigate parallelization of the stochastic gradient descent (SGD) algorithm for solving the matrix completion problem. Applications in the literature show that stale data usage and communication costs are important concerns that affect the performance of parallel SGD applications. We first briefly visit the stochastic gradient descent algorithm and matrix partitioning for parallel SGD. Then we define the stale data problem and communication costs. In order to improve the performance of parallel SGD, we propose a new algorithm with intra-iteration synchronization (referred as sub-iterations) to decrease communication costs and stale data usage. Experimental results show that using sub-iterations can decrease staleness up to 95% and communication volume up to 47%. Furthermore, using sub-iterations can improve test error up to 60% when compared to the conventional parallel SGD implementation that does not use sub-iterations.

Keywords: Stochastic gradient descent, matrix completion, parallel computing, distributed memory systems.

ÖZET

DAĞITIK BELLEKLİ SİSTEMLERDE ALT-İTERASYONLU PARALEL OLASILIKSAL GRADYAN ALÇALMA

Orhun Çağlayan
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: M. Mustafa Özdal
Şubat 2022

Matris tamamlama problemi için paralel olasılıksal gradyan alçalma(SGD) algoritması incelenmektedir. Literatürdeki uygulamalar bayat veri kullanımı ve iletişim ücretlerinin paralel SGD'nin performansını etkileyen önemli etmenler olduğunu göstermektedir. İlk olarak SGD algoritmasını ve paralel SGD için matris bölümlemesini incelenmektedir. Daha sonra paralel SGD'nin performansını iyileştirmek için yeni bir algoritma önermekteyiz. Bu algoritma iterasyon içi senkronizasyonlarla (alt-iterasyon olarak adlandırılmaktadır) iletişim ücretlerini ve bayat veri kullanımını azaltmayı amaçlamaktadır. Deney sonuçları alt-iterasyon kullanımında bayat veri kullanımının %95 oranında ve iletişim hacminin %47 oranında azaldığını göstermektedir. Bununla beraber, alt-iterasyon kullanımının test verisi hatasını alt-iterasyonsuz algoritmayla karşılaştırıldığında %60 oranında iyileştirebildiği görülmüştür.

Anahtar sözcükler: Olasılıksal gradyan alçalma, matris tamamlama, paralel programlama, dağıtık bellekli sistemler.

Acknowledgement

First of all, I would like to thank to my advisors Assoc. Prof. Dr. M. Mustafa Özdal and Prof. Dr. Cevdet Aykanat for their enormous help and guidance. Without their patience and wisdom, I wouldn't be able to complete my studies. They helped me to be a wiser person. I would also like to thank Prof. Dr. Özcan Öztürk and Prof. Dr. Murat Manguoğlu for kindly accepting to be in my thesis committee.

I would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) 1001 program for supporting me throughout my M.S. studies in the EEEAG-119E035 project.

I would like to thank to my parents C. Serap Çağlayan and H. İhsan Çağlayan for their unlimited help. They always supported me even when they disagree with my actions. Without them, I wouldn't be who I am now. I would also like to thank to all my family for their endless support.

I would like to thank to all my friends for their friendship. They make me a better and a much happier person. I would also like to thank to my dear friends Selin Erdem for helping me through anything with her unending support, Necati Berk Özgür and Muhammed Baykal for cheering me up even in the worst of times, and Abdullah Talayhan for listening me everytime when I need somebody to talk to.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

With the emergence of Web 2.0 and user-focused web products, providing the users right content has become more and more important. Recommender systems have been at the center of this since their inception. Especially with the popularization of e-commerce and the streaming platforms, recommendation systems are generously employed in order to increase user engagement and monetary spending. E-commerce platforms use recommender systems to provide users the products they would like. Streaming platforms use them to provide users the content they would probably want to watch. The famous Netflix Prize in 2009 [1, 2, 3] was a competition organized in order to improve Netflix's own recommender system algorithm. With its 1 million dollar prize, it shows how the company places emphasis on recommender systems. BellKor's Pragmatic Chaos team won the competition, predicting the ratings better than Netflix's own algorithm by more than 10% [4]. Hence, even a fraction of improvement is a crucial gain for recommender systems.

One of the most popular methods used in recommended system is the Stochastic Gradient Descent(SGD) Algorithm. Just like any other algorithm that uses data in large scale, SGD has one caveat: it becomes increasingly harder to run SGD on a sequential platform as the data size increase. Run on billions of data

entries with multiple iterations, it is obvious that SGD can benefit from parallelization. However, SGD is not embarrassingly parallel and non-trivial to parallelize. Therefore, parallelization of SGD is throughly studied in the literature.

In this thesis, we first propose a naive row-parallel SGD algorithm on a distributed memory system such that each nonzero of rating matrix is classified as local or non-local and only the non-local nonzeros create parallel communication. On top of the naive algorithm, we propose a second algorithm that aims to decrease communication cost and stale data usage by employing an intra-iteration global gradient update scheme called sub-iterations. Our experimentary results show that using sub-iterations can decrease stale data usage up to 97% and communication volume up to 47%. Moreover, using sub-iterations can decrease the test error up to 60%.

The rest of the thesis contains a brief background information about SGD and Input matrix partitioning in Chapter 2. We define the necessary metrics regarding parallelization of SGD in Chapter 4. Then, several different SGD parallelization methods are explored in detail in Chapter 3. We then explain our proposed parallel SGD algorithms Chapter 5. Experimental results of our proposed algorithm is presented in Chapter 6. Finally, future work and conclusion is presented in Chapter 7.

Chapter 2

Background

2.1 Notation

We denote matrices with capital letters. For a matrix A we denote its entry at i-th row and j-th column as a_{ij}. We also denote a row of a matrix A as a_i.

2.2 Rating Matrix

In order to explain Stochastic Gradient Descent, the rating matrix should be defined. A rating matrix R is consisted of entries r_{ui} such that r_{ui} corresponds to user u's rating to item i. Since it is nearly impossible for users to rate most of the items in the dataset, rating matrices are almost always sparse.

2.3 Stochastic Gradient Descent

Given an $m \times n$ sparse rating matrix R, m being the number of users and n being the number of items in the dataset, Stochastic Gradient Descent (SGD) algorithm

aims to predict r_{ui}, rating of item i given by user u. In order to perform this task, a vector of length f is defined for each user u and item i denoted as p_u and q_i respectively. These vectors are called latent factor vectors. The sets of all p_u and all q_i are called latent factor matrices. Latent factor matrix of users is denoted as P and latent factor matrix of items is denoted as Q in the rest of this thesis. Size of P is $m \times f$ and size of Q is $n \times f$. Latent factor vector of a user or an item essentially captures a user's or item's characteristics. Using these latent factors, a rating is predicted for each nonzero entry.

$$\hat{r}_{ui} = p_u q_i^T \tag{2.1}$$

This predicted rating is then used to compute the prediction error.

$$e_{ui} = r_{ui} - p_u q_i^T \tag{2.2}$$

Koren et al. [5] suggest that the precision of the rating estimation can be increased by adding the rating average and user biases .

$$\hat{r}_{ui} = p_u q_i^T + \mu + b_i + b_u \tag{2.3}$$

Rating average is the average of all nonzero entries of the rating matrix. Biases capture an item's or a user's tendency compared to the average. For example if a user u's rating average is 0.35 higher than the average, then $b_u = 0.35$.

The error function used in SGD can vary. The algorithm converges to optimal solution set by minimizing the error function. In our implementation, we use the sum of squared loss error with L2 regularization:

$$Q(w) = \min_{p*,q*,b*} \sum_{u,i \in \kappa} (r_{ui} - p_u q_i^T - \mu - b_i - b_u)^2 + \lambda(\|p_u^2\| + \|q_i^2\| + b_u^2 + b_i^2) \tag{2.4}$$

Then, latent factors are updated using the gradient of the error function with respect to the respective parameter. As the name suggests, this is the gradient descent step.

$$p_u \leftarrow p_u + \alpha \nabla Q(w) \tag{2.5}$$

Here:

$$\begin{aligned}
\nabla Q(w) &= \frac{\partial}{\partial p_u} \min_{p*,q*,b*} \sum_{u,i \in \kappa} (r_{ui} - p_u q_i^T - \mu - b_i - b_u)^2 + \lambda(\|p_u^2\| + \|q_i^2\| + b_u^2 + b_i^2) \\
&= -2q_i^T (r_{ui} - p_u q_i^T - \mu - b_i - b_u) + 2\lambda(\|p_u\|) \\
&= -2q_i^T e_{ui} + 2\lambda p_u
\end{aligned}$$

Then, substituting the simplified gradient to the equation:

$$p_u \leftarrow p_u + \alpha(-2e_{ui} q_i^T + 2\lambda p_u) \tag{2.6}$$

After simplifying the coefficients:

$$p_u \leftarrow p_u - \alpha(e_{ui} q_i^T - \beta p_u) \tag{2.7}$$

The same method is applied to all the parameters:

$$\begin{aligned}
p_u &\leftarrow p_u - \alpha(e_{ui} q_i^T - \beta p_u) \\
q_i &\leftarrow q_i - \alpha(e_{ui} p_u^T - \beta q_i) \\
b_i &\leftarrow b_i - \alpha(e_{ui} - \beta b_i) \\
b_u &\leftarrow b_u - \alpha(e_{ui} - \beta b_u)
\end{aligned}$$

Here α and β are the learning rate and the decay factor respectively. Learning rate α is a parameter for the speed of learning. Higher learning rate will result in higher descent. However, increasing learning rate might lead to overshooting the optimal point and therefore missing a possibly better solution. Weight decay

β is a regularization method that penalizes large weights, the added weight term in the loss function manifests itself in the update rule as a fraction of the weight subtracted from itself along the gradient update.

2.4 Partitioning the Input Matrix

Partitioning of the input rating matrix and assigning the parts to the processors of the parallel system is an important process as it can affect the performance of the application crucially. In order to partition the matrix effectively, we used the following partitioning scheme.

Although these kind of datasets are sparse in nature, user and item specifications are quite different. Rating matrices are usually "tall and skinny", i.e. columns are densely populated and nonzeros on the rows are sparse. If an item is popular, then it would be rated by lots of users. However, probability that a user rates nearly all the items is negligible. Hence, we prefer partitioning the R matrix on denser dimension. Row-wise partitioning decreases stale data usage and communication of the P-matrix rows. we prefer row-wise partitioning as it is more scalable.

If all the nonzeros in column q_i is assigned to the same processor, the latent factor vector of q_i does not incur communication and is completely local. However, not all columns can be local. A column might be too dense to partition it into a single processor as it might cause load imbalance among the processors. Another possibility is that rows of the nonzeros of the q_i can have other nonzeros that are belonging to a dense column q_j. In those cases nonzeros of q_i will be partitioned among multiple processors and q_i will incur communication. Therefore, the aim of partitioning the data is to minimize communication volume and data staleness while preserving load balance among the processors. Data staleness will be explained further in Chapter 4.

An example 4-way row-wise partition of a matrix is given in Figure 2.1. Each

processor owns two blocks of the same row of the matrix: the block on the left side is the local block and does not create stale data problem. The respective latent factor vectors of the columns of these nonzeros are local and do not participate to the communication.

The right block is the shared block and represents the non-local nonzeros. The columns of these nonzeros are called linking-columns and participate in the communication. In a single sub-iteration scheme, W_1, W_2 and W_3 will update q_i using their local latent factor vector. In that case, neither will know the other two processors' update on their own local q_i vector. Hence, stale data problem occurs. If the number of sub-iterations are more than or equal to 3, then each processor can learn other two processor's updates before updating its local q_i vector. Hence, stale data usage can be minimized.

Our aim is to assigning rows to the processors such that communication is minimized. In order to realize the efficient partitioning scheme described above, PaToH hypergraph partitioning algorithm is employed [6]. Essentially, hypergraph partitioning intends to partition the matrix to the processors by minimizing the connectivity of the column q_i. In other words, connectivity of a column q_i is the number of processors that owns a nonzero in the q_i column. For example, connectivity for q_i and q_j in Figure 2.1 are equal to 3. If all the rows having a nonzero in q_i were assigned to the same processor, q_i's connectivity would be 1.

For the PaToH [7, 8] algorithm, R matrix is represented as a hypergraph. Then, this hypergraph is bisected into 2 parts. For a K-way partitioning, this bisection is performed for $log_2 K$ phases for a total of K parts. The algorithm works in three phases: Coarsening, initial partitioning and uncoarsening. In the coarsening phase, hypergraph is partitioned into m smaller hypergraphs. In the initial partitioning phase, the best bipartition for the coarsest graph is determined using a greedy algorithm. During the uncoarsesining phase, the bipartition found in the partitioning phase of the level i (for $i = m, m-1, ...1$) is refined and a partition for level $i-1$ is determined.

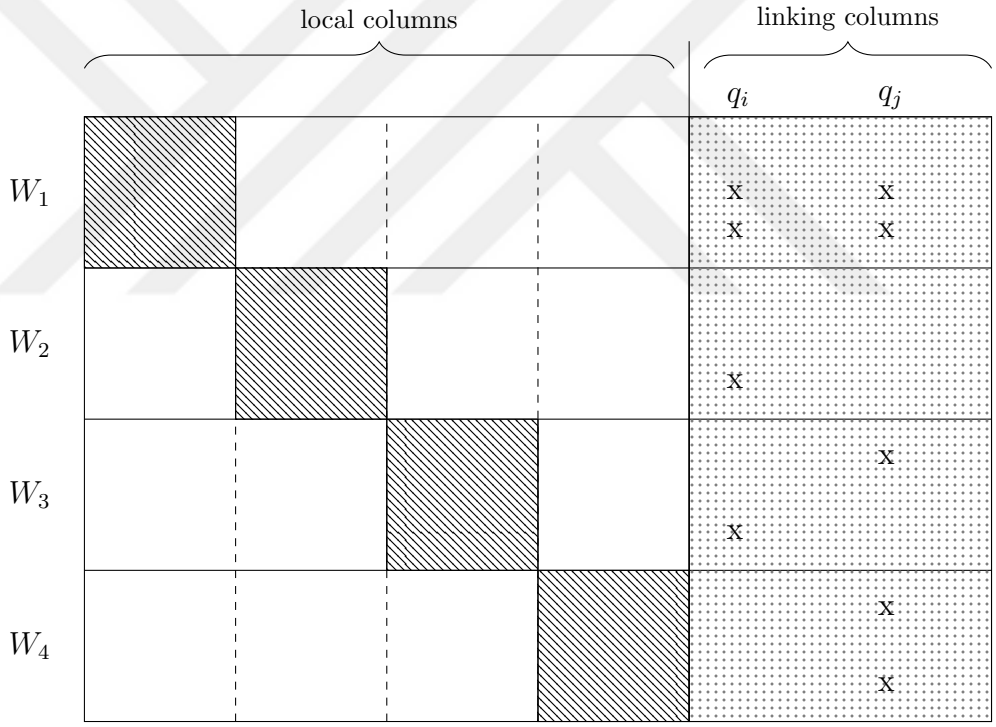

Figure 2.1: an example partioning for 4 processors

Chapter 3

Related Work

In this chapter, we briefly mention several parallel SGD implementations from the literature.

3.1 Distributed Stochastic Gradient Descent

Distributed Stochastic Gradient Descent (DSGD) algorithm by Gemulla et al. [9] proposes a block partitioning of the matrix such that no two distinct processors work on the same latent factor block at the same time. Each partition is called a stratum. Essentially, each processor selects a distinct block from a stratum and works on that block exclusively. After each stratum is processed, processing a stratum is defined as a sub-epoch, latent factor blocks are transmitted to the processor that will be using it in the next sub-epoch. Here, only communication cost comes from the transmission of latent factor blocks. However this can be especially costly when the latent factor size f increases. On top of that, a block in a stratum can be empty and then the processor assigned to that block would idle during the sub-epoch. This might decrease the performance of SGD especially in large-scale sparse matrices.

3.2 NOMAD

The NOMAD algorithm proposed by Yun et al. [10], goes one step further than DSGD and sends latent factors individually instead of transmitting latent factors as blocks . This approach naturally introduces a flexibility to the algorithm since the latent factors can be individually communicated instead of blocks. Furthermore, the algorithm is row parallel. Therefore, there is no communication regarding the latent factor matrix of the users. The algorithm introduces a queue for each processor. This is actually where the parallel communication happens: a processor sends the latent factors that it has to the processors that will be using them for the next update. This queue stores rating matrix entries and their respective item latent factors as pairs. In other words, each queue entry is a (r_{ui}, q_i), r_{ui} being user u's rating to item i and q_i being the latent factor of item i. While there is an item on the queue, an item is popped and SGD step is completed, therefore updating the latent factors corresponding to the rating matrix entry. Yun et. al. also introduce a load balancing scheme so that processors are not idle at any time. One downside of this approach is the possible overhead of communication: sending the latent factors after each single update can be costly especially if the latent factor size f increases.

3.3 Downpour-SGD

Downpour-SGD proposed by Dean et al. [11], utilizes asynchronous SGD scheme to improve the performance of SGD. The algorithm contains a set of worker processors and a master processor that communicates with workers. They partition the data into n batches and each batch is distributed to a worker. After each worker computes the gradients belonging to its batch, it sends the gradients to its corresponding shard in the master. The master then updates the respective gradients in each shard and workers fetch the gradients from their respective shards. As a result, each processor has the globally updated gradient.

This approach has its downsides though. Each worker needs to communicate with a single master and this might create a communication bottleneck, thus decreasing the performance. On top of that, a delayed worker sending the local gradients later can disrupt the gradient descent. The delayed gradient update can distrupt the descent when it is This might significantly alter the convergence time of SGD.

3.4 GASGD

The Graph Asynchronous SGD (GASGD) algorithm by Petroni and Quarzoni [12] introduce a novel partitioning approach to the SGD. They treat the data as a bipartite graph in which nodes are items and users and edges are the nonzero entries of the input matrix. They then use a greedy vertex-cut partitioning algorithm to distribute the data to the parallel processors. The greedy algorithm essentially partitions the data in a load balanced manner. Say that the set of processors that vertex v's nonzeros are distributed to are denoted as $A(v)$. Then, replication factor of the latent factor of v is proportional to the $|A(v)|$. The replication factor is important since the higher replication factor will mean more memory usage and higher amount of communication. Furthermore, user u's rating to item i is denoted as an edge e_{iv} between vertices u and i in the graph.

In order to decrease the replication factor of latent factors, they eliminate randomness by assigning the nonzeros according to a scheme. The aim is to assign each $e_{iv} \in E$ to a processor in either $A(v)$ or $A(i)$ so that replication factor is minimized. The synchronization is bulk and done f times in an iteration. This is quite similar to what we call sub-iteration in this thesis.

Chapter 4

Communication and Stale Data Issues

When parallelizing the SGD algorithm on a distributed memory systems, the main bottlenecks are communication costs and the stale data problem.

4.1 Communication Cost

The requirement to synchronize updated P and Q matrix vectors among all processors incurs communication cost. In the parallel implementations of the SGD algorithm, communication volume plays an important part. Considering the fact that latent factor matrix size f can be huge, communication overhead becomes quite important for the performance of the algorithm. If the importance of communication volume is ignored when implementing the sub-iteration method, performance of the algorithm can suffer significantly. Therefore, communication volume is one of the most important performance metrics for this problem.

Let σ_i be the number of sub-iterations that q_i is updated. Let $procs(q_i, s_j^i)$ be the number of processors that use q_i in the sub-iteration s_j^i. Processors that

update q_i is assumed to be distinct among all sub-iterations. Furthermore, let $\phi_i = \sum_{j=1}^{\sigma_i} |procs(q_i, s_j^i)|$ indicate the number of processors that update q_i in an iteration. Then, communication volume for q_i is:

$$\begin{aligned} volume(q_i) = (|procs(q_i, s_1^i)| - 1 + |procs(q_i, s_2^i)|) \times f + \\ (|procs(q_i, s_2^i)| - 1 + |procs(q_i, s_3^i)|) \times f + \\ (|procs(q_i, s_3^i)| - 1 + |procs(q_i, s_4^i)|) \times f + \\ \vdots \\ (|procs(q_i, s_{\sigma_i-1}^i)| - 1 + |procs(q_i, s_{\sigma_i}^i)|) \times f + \\ (|procs(q_i, s_{\sigma_i}^i)| - 1 + |procs(q_i, s_1^i)|) \times f. \end{aligned} \tag{4.1}$$

Note that each term is used twice. Therefore, total communication volume for the factor vector is q_i:

$$volume(q_i) = (2 \sum_{j=1}^{\sigma_i} (|procs(q_i, s_1^i)| - \sum_{j=1}^{\sigma_i} 1) \times f \tag{4.2}$$

Hence, if number of processors that use q_i is expressed as $\phi_i = \sum_{j=1}^{\sigma_i} |procs(q_i, s_j^i)|$, communication volume that q_i incurs can be simplified as:

$$volume(q_i) = \begin{cases} (2\phi_i - \sigma_i) \times f & if\ \sigma_i > 1 \\ 2(\phi_i - 1) \times f & if\ \sigma_i = 1 \end{cases} \tag{4.3}$$

If q_i is updated only in one sub-iteration throughout an iteration, owner of the q_i expands it to the processors using q_i before the sub-iteration that q_i is used in the next iteration. The case where q_i is being updated in only one sub-iteration is similar to the non sub-iteration algorithm. If q_i is updated in more than one sub-iteration throughout an iteration ($\sigma_i > 1$), increasing the number of sub-iterations that q_i is updated or decreasing the number of processors that update q_i will decrease the communication volume. In the best case scenario that minimizes communication volume, the number of processors that update q_i is equal to the number of sub-iterations that q_i is updated.

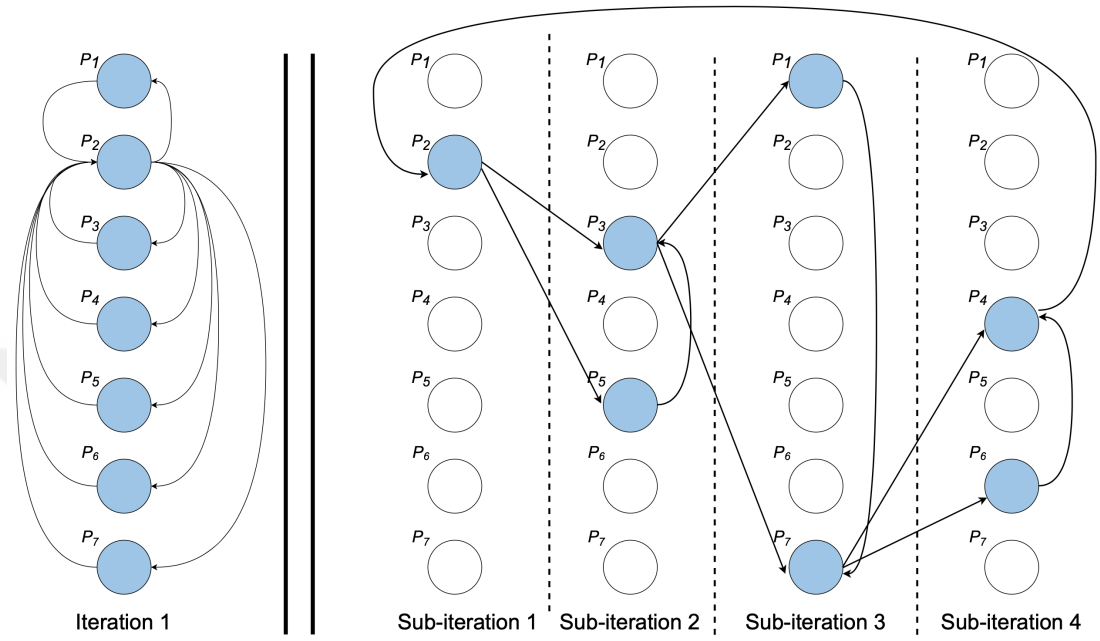

Figure 4.1: a sample communication pattern of a column q_i

An example communication pattern of a column q_i is given in Figure 4.1. The column is used by all the processors. The left side of the figure corresponds to an iteration without any sub-iterations. For the left side of the Figure, $\phi_i = 7$ and $\sigma_i = 1$. Therefore, $volume(q_i) = 2(7-1) = 12 \times f$. The right side of the Figure 2.1 corresponds to an iteration with 4 sub-iterations and $\phi_i = 7$ and $\sigma_i = 4$. Hence, $volume(q_i) = (2 \times 7 - 4) = 10 \times f$. This shows that using sub-iterations can decrease the communication volume.

4.2 Stale Data Problem

Stale data problem occurs when the calculation regarding the two nonzero entries in the same column or row of the R matrix are assigned to two different processors. As mentioned before, since we focus on row-wise partitioning, stale data problem occurs along the columns of R matrix. For example say that nonzeros r_{ui} and r_{vi} are partitioned to the processors W_x and W_y respectively. While updating their local q_i latent factor vector, W_x and W_y do not know the updates in each other.

Naturally, sequential SGD algorithm does not have any stale data problem. Stale data usage tends to slow down convergence time of SGD algorithm.

In the parallel distributed memory implementations of the SGD algorithm, P and Q vector matrices are synchronised among all processors at the end of each iteration, so that all the processors have the updated P and Q. In order to reduce stale data usage, an iteration can be partitioned into several sub-iterations and P and Q synchronizations can be done after each sub-iterations. While this approach decreases stale data usage, it also introduces a latency cost as a result of increasing number of synchronizations at each iteration. Therefore, this approach creates a trade-off between stale data usage and synchronization cost. In our research, we include optimization of stale data and synchronization cost trade-off.

Let $stl(q_i)$ denote the stale data usage for q_i in σ_i sub-iterations. Furthermore, let $s_1^i, s_2^i, s_3^i ... s_{\sigma_i}^i$ be the list of sub-iterations that q_i communicated. Then, $stl(q_i)$ can be expressed as:

$$\begin{aligned} stl(q_i) = &(|procs(q_i, s_1^i)| - 1) \times f + \\ &(|procs(q_i, s_2^i)| - 1) \times f + \\ &(|procs(q_i, s_3^i)| - 1) \times f + \\ &\vdots \\ &(|procs(q_i, s_{\sigma_i - 1}^i)| - 1) \times f + \\ &(|procs(q_i, s_{\sigma_i}^i)| - 1) \times f. \end{aligned} \tag{4.4}$$

In other words, distributing the processors updating q_i to as much sub-iterations as possible will yield better results in terms of stale data usage. If there is only one processor updating q_i in each sub-iteration, then stale data regarding q_i will be 0. Equation (4.4) can be simplified as:

$$stl(q_i) = \sum_{j=1}^{\sigma_i} (|procs(q_i, s_1^i)| - 1) \times f = \sum_{j=1}^{\sigma_i} (|procs(q_i, s_1^i)|) - \sum_{j=1}^{\sigma_i} 1)$$

It can be further simplified when ϕ_i defined earlier is used:

$$stl(q_i) = (\phi_i - \sigma_i) \times f$$

In the left part of the Figure 4.1, stale data coming from q_i is $stl(q_i) = (7 - 1) \times f = 6 \times f$, $\phi_i = 7$ and $\sigma_i = 1$. In The right side of the Figure 2.1, stale data coming from q_i is $stl(q_i) = (7 - 4) \times f = 3 \times f$, $\phi_i = 7$ and $\sigma_i = 4$. Therefore, using sub-iterations to update the latent factors can dramatically decrease stale data usage.

Chapter 5

Row-Parallel SGD Algorithm

In this chapter, we will first explain Row Parallel SGD Algorithm, without the sub-iterations. Then we will explain our parallel SGD algorithm with sub-iterations(PSGD-SI). As the latter is built on top of the former, understanding the naive algorithm is essential.

5.1 Parallel SGD Algorithm Without Sub-iterations

Assume that the rating matrix R is K-way row-based partitioned for K processors in a distributed memory system. Since each of the P-matrix rows are only updated by one processors and no communication is required for P-matrix, all the P-matrix rows are categorized as local in this partitioning. Contrarily, Q-matrix rows can be categorized both as non-local and as shared. Local Q-matrix rows correspond to the diagonal blocks of R matrix in Figure 2.1. Shared Q-matrix rows correspond to the linking-columns of R matrix and shown as the block on the right side of Figure 2.1. As discussed earlier, only shared Q-matrix rows require communication. Shared Q-matrix rows are further categorized as owner and non-owner for each processor.

Communication pattern of a shared Q-matrix row q_i is as follows: at the end of an iteration, processors sharing the q_i row send their local q_i row vector to the owner processor with the `REDUCE` operation. Then, owner of the q_i takes the average of all the incoming q_i rows and updates its local q_i. After that, owner of the q_i sends the updated q_i value to the non-owner processors via the `EXPAND` operation. The pseudocode for row parallel SGD algorithm is given in Algorithm 1.

Algorithm 1 Row Parallel SGD

1: **Input:** $R_k, \hat{P}, \hat{Q}, f, \alpha, \beta, numIter$
2: **for** $c \leftarrow 0$ **to** $numIter$ **do**
3: **for each** $r_{ui} \in R_k$ **do**
4: $e_{ui} \leftarrow r_{ui} - p_u q_i^T$
5: $p_u \leftarrow p_u + \alpha(e_{ui} q_i^T - \beta p_u)$
6: $q_i \leftarrow q_i + \alpha(e_{ui} p_u^T - \beta q_i)$
7: **end for**
8: Sparse `REDUCE` for shared $\hat{Q}$ rows
9: //`SEND` shared non-owned $\hat{Q}$ matrix rows, `RECEIVE` shared owned $\hat{Q}$ matrix rows
10: Compute shared owned $\hat{Q}$ rows
11: compute owned q_i vectors by taking the overall of incoming $\hat{q}_i$ vectors
12: Sparse `EXPAND` for shared updated $\hat{Q}$ rows
13: //`SEND` shared owned $\hat{Q}$ matrix rows, `RECEIVE` shared non-owned $\hat{Q}$ matrix rows
14: $localError \leftarrow 0$
15: **for each** $r_{ui} \in R_k$ **do**
16: $localError \leftarrow localError + (r_{ui} - p_u q_i^T)^2$
17: **end for**
18: $globalError \leftarrow$ `ALL-REDUCE` $localError$
19: $globalError \leftarrow globalError/localError$
20: **if then** $globalError \leq targetError$
21: **break**
22: **end if**
23: **end for**

5.1.1 Algorithm Inputs

In the row parallel SGD algorithm each processor takes the respective part of the R matrix as the first input. The nonzero matrix is stored in COO (coordinate list) sparse matrix format. Although CSR (compressed sparse row) matrix format is generally used in matrix applications, randomization required for the SGD applications is hard to achieve and can be costly when CSR format is employed. Therefore, COO format is selected for this application.

Second input of the algorithm is the list of distinct row and column indices of nonzeros assigned to each processor. For example, assume that nonzero entry $r_{37,15}$ is assigned to W_1. Then W_1's row and column indices file will contain 37 in row indices and 15 in column indices. Column indices of the R matrix are ordered as local, owner and non-owner columns. Since all rows are local in this algorithm, no such ordering is required for row indices.

nonzeros assigned to individual processors are read in parallel. Then, the global indices of nonzeros are converted to local indices and stored in that way. Global indices are localized such that local indices start from zero and numbered consecutively. With this conversion, the SGD algorithm can run on these partitioned matrices as if those partitions are independent matrices that are processed in parallel. The third input of the algorithm is the partition vector. The content of this input is the rank of the owner processor of each linking column. Since a processor needs to know to whom it should send its Q matrix rows corresponding to the non-owner linking columns after SGD update, this input is crucial for communication. This file is same for all processors. Therefore, it is first read by a single processor and is broadcasted to the rest of the processors afterwards.

5.1.2 Data Structure for Communication

After Parallel I/O and constructing data structures for local matrices, communication structure is created. First of all, each processor determines to whom it will

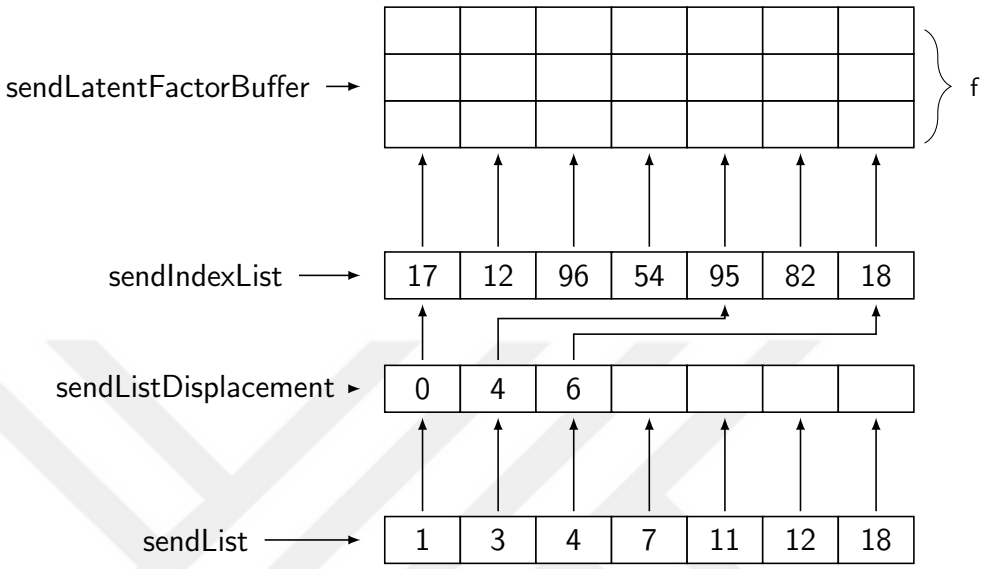

Figure 5.1: an example send data structure for a processor

send latent factors. During this process, each processor also counts the number of latent factor vectors that it will send. This information is sent to respective processors with `ALLTOALL` operation. At this point, each processor knows how many latent factor vectors it will send to and receive from each processor.

After this `ALLTOALL` communication, each processor determines several different vectors for communication. Send and receive processor vectors contains all the processor ids that a processor is communicating. Furthermore, send and receive index lists are stored. These index lists consist of the global indices of Q-matrix vectors to be sent to or be received from the other processors. To connect these two vectors, send and receive displacement vectors are stored. Displacement vectors indicate the start indices on the send and receive lists for each processor on the send and receive list.

Figure 5.1 shows an exemplary communication data structure for processor 0's send operation. sendList vector stores the indices of the owner processors of processor 0's non-owner columns. sendListDisplacement stores the starting indices on the sendIndexList for each processor in sendList. sendIndexList stores the global indices of the Q-matrix vectors to be sent. sendLatentFactorBuffer

structure stores the Q-matrix vectors for each global index. This buffer is used in the communication stage of each parallel SGD iteration.

Consider the communication between processor 0 and 1 as described in Figure 5.1. Processor 0 sends 4 latent factor vectors to the processor 1. Global indices of these vectors are 17, 12, 96 and 54, and these indices are stored in sendIndexList. The latent factor vectors that processor 0 will send to processor 1 are the first 4 vectors of sendLatentFactorBuffer. Since processor 1 is the first element of sendList, the value in sendListDisplacement corresponding to processor 1 is 0. Second element of sendListDisplacement is 4. This is because there are 4 vectors to be sent before.

5.1.3 Parallel SGD Iteration Stage

After each processor runs the local SGD algorithm, communication stage starts. This communication consists of two steps:

1. In the first step, non-owned shared Q-matrix rows that are updated during local SGD stage are sent to the owner processors. Owner processors sum the incoming Q-matrix rows according to their global index and take the average, thus finding the updated Q-matrix row.

2. In the second step, owners of the shared Q-matrix rows send the updated rows to the non-owner processors. Non-owner processors receiving these updated Q-matrix rows update their local corresponding Q-matrix row. Therefore, all the latent factor vectors are updated and snychronized among all processors. P-matrix rows are completely local and therefore do not require any communication.

After communication stage ends, local error is computed as described in Algorithm 1. Then, the global error is computed with an `ALLREDUCE` operation on all processors. After the global error is computed, the iteration finishes. Parallel

SGD algorithm can run until RMSE reaches a designated value or until a specific runtime or number of iterations.

5.2 Parallel SGD Algorithm with Sub-iterations

Algorithm 2 Row Parallel SGD with Sub-iterations

1: **Input:** $R_k, \hat{P}, \hat{Q}, f, \alpha, \beta, numIter, numSubiter$
2: **for** $c \leftarrow 0$ **to** $numIter$ **do**
3: **for** $s \leftarrow 0$ **to** numSubiter **do**
4: **for each** $r_{ui} \in R_k^s$ **do**
5: $e_{ui} \leftarrow r_{ui} - p_u q_i^T$
6: $p_u \leftarrow p_u + \alpha(e_{ui} q_i^T - \beta p_u)$
7: $q_i \leftarrow q_i + \alpha(e_{ui} p_u^T - \beta q_i)$
8: **end for**
9: Sparse `REDUCE` for shared $\hat{Q}^s$ rows
10: //`SEND` shared non-owned $\hat{Q}^s$ matrix rows, `RECEIVE` shared owned $\hat{Q}^s$ matrix rows
11: Compute shared owned $\hat{Q}^s$ rows
12: compute owned q_i^s vectors by taking the overall of incoming $\hat{q_i^s}$ vectors
13: Sparse `REDUCE` for shared updated $\hat{Q}^s$ rows
14: //`SEND` shared owned $\hat{Q}^s$ matrix rows, `RECEIVE` shared non-owned $\hat{Q}^s$ matrix rows
15: **end for**
16: $localError \leftarrow 0$
17: **for each** $r_{ui} \in R_k$ **do**
18: $localError \leftarrow localError + (r_{ui} - p_u q_i^T)^2$
19: **end for**
20: $globalError \leftarrow$ `ALL-REDUCE` $localError$
21: $globalError \leftarrow globalError / localError$
22: **if** $globalError \leq targetError$ **then**
23: **break**
24: **end if**
25: **end for**

As explained in Chapter 4 using sub-iterations can decrease communication cost and data staleness. Therefore, sub-iteration can greatly improve performance of the parallel SGD algorithm. As a result, we have incorparated sub-iterations to our algorithm in order to further enhance its performance. The pseudocode

for this algorithm is provided in Algorithm 2.

5.2.1 Algorithm Inputs

In the row-parallel SGD algorithm with sub-iterations each processor takes the respective part of the R matrix as the first input. The nonzero matrix is stored in COO (coordinate list) sparse matrix format. The nonzeros are ordered according to the sub-iterations. For each sub-iteration, nonzeros are ordered as local, non-owner and owner nonzeros.

Second input of the algorithm is the list of distinct row and column indices of nonzeros partitioned to each processor. This file is described in Subsection 5.1.1.

The third input of the algorithm is the partition vector file. Although this file is similar to the one required for row-parallel SGD without sub-iterations, there are differences. Each processor updates a Q-matrix row once every iteration. In other words, a Q-matrix row is updated in only one sub-iteration. Therefore, a Q-matrix row's owner is different for each sub-iteration. As a result, each sub-iteration requires its own partition vector.

Although a processor can find out to whom it should send a Q-matrix row without a partition vector, it would be costly in terms of communication. Each processor can send the global indices of the columns it owns with an `ALLTOALL` operation. However, this would require more communication than using the partitioning vector. Therefore, reading the partitioning vector obtained as a result of graph/hypergraph partitioning in I/O stage and broadcasting is a cheaper option.

Last input required is the list of distinct global column indices of nonzeros used in a sub-iteration, for each sub-iteration. A processor can determine to whom it would send to or receive from a Q-matrix row before the `EXPAND` and `FOLD` operations using this list and partition vector.

5.2.2 Data Structures for Communication

After the I/O stage ends and data structures for local matrices are created, data structures for the communication are created. Communication for the PSGD-SI algorithm has a one important difference from the naive algorithm which does not utilize sub-iterations. Note that in the naive algorithm `EXPAND` and `FOLD` operations are completed in the same iteration. On top of that, for each latent factor that is sent with `EXPAND`, there is a corresponding latent factor received with `FOLD` in an iteration. We define this property of naive algorithm as symmetrical. The benefit of symmetric communication is the fact that buffers used for `EXPAND` can also be used for `FOLD`.

In PSGD-SI, `FOLD` operation is completed inside the sub-iteration but `EXPAND` operation is done between the current and the next sub-iteration. Therefore, PSGD-SI utilizes asymmetrical communication. Furthermore, fold and expand data structures in PSGD-SI are created for each sub-iteration. these structures are constructed in a similar manner to the naive algorithm explained in Section 5.1.

As a result of the communication asymmetry, it is not possible to use the same buffers for `EXPAND` and `FOLD` operations and different buffers are used in `EXPAND` and `FOLD` operations for a total of four buffers per processor. Buffers are re-used as much as possible in order to preserve memory efficiency. Pseudocode for this construction step is given in Algorithms 3 and 4. *setup_comm_expand* function used in Algorithm 3 is quite similar to the Algorithm 4, therefore not included.

5.2.3 Parallel SGD iteration Stage

After the data structures are formed, parallel SGD iteration stage starts. In this stage, sub-iterations play an important role. Each processor completes an iteration on its local nonzero matrix by running constant number of sub-iterations. Number of sub-iterations are equal for all processors. After a processor completes

Algorithm 3 Sub-iteration Communication Setup

```
1: Input: s                                   ▷ s is number of sub-iterations
2: for each sub-iteration i do
3:     if rank == 1 then
4:         read partition vector of i
5:     else
6:         allocate memory for partition vector of i
7:     end if
8:     BROADCAST partition vector of s
9:     setup_comm_fold(i)
10:    if i != 0 then
11:        setup_comm_expand(i)
12:        free partvec of subiter i
13:    end if
14: end for          ▷ setup communication between sub-iterations s − 1 and 0
15: setup_comm_expand(0)
16: free partvec of subiter s − 1
17: determine maximum buffer size for to initialize the buffers
18: initialize buffers
```

the local SGD in a sub-iteration, communication stage starts. This stage is completed in two steps:

1. In the first step, non-owner Q-matrix rows that are updated in the current sub-iteration are sent to the owner processor. Owner processors sum the incoming Q-matrix rows according to their global index and take the average, thus finding the updated Q-matrix row.

2. In the second step, owners of the updated Q-matrix rows send the updated rows to the processors that will use these in the next sub-iteration. Non-owner processors receiving these updated Q-matrix rows update their local corresponding Q-matrix row. Therefore, all the latent factor vectors are updated and synchronized among all processors. A Q-matrix row is sent to the next sub-iteration if and only if that Q-matrix row will immediately be used in the next sub-iteration. P-matrix rows are completely local and therefore do not require any communication.

Actually our PSGD-SI algorithm is similar to the GASGD [12] explained in

Algorithm 4 setup_comm_fold

Input: $sz, partvec$ ▷ sz is number of processors
$sendcounts[1 \ldots sz] \leftarrow 0$
$recvcounts[1 \ldots sz] \leftarrow 0$
$nsend, nrecv \leftarrow 0$
for each non-owned column c **do**
    $p \leftarrow partvec[c]$
    $sendcounts[p] \leftarrow sendcounts[p] + 1$
    if sendcounts[p]==1 **then**
        $nsend \leftarrow nsend + 1$
    end if
end for
`ALLTOALL` on $sendcounts$ ▷ data coming from `ALLTOALL` fills $recvcounts$
initialize send data structure
$totalSendCnt \leftarrow 0$
for each processor p **do**
    $totalSendCnt+ = sendcount[p]$
    if $recvcounts[p]! = 0$ **then**
        $nrecv \leftarrow nrecv + 1$
    end if
    fill the send data structure accordingly
end for
create $recv$ data structure
for each processor p **do**
    $totalRecvCnt+ = recvcount[p]$
    fill the recv data structure accordingly
end for
$i \leftarrow 0$
for each non-owned column c **do**
    $sendinds[i] = c$
    $i \leftarrow i + 1$
end for
for each processor p in recvlist **do**
    `IRECV` on recv indices of p
end for
for each processor p in sendlist **do**
    `SEND` on send indices of p
end for
`WAITALL` for `SEND` calls
localize receive indices
localize send indices
`free` local data structures

Section 3.4. However, there are three main differences between the two algorithms. First of all, We use the PaToH hypergraph partitioning algorithm to partition the nonzeros to the processors against GASGD's greedy graph partitioning. Secondly, unlike GASGD, not all latent factors participate to the communication in our algorithm. Thirdly, the owner processors change after each sub-iteration in our algorithm. In GASGD, master processors (which we call owner processors) do not change throughout the iteration.

Chapter 6

Experiments and Results

6.1 Datasets

We use four different datasets for our experiments. The datasets are: movielens_20m, netflix, yahoo_music_1 and yahoo_music_2. These datasets are taken from several open source publications [3, 13, 14]. The matrices are formatted in the Matrix Market Exchange Format [15]. Several relevant statistics for these datasets are provided in Table 6.1, namely number of rows (m), number of columns (n), number of nonzeros ($|R\ |$) and density (d). Density of a matrix is defined as:

$$d = \frac{|R\ |}{m \times n} \tag{6.1}$$

Dataset	Rows(m)	Columns (n)	Nonzeros ($|R\ |$)	Density(d)
movielens_20m	138,493	26,744	20,000,263	5.3e-03
netflix	480,189	17,770	100,480,507	1.1e-02
yahoo_music_1	249,012	296,111	61,944,406	8.4e-04
yahoo_music_2	1,000,990	624,961	252,800,275	4.0e-04

Table 6.1: Statistics for the datasets

6.2 Experiment Details

The algorithm used in the following experiments are implemented by using the C programming language and OpenMPI message passing library [16]. We conduct our experiments on a high performance computing system equipped with 48 Intel Skylake Xeon Platinum 8174 processors with 33MB cache. There are 8 islands and 792 nodes per island for a total of 6,336 nodes in the system as well as 96GB of RAM per node. Intra-island topology is non-blocking fat tree.

Experiment hyperparameters for the datasets are given in Table 6.2. Latent factor size f is fixed to 50. Test-train split for the dataset is 90%-10%. Latent factors are initialized according to Glorot and Bengio's [17] weight initialization method commonly known as Xavier initialization:

$$W_{ij} = U \sim \left[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}} \right], \tag{6.2}$$

where $U \sim (-x, x)$ is uniform distribution on the interval $(-x, x)$ and f is the latent factor size.

Dataset	Learning Rate ($alpha$)	Decay (β)
movielens_20m	0.015	0.05
netflix	0.015	0.05
yahoo_music_1	0.00075	1.0
yahoo_music_2	0.00075	1.0

Table 6.2: Statistics for the datasets

6.3 Results

In order to quantify the performance of our algorithm we are using staleness and communication volume metrics described in Chapter 4 as well as test and train RMSE values. For the staleness and communication volume metrics, the naive algorithm described in Section 5.1 is selected to be used as the baseline algorithm.

Table 6.3 gives the performance metrics for 4 different data sets for 512 processors. These results show that as the number of sub-iterations increase, staleness and communication volume decrease. These empirical results correlate with the observations in Chapter 4. Furthermore, staleness decrease as much as 95% for Yahoo music 2 dataset with 512 sub-iterations and communication volume decreases up to 47% for 512 sub-iterations.

Interestingly, the change in RMSE is concave and the best RMSE generally lies in the middle. According to the statistics in Table 6.3, test RMSE can decrease up to 61%. On average, there is a 51% improvement among all datasets when sub-iterations are used. In fact, even 2 sub-iterations improve the RMSE values significantly, up to 56% for Netflix ratings dataset. To explain this result, Figure 6.1 presents the histograms for the ϕ_i values of the columns for each dataset. In 3 of the 4 datasets, columns tend to have lower connectivity. Using 2 sub-iterations works well since $\phi_i \approx 2$ for most q_i. Hence, decrease in staleness of lower connectivity columns affect overall performance significantly.

In Figure 6.2, plots of train RMSE values vs number of iterations are presented. The plots show that using PSGD-SI beats the naive algorithm along the whole curve even for two sub-iterations. Although different number of sub-iterations yields similar results, using the PSGD-SI algorithm gives significantly better curves compared to the naive algorithm.

Figure 6.3 presents the variation of cost and test RMSE values of the PSGD-SI algorithm with different number of processors. Cost is defined as $c = p \times t$ where p is the number of processors and t is the runtime of parallel SGD step. Number of sub-iterations for each different run is $s = p/4$. The upper two plots for Netflix and Movielens datasets in Figure 6.3 show that increasing number of processors yield similar RMSE values. Contrarily, the bottom two plots for Yahoo datasets perform better as the processor number decreases. Considering the fact that Netlifx and Movielens datasets are denser than the Yahoo datasets, our algorithm can be described as serializable for dense datasets.

Figure 6.4 shows the test RMSE values of the algorithm as the wall clock time

increases for different number of processors, in which $s = p/4$. For the Netflix and Movielens datasets, 1024 processors give the best test RMSE. Furthermore, 512 and 128 processors give the best results for Yahoo Music 1 and Yahoo Music 2 libraries respectively. The reason for the decrease in performance to Yahoo Music datasets on higher number of processors can be using $s = p/4$ as number of sub-iterations. An optimal sub-iteration count for $p = 1024$ might be found which can show that $p = 1024$ also works well for Yahoo Music datasets. Therefore, increasing the number of processors yields better results up to an optimal processor count. Again, Figure 6.4 shows that our algorithm serializes well.

Figure 6.5 shows the number of updates completed per second for different number of processors. Table 6.4 presents the statistics for the same experiment. Again, the number of sub-iterations for each different run is $s = p/4$. The figure shows that for all the datasets, the number of updates completed in the same duration increases with the number of processors. Hence, our algorithm scales well as the number of processors increases. According to Table 6.4, average staleness and average communication volume per processor decrease as number of processors increases. Furthermore, compared to $s = 1$, using sub-iterations decreases staleness and communication volume for all values of p, as the normalized statistics show. Therefore, our experimental results correlate with our theoretical expectations.

Dataset Name	sub iterations	staleness	normalized staleness	communication volume	normalized comm. vol.	train RMSE	Test RMSE
movielens Ratings 20m	1	2,075,707	1.000	4,151,414	1.000	0.875	1.564
	2	2,053,078	0.989	4,151,414	1.000	0.741	1.136
	4	2,013,661	0.970	4,111,997	0.991	0.735	0.953
	8	1,946,787	0.938	4,045,123	0.974	0.733	0.843
	16	1,834,458	0.884	3,932,794	0.947	0.733	0.789
	32	1,645,530	0.793	3,743,866	0.902	0.735	0.762
	64	1,343,124	0.647	3,441,460	0.829	0.737	0.750
	128	897,579	0.432	2,995,915	0.722	0.741	0.747
	256	360,738	0.174	2,459,074	0.592	0.747	0.750
Netflix Ratings	1	4,349,691	1.000	8,699,382	1.000	0.776	1.640
	2	4,331,921	0.996	8,699,382	1.000	0.624	0.733
	4	4,296,383	0.988	8,663,844	0.996	0.614	0.682
	8	4,225,321	0.971	8,592,782	0.988	0.612	0.673
	16	4,083,216	0.939	8,450,677	0.971	0.615	0.643
	32	3,799,824	0.874	8,167,285	0.939	0.620	0.639
	64	3,261,296	0.750	7,628,757	0.877	0.626	0.636
	128	2,418,606	0.556	6,786,067	0.780	0.632	0.639
	256	1,254,169	0.288	5,621,630	0.646	0.638	0.642
Yahoo Music Ratings 1	1	51,260,639	1.000	102,521,278	1.000	28.523	39.816
	2	50,635,678	0.988	102,521,278	1.000	24.458	25.070
	4	49,385,758	0.963	101,271,358	0.988	23.844	24.247
	8	46,886,698	0.915	98,772,298	0.963	21.901	22.364
	16	41,943,335	0.818	93,828,935	0.915	21.969	22.346
	32	33,818,867	0.660	85,704,467	0.836	22.911	23.157
	64	23,537,306	0.459	75,422,906	0.736	24.224	24.373
	128	12,711,884	0.248	64,597,484	0.630	23.175	23.404
	256	3,726,632	0.073	55,612,232	0.542	24.256	24.422
Yahoo Music Ratings 2	1	19,269,421	1.000	38,538,842	1.000	27.573	39.680
	2	18,973,310	0.985	38,538,842	1.000	21.972	23.265
	4	18,381,088	0.954	37,946,620	0.985	19.447	21.131
	8	17,196,747	0.892	36,762,279	0.954	19.412	20.887
	16	14,841,859	0.770	34,407,391	0.893	22.335	22.933
	32	11,213,933	0.582	30,779,465	0.799	26.036	26.222
	64	7,306,897	0.379	26,872,429	0.697	21.009	21.713
	128	3,670,674	0.190	23,236,206	0.603	22.120	22.673
	256	1,008,908	0.052	20,574,440	0.534	23.097	23.570

Table 6.3: Statistics for the naive and PSGD-SI algorithm run on 512 processors

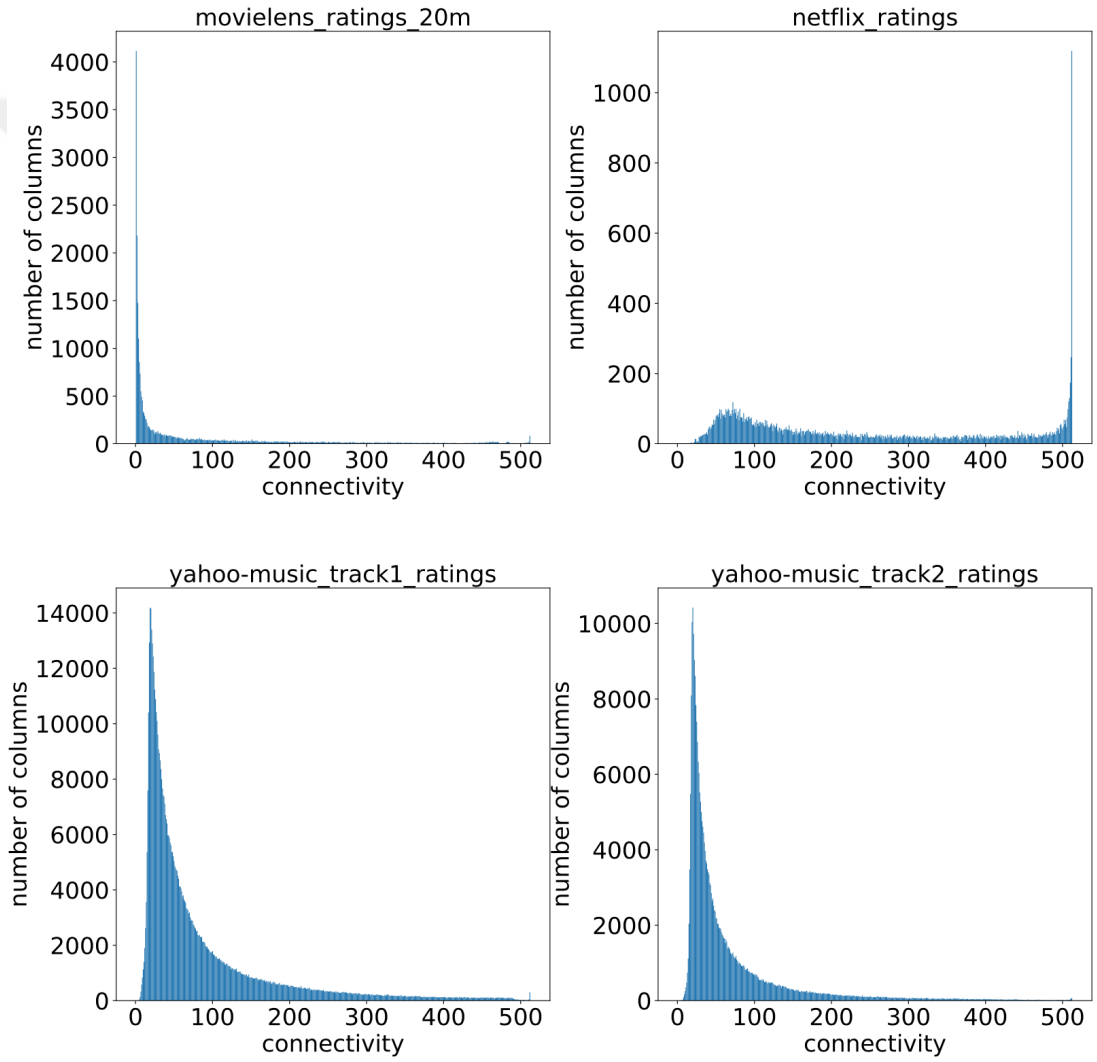

Figure 6.1: Connectivity histograms for the datasets

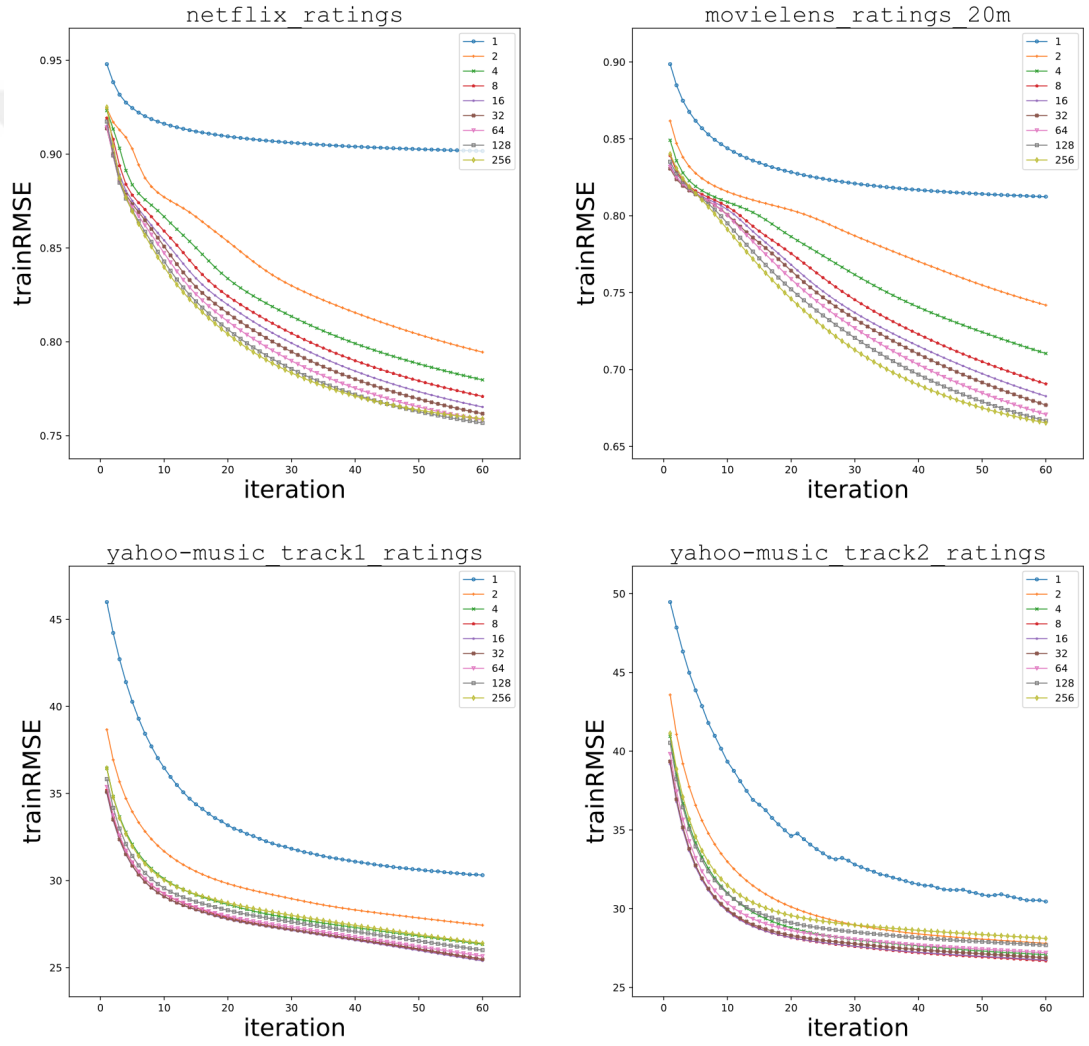

Figure 6.2: Number of iterations vs train RMSE for 4 different datasets, k=512

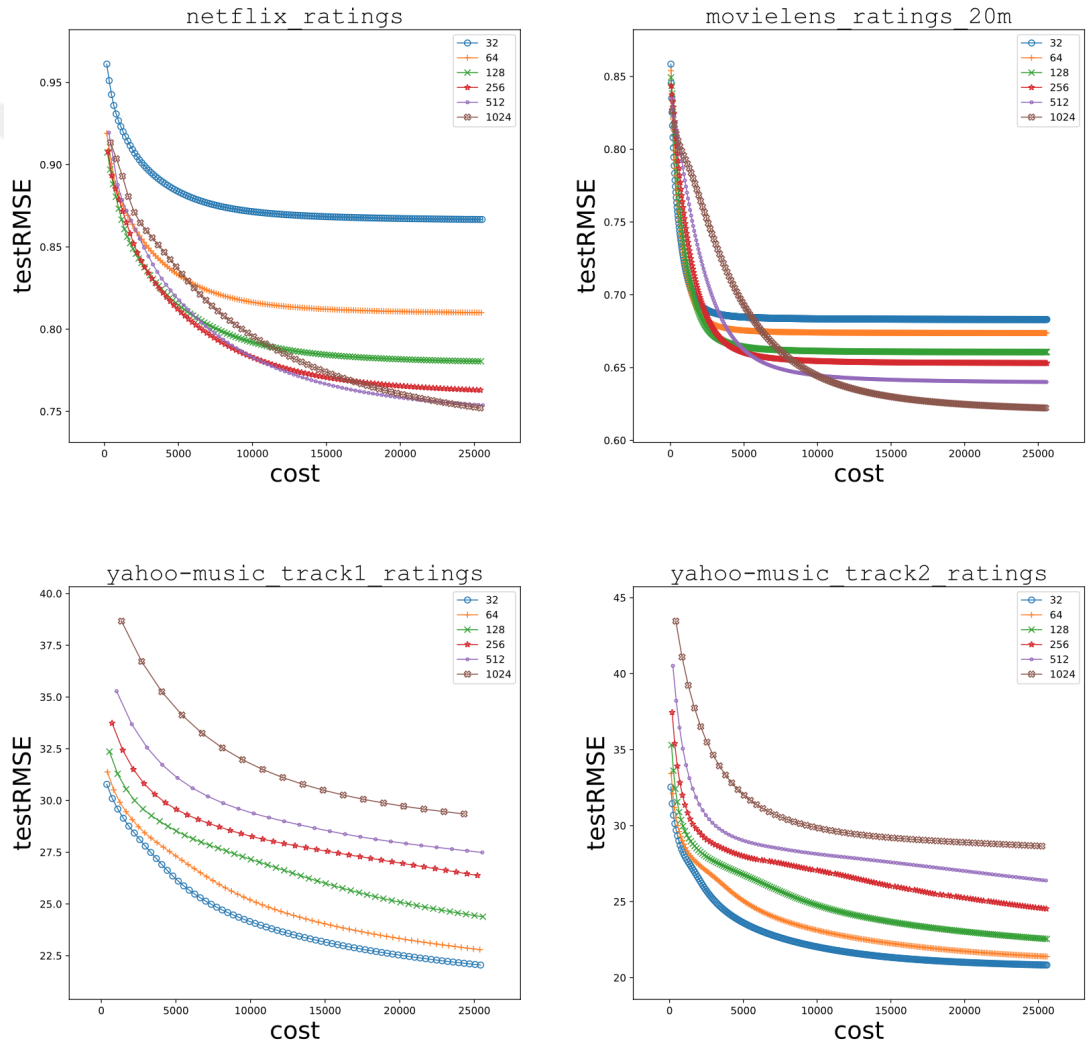

Figure 6.3: Cost (number of processors $\times$ time) vs test RMSE for different k values for 4 different datasets

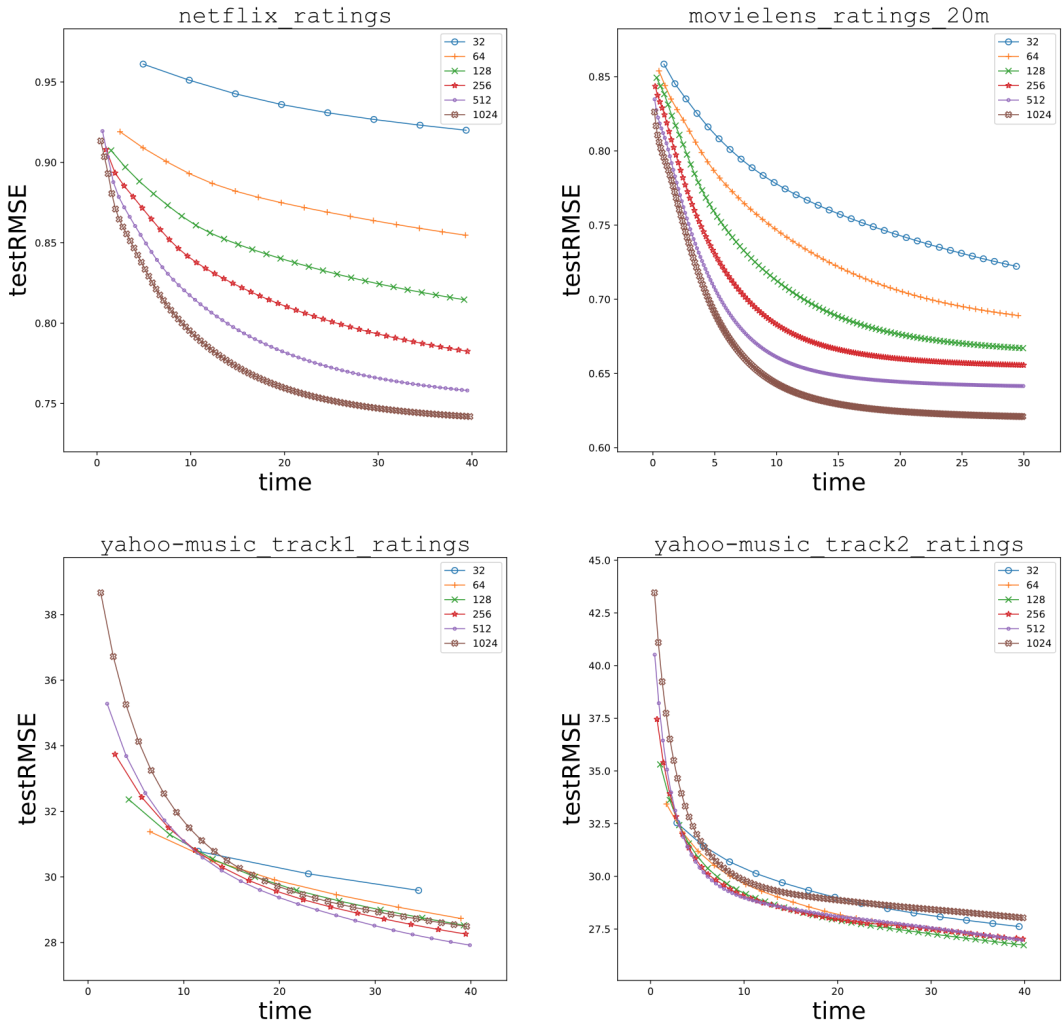

Figure 6.4: Wall clock time vs test RMSE for 4 different datasets

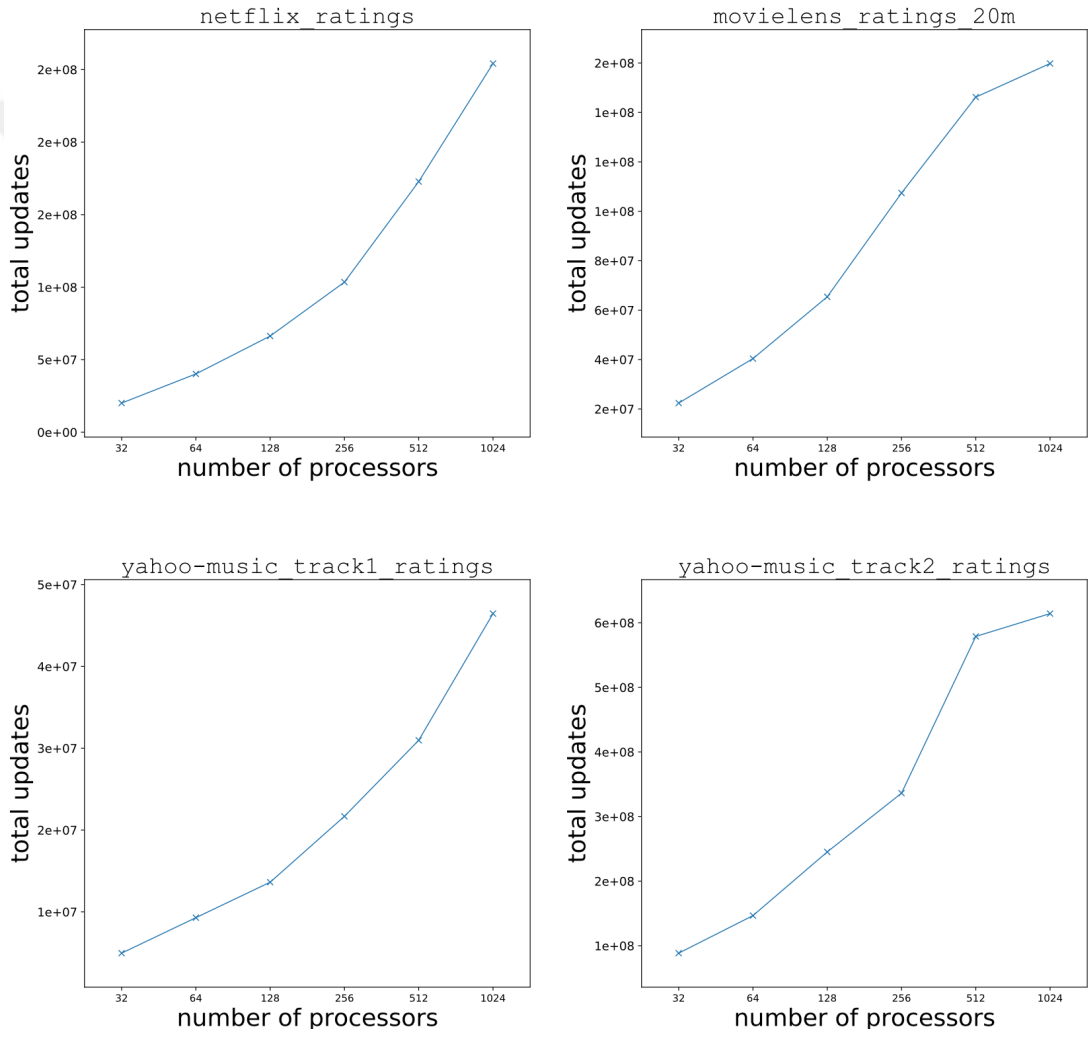

Figure 6.5: Number of processors vs number of updates completed per second for 4 different datasets

Dataset Name	number of processors	avg. staleness	norm. staleness	avg. comm. volume	norm. comm. volume	avg. comm. msg	norm. comm. msg	avg. SGD iteration time(ms)
movielens Ratings20m	32	4,151	0.551	12,345	0.819	333	5.369	886
	64	3,432	0.510	10,499	0.780	1,128	8.952	490
	128	2,894	0.489	8,986	0.759	2,997	11.801	303
	256	2,282	0.460	7,331	0.739	4,167	8.205	184
	512	1,753	0.432	5,851	0.722	3,925	3.956	135
	1,024	1,293	0.404	4,515	0.706	3,328	1.766	125
Netflix Ratings	32	9,310	0.706	23,048	0.874	320	5.167	4,776
	64	8,147	0.664	20,699	0.843	1,164	9.238	2,384
	128	7,078	0.627	18,501	0.820	3,695	14.548	1,451
	256	5,833	0.588	15,819	0.798	6,953	13.633	919
	512	4,724	0.556	13,254	0.780	8,261	8.083	549
	1,024	3,699	0.527	10,736	0.765	8,177	3.998	389
Yahoo Music Ratings1	32	195,810	0.593	545,814	0.826	356	5.734	10,741
	64	125,896	0.478	398,941	0.758	1,391	11.039	6,022
	128	76,679	0.384	281,400	0.704	5,350	21.064	3,912
	256	44,425	0.307	191,482	0.662	18,862	36.985	2,471
	512	24,828	0.248	126,167	0.630	36,938	36.182	1,669
	1,024	13,242	0.200	80,205	0.604	31,133	15.499	1,298
Yahoo Music Ratings2	32	86,793	0.575	247,116	0.818	363	5.858	2,766
	64	47,246	0.420	164,274	0.731	1,427	11.323	1,659
	128	26,217	0.319	110,714	0.674	5,368	21.132	999
	256	13,558	0.241	71,032	0.631	14,540	28.509	725
	512	7,169	0.190	45,383	0.603	15,759	15.441	418
	1,024	3,570	0.149	27,760	0.581	11,358	5.666	408

Table 6.4: Statistics for the PSGD-SI algorithm run on different number of processors, $s = p/4$. Statistics are normalized to $s = 1$

Chapter 7

Conclusion and Future Work

We worked on the SGD parallelization problem on distributed memory. After reviewing previous implementations in the literature, we identified the stale data problem and communication cost. We pointed out that decreasing stale data usage and communication cost can improve parallel SGD performance. Then, we introduced a new SGD parallelization algorithm called PSGD-SI. In short, PSGD-SI introduces sub-iteration synchronizations in order to decrease the stale data usage and improve the performance of SGD in turn. Our experiments show that PSGD-SI algorithm scales up to 512 processors. We show that introduction of sub-iterations can decrease the staleness up to 97% and communication volume up to 47%. According to our experiments, even using only 2 sub-iterations can improve test RMSE as much as 56%. In best case, utilizing sub-iterations improve the test RMSE by up to 60%.

As future work, improving the efficiency of the local SGD step can be a suitable target. On the implementation level, a low-level implementation of the dot-product inside the SGD can decrease the local SGD execution time.

Moreover, local SGD computation can be divided into two parts such that non-local nonzeros are processed and then sent first, then local latent factors are computed. Hence, communication latency is overlapped with local latent factor

computation. This improvement can decrease the overall iteration runtime.

Bibliography

[1] R. M. Bell, Y. Koren, and C. Volinsky, "All together now: A perspective on the netflix prize," *CHANCE*, vol. 23, no. 1, pp. 24–29, 2010.

[2] R. M. Bell and Y. Koren, "Lessons from the netflix prize challenge," *Acm Sigkdd Explorations Newsletter*, vol. 9, no. 2, pp. 75–79, 2007.

[3] J. Bennett, S. Lanning, *et al.*, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007, p. 35, New York, NY, USA., 2007.

[4] B. Hallinan and T. Striphas, "Recommended for you: The netflix prize and the production of algorithmic culture," *New Media & Society*, vol. 18, no. 1, pp. 117–137, 2016.

[5] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[6] U. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.

[7] M. O. Karsavuran, K. Akbudak, and C. Aykanat, "Locality-aware parallel sparse matrix-vector and matrix-transpose-vector multiplication on many-core processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, pp. 1713–1726, 2016.

[8] "PaToH manual." `https://www.cc.gatech.edu/~umit/PaToH/manual.pdf`. [Online; accessed 28-January-2022].

[9] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, (New York, NY, USA), p. 69–77, Association for Computing Machinery, 2011.

[10] H. Yun, H.-F. Yu, C.-J. Hsieh, S. V. N. Vishwanathan, and I. Dhillon, "Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion," *Proc. VLDB Endow.*, vol. 7, p. 975–986, jul 2014.

[11] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. a. Ranzato, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[12] F. Petroni and L. Querzoni, "Gasgd: Stochastic gradient descent for distributed asynchronous matrix completion via graph partitioning.," in *Proceedings of the 8th ACM Conference on Recommender Systems*, RecSys '14, (New York, NY, USA), p. 241–248, Association for Computing Machinery, 2014.

[13] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The yahoo! music dataset and kdd-cup'11," in *Proceedings of KDD Cup 2011*, pp. 3–18, PMLR, 2012.

[14] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *Acm transactions on interactive intelligent systems (tiis)*, vol. 5, no. 4, pp. 1–19, 2015.

[15] "Matrix Market Formats." `https://math.nist.gov/MatrixMarket/formats.html`. [Online; accessed 27-January-2022].

[16] "OpenMPI homepage." `https://www.open-mpi.org`. [Online; accessed 27-January-2022].

[17] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Y. W. Teh and M. Titterington, eds.), vol. 9 of *Proceedings of Machine Learning Research*, (Chia Laguna Resort, Sardinia, Italy), pp. 249–256, PMLR, 13–15 May 2010.