修士論文

マルチコア共有メモリ環境における行列補完アルゴリズムのタスク並列化

Scalable Task-Parallel SGD on Matrix Factorization in Multicore Architectures

平成 27 年 2 月 5 日提出

指導教員　田浦 健次朗 准教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-136426 西岡 祐輔

概 要

レコメンドとは、オンライン通販サービスなどでユーザの好みにあったアイテムを推定、推薦する技術であり、近年大きな注目を集めている。レコメンドのアルゴリズムの１つに行列補完がある。行列補完ではユーザ同士、あるいはアイテム同士の類似度とユーザが他のアイテムに与えたレーティングからユーザが興味を持ちそうなアイテムを推定する。

行列補完に stochastic gradient descent（SGD）法というオンライン最適化手法を適用することができる。SGD は１つのサンプルを処理するたびに学習モデルの更新をおこない、省メモリかつ高い収束速度を有することから大規模データの学習に特に効果的であるとされるが、その逐次的性質から並列化が難しいという欠点がある。

行列補完に SGD を適用し、さらに並列化させようという研究はいくつか既に存在し、中でも FPSGD [37] は state-of-the-art な手法である。FPSGD では入力行列を多数のブロックに分割し、それらをワーカに割り当てる。この際、同じ行あるいは同じ列を共有したブロックは複数スレッドに同時に割り当てないようにすることで更新の競合を防ぐことができ、ワーカは独立にブロックを処理できる。このブロックのスケジューリング方針により FPSGD は既存研究よりも高い収束性能を示している。しかし、我々が FPSGD のスケーラビリティを計測したところ、ワーカ数が多いとスケーラビリティが著しく損なわれることがわかった。

本研究では、マルチコア共有メモリ環境において、SGD を行列補完に適用しタスク並列モデルを使用して並列化させた新しい手法を提案する。結果として、FPSGD のスケーラビリティを大幅に上回る結果を示し、同一コア使用時でも高い収束性能を示すことが確かめられた。

目 次

図 目 次

表 目 次

第1章　序論

1.1　背景

インターネットの世界的な普及により、Amazon や楽天市場などの通販サービス、Netflix や Hulu などの動画配信サービス、iTunes や Spotify などの音楽配信サービスなどから商品をレンタルあるいは購入することができるようになった。これらの Web サービスには、これまでユーザが実際に購入した商品や高い評価をつけた商品と関連性の強い商品をレコメンドする仕組みがある。これにより、非常に多くのアイテムの中からユーザが自身の興味を持つ可能性の高いものを見つけることが容易になり、上に挙げたような通販サービスでは必須の技術となっている。

Netflix は 2009 年にレコメンデーションアルゴリズムの優劣を競う「Netflix Prize」というコンペティションを開催し、様々なアルゴリズムが考案された。このことからも、レコメンデーションという技術に対する関心の高さが伺える。

レコメンデーションの手法は、大きく content filtering と collaborative filtering の 2 つに分類される。content filtering とは、例えば映画をユーザにレコメンドする場合ではその映画のジャンル、出演俳優など、アイテムの内容に関する情報を用いる手法のことである。一方、collaborative filtering では、アイテムの内容は見ることなく、ユーザ間の類似度、あるいはアイテム間の類似度を計算する。つまり、「同じアイテムを購入したユーザは相関性が高い」「同じユーザから購入されたアイテムは相関性が高い」と予想してレコメンドをおこなう。商品の内容に関する情報を用いることないため、簡単かつ高精度な結果を得ることができることから、collborative filtering に注目した研究が数多く行われている。

collaborative filtering を用いたアルゴリズムの 1 つとして、行列補完アルゴリズム (matrix factorization) がある。行列補完では、ユーザを行、アイテムを列、ユーザがアイテムにつけたレーティングの値を要素にとった行列計算に落とし込み、入力行列を 2 つの低ランク行列の積として近似、欠損値をもとめる。最適化には stochastic gradient descent(SGD) 法が収束の速さ、メモリ消費量の観点から有効であることが知られているが、SGD は 1 つのサンプルを処理するごとにモデルの更新をおこなうというその逐次的な性質から、並列化が難しいという問題がある。

SGD を用いた行列補完の並列化を提案した研究として、FPSGD[37] がある。FPSGD では、入力行列を多数のブロックに区切り、スレッドにブロック単位で処理を任せる。その際、同じ行あるいは列を共有しないようにブロック更新をおこなうことで並列化によるデータの競合を防ぐことができる。しかし、スレッドに新しいブロックを割り当てる際に該当する行と列が他のスレッドによってアクセスされないようにフラグをたてるなど共有変数への書き込み操作が必要となるため、スレッドはロックを取得する必要がある。スレッドが増えるとともにロックにかかる時間が増大するため、使用スレッド数が多いとスケーラビリティは著しく悪化する。

また、スレッドへのブロックの割り当ては完全にランダムにおこなわれるため、前のブロックと行と列を共有していないブロックが割り当てられることでメモリからキャッシュへのデータの

入れ替えが必要となる。そのためキャッシュの再利用性が低く、処理時間の増加を引き起こす可能性がある。

1.2　本研究の目的と貢献

本研究では、マルチコアの共有メモリ環境において、最適化手法として SGD を用いた行列補完アルゴリズムにおいて、分割統治法とタスク並列モデルによる並列化手法を提案する。提案手法では、分割統治法により入力行列を再帰的に分割し、ワーカ間での負荷分散をおこなう。その際、再帰関数内で同じ行、同じ列にあるブロックは同時には処理しないことで、複数のワーカによる更新の競合を起こすことなく並列化できる。タスク並列ではタスクの割り当ては処理系任せであるため、FPSGD で用いられているロックのような中央集権的なデータ構造を持つ必要がなく、ロック待ちにかかる時間を排除できる。また、タスク並列によりデータのローカリティを意識したタスク割り当てが行われることで、キャッシュミスも削減することができる。

本研究では、我々の先行研究にあたる FPSGD についてオープンソースの実装[1]を用いて実験をおこない、スケーラビリティが悪化する原因などを分析した。評価実験では、特性の異なる複数のデータセットを用いて FPSGD に比べスケーラビリティが優れていることを確認した。また、いくつかのハイパーパラメータによる性能への影響についても調査した。

1.3　本稿の構成

本稿の構成は以下の通りである。

2 章 レコメンドの概要、content-based filtering と collaborative filtering という 2 つの手法について簡単に説明する

3 章 collaborative filtering に属する neighborhood-based なアプローチ、latent-model-based なアプローチ、content-based と組み合わせた hybrid なアプローチの 3 つについて、具体的な手法を紹介する。

4 章 collaborative filtering の latent-model-based なアプローチの中から、本研究でも用いる行列補完アルゴリズムについて説明する。

5 章 最適化アルゴリズムとして SGD を適用した行列補完の並列化を試みた研究として、Hog-Wild! [7]、DSGD [26]、FPSGD [37] の 3 つを紹介する。

6 章 5 章で紹介した FPSGD という手法について、スケーラビリティが阻害されている原因を説明する。

7 章 本研究で提案する、分割統治法とタスク並列モデルをを用いた行列補完の並列化について述べる。

8 章 FPSGD と我々の提案手法で比較・性能評価をおこなう。

9 章 まとめと今後の展望について述べる。

[1] `http://www.csie.ntu.edu.tw/~cjlin/libmf/`

1.4　用語の定義

本稿において使用する用語についてあらかじめ定義する。レコメンドする映画、本、音楽などの商品を「アイテム」、アイテムをレコメンドする相手であるサービスの利用者を「ユーザ」、ユーザがアイテムに対して付与する評価情報を「レーティング」と呼ぶこととする。

第2章　レコメンド

図 2.1. Amazon におけるレコメンドの図

レコメンドは我々一般のインターネットユーザにも非常に身近な技術である。図 2.1 はオンライン通販サービス大手の Amazon の商品詳細ページにて推薦された商品である。Amazon には商品に対する評価をつけるために「星」をつける仕組みがある。星は 1〜5 つつけることができ、星 5 つはユーザがそのアイテムを非常に好き、あるいは気に入ったということを示し、星 1 つは質が低い、問題があるなどなんらかの理由で嫌い、あるいは気に入っていないということを示す。Amazon は星の評価やユーザの購入履歴、閲覧履歴などを総合的に判断し、図 2.1 のような形でアイテムを推薦している。

レコメンド手法には、大きく分けて

- content-based filtering
- collaborative filtering

の 2 種類がある。以下ではこれらの 2 つの手法について具体例を述べながら簡単に紹介する。

2.1　content-based filtering

2.1.1　概要

content-based filtering[24] とは、ユーザやアイテムの属性からレコメンドをおこなう手法である。アイテムとして映画を例にとると、ジャンル、出演している俳優、興行成績などがその映画を特徴付ける属性として挙げられ、同じジャンルである、あるいは同じ俳優が出演しているといった情報から映画を推薦することができる。また、例えばオンライン動画配信サービスなどに登録しているユーザはサインアップ時にどのジャンルの映画が好きかを明示的に登録していることが

あり、ユーザの属性からもアイテムを推薦することが可能である。content-based filtering の欠点として、ユーザやアイテムに関する外部の情報を手に入れる手間が考えられる。

2.1.2　例: Music Genome Project

インターネットラジオサービスを提供する Pandora では、Music Genome Project[2] と呼ばれるプロジェクトを立ち上げている。このプロジェクトでは、ミュージシャンや音楽の専門家たちがジャンルやテンポ、使用されている楽器など、その数なんと 450 種類にも及ぶ属性を楽曲に付与し、ユーザの好みに応じて再生される音楽を最適化しようとする試みである。このプロジェクトは楽曲に関する属性を元に推薦をおこなうことから content-based filtering の一例であると言える。一曲ずつ人手で属性付与をおこなっているため Pandora では機械を用いた最適化はおこなわないとしており、これらの属性付けを 1 曲ずつ全て人手でおこなっているため、尋常ではない手間がかかっていると考えられる。

2.2　collaborative filtering

collaborative filtering(協調フィルタリング) [8, 25, 30, 3, 32] は、ユーザの購買履歴や閲覧履歴、アイテムにつけたレーティングの点数などからアイテムを推薦する手法である。

	プライベートライアン	パイレーツオブカリビアン	美女と野獣	硫黄島からの手紙
ユーザA	5		2	4
ユーザB		4	4	
ユーザC	5	1		(1)
ユーザD	3	5	(2)	

図 2.2. 協調フィルタリングの模式図

ここで、図 2.2 に示されるテーブルを用いて協調フィルタリングを説明する。このテーブルの行はユーザを、列はアイテムを、各エントリはユーザがそのアイテムに与えたレーティングを表している。レーティングとは先の Amazon の例で述べたような星の数と同義である。

ここで、“プライベートライアン” という映画に着目してみると、ユーザ A とユーザ C の 2 人がこの映画を高く評価していることがわかる。つまり、ユーザ A とユーザ C は映画の好みが似ている可能性が高いということを示している。ユーザ A に着目してみると、このユーザは “プライベートライアン” の他に “硫黄島からの手紙” に対しても高いレーティングをつけている。ユーザ A とユーザ C の好みが似ているのであれば、ユーザ C に “硫黄島からの手紙” をレコメンドしても気にいる可能性が高い（図の (1)）。事実、これら 2 つの映画は 2 つとも戦争映画のジャンルにあてはまるので、この推定はあながち的外れではないということになる。また同様に、ユーザ B、ユーザ C ともに “パイレーツ・オブ・カリビアン” を高く評価しており、ユーザ B はそれに加え

て “美女と野獣” も評価しているので、ユーザ C に “美女と野獣” をレコメンドしてはどうか（図の (2)）となる。

collaborative filtering は通常、このようにユーザ同士、アイテム同時の相関関係をレーティングの値をもとに分析することでおこなわれる。前述した content-based filtering よりも高精度に推薦をおこなえることが経験的に知られており、ユーザやアイテムに関する外部情報を手に入れる必要もないことから、レコメンド手法として人気を得ている。

collaborative filtering の特徴やその具体的な手法については章を分けて説明することとする。

第3章　協調フィルタリング

collaborative filtering（CF）という単語は、Tapestry [8] というメールやネットニュースのフィルタリングをおこなうシステムの開発者らが彼らの論文中で提案したのが始まりである。それ以前のフィルタリングシステムは、届いたメールやネットニュースを逐一精査してユーザの興味のありそうな単語を登録しておくことでフィルタリングをする、いわゆる content-based なアプローチであった。このアプローチでは単語が含まれているだけの全く関係ない文書もフィルタを通り抜けてしまうため、精度が大きな問題となっていた。そこで Tapestry では、ユーザは自分と同じような興味を持っている人をあらかじめ登録しておき、届いたメールやニュースに対して annotation をつける（メールに返信する、ニュースにコメントする等といった行為を指す）。システムは、似た興味を持つユーザは同じアイテムに対して似た評価をつけるという前提に基づき、その annotation の情報を蓄積してフィルタリングに活かす。この様子がまるで複数の人々がフィルタリングのために “協力” し合っているように見えることから collaborative filtering という名前がつけられた。

つまり、collaborative filtering は、あるユーザ X と Y が n 個のアイテムに対して同じように評価している（そのアイテムを買う、閲覧するなどの別の行為でもよい）場合、他のアイテムに対しても彼らは似た様な評価をするだろうという前提に基づいている。

3.1　特徴と課題

collaborative filtering はその誕生以来、様々なサービスに応用されてきた。それにより、collaborative filtering の持つ特徴やレコメンドをおこなう上での課題が特定されてきた。本項では、collaborative filtering に基づいてレコメンドをおこなう際の特徴と課題について説明する。なお、この分類は collaborative filtering のサーベイ論文 [32] を参考に記述している。

3.1.1　データのスパース性

実際にレコメンドをおこなう際に用いられる入力データは、レーティングなどの評価情報は非常に少ない、巨大な疎行列になりがちである。このスパース性は多くの問題を引き起こすが、その 1 つが “cold start 問題” [34] である。これは新しいユーザや新しいアイテムが入力データとして入ってきたときに、新しいユーザはまだ何のアイテムも評価しておらず新しいアイテムはまだどのユーザからも評価されていないので、十分な情報がないためにレコメンドができないという問題である。また、本来好みが似通っているはずのユーザなのに、同じアイテムにレーティングを付与していないためにレコメンドエンジンはそれを認識できないという “neighbor transivity 問題” も引き起こされる可能性がある。

3.1.2　データ量の増加

入力データのユーザとアイテムの数の増加により、現実的な時間で応答を返すことが困難となる。レコメンドするアイテムの情報を取得するために Web サイトの描画が遅れてしまってはユーザエクスペリエンスを損ねてしまう。そのため、レコメンドエンジンはユーザ・アイテムの増加にも対応できるスケーラビリティを備えていなければならない [17]。

3.1.3　synonymy 問題と polynomy 問題

アイテムの数が増えてくるにつれ、複数のアイテムがデータベース上には存在するものの意味として指すものは同じであるという問題が生じる。例えば、中身は同じ「ハリーポッター」の本であるにも関わらず「ハリーポッター」と日本語で表記されているか「Harry Potter」と英語で表記されているかで異なるアイテムとして扱われているような場合である。これは synonymy 問題と呼ばれ、同じようなアイテムがユーザに推薦されてしまうなど、レコメンドの精度が低下するという問題を引き起こす。

また、複数の意味を持つはずのアイテムが 1 つにまとめられてしまうという問題も生じる。例えば、「ringo」という単語がデータベースにアイテムの 1 つとして登録されていたとする。この「ringo」は果物の「りんご」のことかもしれないし、歌手の「椎名林檎」のことかもしれないし、はたまた海外のミュージシャンの「リンゴ・スター」かもしれない。これは polynomy 問題と呼ばれ、レコメンドの正確性という意味で悪影響を与えてしまう。

3.1.4　gray sheep 問題

後述する neighborhood-based なアプローチでは、あらかじめ似たユーザ同士をグループ分けしておき、そのグループに属する他のユーザの評価情報を活用してレコメンドをおこなうが、ユーザの中にはどのユーザグループとも一致しないためにレコメンドの恩恵にあずかれないユーザがいることがある [5]。

3.1.5　shilling attack 問題

基本的にレーティングなどの評価情報は全ユーザがつけることができるため、自分の勤める企業の商品には全て高い評価をつけてライバル企業の商品は低い評価をつけるようなほぼスパマー同然の行為を働く悪質なユーザがいる可能性がある。そのため、レコメンドシステムはそのようなほぼ意味のないレーティングやユーザの影響を受けないものでなければならない。

3.1.6　プライバシー問題

Web サービスを利用するユーザとしては自分がどのアイテムを評価した、購入した、閲覧したというパーソナルな情報は他人に知られたくないので、「このユーザがこの商品を買ったのであなたにもオススメです」というような情報はオープンにするべきではない。その一方で、レコメン

ドしているアイテムはどういった理由で薦められているのかの説明責任を果たすこともユーザの安心感という観点で必要となる。

3.2　手法

collaborative filtering に属する手法の分類は大きく分けて次の 3 つである。

- neighborhood-based CF
- latent-model-based CF
- hybrid CF

本項では、それぞれの詳細と具体的なアルゴリズムについて紹介する。

3.2.1　neighborhood-based CF

neighborhood-based な collaborative filtering ではユーザ-アイテムのデータベースから評価予測をおこなう。ユーザとアイテムのどちらに着目するかによって、neighborhood-based な手法は

- 評価予測のためにユーザに着目するユーザベースのアプローチ [35] と
- アイテムに着目するアイテムベースなアプローチ [17, 29]

の 2 つに分かれる。

ユーザベースのアプローチでは、あるアイテムに複数のユーザが高い評価を与えていればこれらのユーザは似ているとシステムが判断し、片方のユーザが高い評価をつけている別のアイテムをもう片方のユーザに推薦する。アイテムベースのアプローチでは、あるユーザが複数のアイテムに対して高い評価をつけているとすると、システムがこれらのアイテムを似ていると判断し、このうちの片方のアイテムにしか評価をつけていないユーザに対しもう片方のアイテムを推薦する。Amazon で用いられているレコメンドエンジンもアイテムベースのアプローチを基にしている [17]。

システムがユーザ同士、アイテム同士を「似ている」と判断すると書いたが、レコメンドにおいてはどれくらい似ているか、つまり類似度を知ることが重要である [29]。類似度の高いアイテムを推薦する、類似度の高いユーザに対して推薦する方がレコメンドの効果がより大きいと考えられるからである。そのため、neighborhood-based な手法では

1. 類似度を計算するステップと、
2. 評価予測をおこなうステップ

の 2 種類がある。以降では、それぞれのステップについて計算式を交えながら具体的な説明をおこなう。

類似度計算

前述した通り、neighborhood-based CF では類似度の計算は非常に重要なステップである。アイテムベースのアプローチの場合、あるユーザが評価したアイテム v_x、v_y の類似度 w_{v_x,v_y} を計算し、ユーザベースのアプローチの場合は、同じアイテムを評価したユーザ u_x、u_y の類似度 w_{u_x,u_y} を計算する。ユーザ間、あるいはアイテム間の類似度を計算する方法はいくつかあるが、ここではピアソン相関に基づいた方法とコサイン類似度の 2 つを紹介する。

- ピアソン相関に基づいた類似度

  ピアソン相関は 2 つの確率変数間における類似度の度合いを測る統計的な指標である [25]。ユーザベースのアプローチにおいては、ユーザ u_x と u_y のピアソン相関は以下の式で表される。

$$w_{u_x,u_y} = \frac{\sum_{i\in I}(r_{u_x,i} - \bar{r}_{u_x})(r_{u_y,i} - \bar{r}_{u_y})}{\sqrt{\sum_{i\in I}(r_{u_x,i} - \bar{r}_{u_x})^2}\sqrt{\sum_{i\in I}(r_{u_y,i} - \bar{r}_{u_y})^2}} \tag{3.1}$$

  ここで、$r_{u_x,i}$ はユーザ u_x がアイテム i に対して与えたレーティング、I は両方のユーザが評価したアイテムの集合、$\bar{r}_{u_x}$ は I に属するアイテムに対してユーザ u_x が与えたレーティングの平均値である。
  一方、アイテムベースなアプローチにおいて、アイテム v_x と v_y の両方に対し評価を与えたユーザの集合 $u \in U$ において、アイテム間のピアソン相関は、

$$w_{v_x,v_y} = \frac{\sum_{u\in U}(r_{u,v_x} - \bar{r}_{v_x})(r_{y,v_y} - \bar{r}_{v_y})}{\sqrt{\sum_{u\in U}(r_{u,v_x} - \bar{r}_{v_x})^2}\sqrt{\sum_{u\in U}(r_{u,v_y} - \bar{r}_{v_y})^2}} \tag{3.2}$$

  で表される。ここで、r_{u,v_x} はユーザ u がアイテム v_x に対して与えたレーティング、U は両方のアイテムを評価したユーザの集合、$\bar{r}_{v_x}$ は U に属するユーザがユーザ v_x が与えたレーティングの平均値である。

- コサイン類似度

  例えば 2 つの文書館の類似度を求める際には、それぞれの文書中に現れる単語の出現頻度をベクトル化しその 2 つのベクトルのコサインを計算する。コサイン類似度の定式化は collaborative filtering にもあてはめることができ、文書をユーザ・アイテム、文書中の単語をレーティングに置き換えることで適用できる [27]。入力データが m ユーザ $\times$ n アイテムの行列とすると、2 つのアイテムの類似度は m 要素のベクトルの積として表される。

$$w_{i,j} = \cos(\boldsymbol{i}, \boldsymbol{j}) = \frac{\boldsymbol{i}\cdot\boldsymbol{j}}{\|vectori\| \times \|/vectorj\|} \tag{3.3}$$

評価予測

neighborhood-based CF では、あるユーザ a に対しレコメンドをする際に、あるユーザがあるアイテムに対してつけるであろう評価値を予測する際には、上記のようにして求めた類似度を重みとして用いる [9]。そこで、ユーザ間あるいはアイテム間の類似度をもとに評価予測をおこなう方法をいくつか紹介する。

- 他のレーティングの重み付き合計

  ユーザベースなアプローチにおいて、ユーザ a がアイテム i につけるであろう評価 $P_{a,i}$ を推定するには、アイテム i についている全レーティングの重み付き平均を計算することで得ることができる [25]。

$$P_{a,i} = \bar{r}_a + \frac{\sum_{u \in U}(r_{u,i} - \bar{r}_u) \cdot w_{a,u}}{\sum_{u \in U} |w_{a,u}|} \tag{3.4}$$

  ここで、$\bar{r}_a$ と $\bar{r}_u$ はそれぞれユーザ a とユーザ u が他の評価済みのアイテムに対して与えたレーティングの平均値であり、$w_{a,u}$ はユーザ a とユーザ u の間の重みである。
- 重み付き平均

  アイテムベースなアプローチにおいて、ユーザ u がアイテム i につけるであろう評価 $P_{u,i}$ を予測するには、単純な重み付き平均を用いることもできる [29]。

$$P_{u,i} = \frac{\sum_{n \in N} r_{u,n} w_{i,n}}{\sum_{n \in N} |w_{i,n}|} \tag{3.5}$$

  ここで、$w_{i,n}$ はアイテム i とアイテム n の間の重みである。

特徴と問題点

neighborhood-based なアプローチの特徴として、類似度の計算は簡単な実装で実現することができ、かつその類似度に基づいたレコメンドの精度が高いという強みがある。様々な派生手法が出てきていることからもわかるようにカスタマイズ性が高く、多種多様なサービスで使用することができることも大きな特徴である。その一方で、類似度の計算は多くのユーザが評価をつけているような一般的なアイテムに基づいておこなわれるため、入力が疎行列でそういった一般的なアイテムが少ない場合は prediction が難しいという問題がある。

3.2.2　latent-model-based CF

latent-model-based な collaborative filtering は、ユーザ-アイテムの行列そのものを評価予測に用いる neighborhood-based とは異なり、機械学習やデータマイニングに用いられるような学習モデルを組み立てる。トレーニングデータによって学習モデルを訓練し、テストデータを与えて評価予測をおこなう。

以下では、latent-model-based な手法についていくつか紹介する。

- ベイジアンネット
- クラスタリング

ベイジアンネット

ここでは、もっともシンプルなアプローチとしてナイーブベイズを用いた手法 [19] を紹介する。この手法では、ユーザ u がアイテム v に対してつけるであろう評価 $\hat{r}_{u,v}$ を予測をするにあたり、レーティングの値をクラスラベルとする。ユーザ u がアイテム i に対して与えたレーティング $class_i \in 1,2,3,4,5, (0 < i < n)$ として、ユーザが他のアイテムに対して $class_i$ のレーティングを与えた時にアイテム v に与えるであろう条件付き確率 $P(class_v|class_1, class_2, \cdots, class_{(v-1)}, class_{(v+1)}, \cdots, class_n)$ を求める。

$$\begin{aligned} class &= \underset{class_v \in 1,2,3,4,5}{\arg\max} P(class_v|class_1, class_2, \cdots, class_{(v-1)}, class_{(v+1)}, \cdots, class_n) \\ &= \underset{class_v \in 1,2,3,4,5}{\arg\max} P(class_v)P(class_1, class_2, \cdots, class_n) \\ &= \underset{class_v \in 1,2,3,4,5}{\arg\max} P(class_v)\prod_{i=0}^{n} P(class_v|P(class_i)) \end{aligned} \tag{3.6}$$

ここで、 式 3.6 への変換において $class_i$ の独立性を仮定していることがナイーブベイズたる所以である。つまり、ユーザ u がアイテムにつけたレーティングがお互いに関係し得ないという事を仮定しているということだが、現実にはあるジャンルのアイテムを高く評価し、あるアイテムに対しては評価が低いというそのユーザなりの一貫したコンテクストがあるはずである。しかし、その独立性にも関わらず比較的良い精度で推定をおこなえることが知られている。

クラスタリング

クラスタリングとは、全データ点をその類似度からいくつかの集合に分けること手法のことであり。その集合のことをクラスタと呼ぶ。類似度の計算は通常、 3.2.1 で述べたピアソン相関や、ミンコフスキー距離を用いておこなわれる。

2 つのデータ点 $X = (x_1, x_2, \cdots, x_n)$ と $Y = (y_1, y_2, \cdots, y_n)$ 間のミンコフスキー距離は以下のようになる。

$$d(X, Y) = \sqrt[q]{\sum_{i=1}^{n} |x_i - y_i|^q} \tag{3.7}$$

q は正の整数であり、$q = 1$ のときはマンハッタン距離を表し、$q = 2$ はユークリッド距離を表す。

collaborative filtering において、クラスタリングは中間ステップとして使われることが多い。例えば、全データ点をいくつかのクラスタに分け、そのそれぞれのクラスタ内において **??**のようにデータ点間の類似度計算をおこなう手法がある。[21, 28] クラスタリングによく用いられるアルゴリズムには k-means 法 [18] などがある。

特徴と問題点

学習モデルを構築することで巨大なユーザ-アイテムの行列を扱う必要がなくなるなど、次元削減によって処理時間の短縮が狙えるだけでなく精度の向上が見込める latent-model-based CF であるが、次元削減により有用な情報が抜け落ちる可能性がある。そもそもモデルを構築するのが高コストであるという問題もある。

3.2.3　hybrid CF

hybrid CF は collaborative filtering と content-based filtering を組み合わせた手法である。本稿では簡単に説明するにとどめるが、例えば、content-based filtering によってアイテムに関する説明文やユーザのプロフィールなどからユーザ-アイテム行列の欠損値を埋め、collaborative filtering につきもののデータのスパース性を解消するような試みである [23, 31, 4]。

3.3　評価指標

neighborhood-based、latent-model-based のどちらを用いた場合においても、そのアルゴリズムの性能は prediction 結果の精度の高さに直結している。そのため prediction 結果を評価するための指標が必要となる。アプリケーションによって適した評価指標は異なるが、ここでは、[10] から、一般によく用いられる

- Mean Absolute Error (MAE)
- Normalized Mean Absolute Error (NMAE)
- Root Mean Square Error (RMSE)

の 3 つの評価指標について紹介する。

- MAE、NMAE

  MAE とは prediction 結果と入力行列のレーティングの値との差の絶対値である。

  $$MAE = \frac{\sum_{u,v} |p_{u,v} - r_{u,v}|}{n} \tag{3.8}$$

  ここで、n は入力行列に含まれる全レーティングの数、$p_{u,v}$ はユーザ u がアイテム v に対してつけるレーティングの予測値、$r_{u,v}$ は実際のレーティングの値である。この値が小さいほど、精度が高いということになる。
  NMAE は MAE の値を割合として表したものである。

  $$NMAE = \frac{MAE}{r_{max} - r_{min}} \tag{3.9}$$

  ここで、r_{max} と r_{min} はそれぞれレーティングの最大値、最小値であり、MAE が最大値と最小値の差に対しての割合として正規化されている。

- RMSE

  RMSE は MAE とよく似た形をしているが、Netflix Prize でも評価指標として用いられたことから一般的になりつつある指標である。

$$RMSE = \sqrt{\frac{1}{n}\sum_{u,v}(p_{u,v} - r_{u,v})^2} \tag{3.10}$$

  ここで、n は入力行列に含まれる全レーティングの数、$p_{u,v}$ はユーザ u がアイテム v に対してつけるレーティングの予測値、$r_{u,v}$ は実際のレーティングの値である。この値が小さいほど、精度が高いということになる。

3.4　Netflix Prize

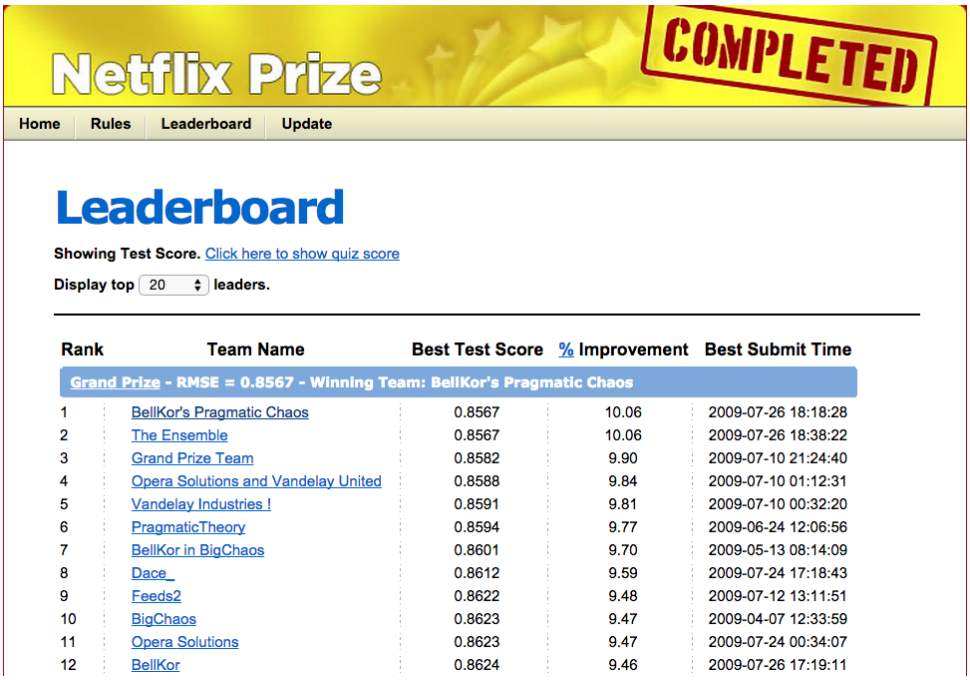


図 3.1. Netflix Prize のスコアボード（http://www.netflixprize.com/leaderboard のスクリーンショット）

Netflix Prize[1] とは、オンライン動画配信サービスを展開する Netflix が、より精度の高い推薦アルゴリズムを世界中の募集するために実施したコンテストである。当時 Netflix 社内では Cinematch というレコメンドエンジンが用いられていた。Cinematch は、ユーザがどの映画を like、dislike として評価したかに基づいて推薦をおこなうという点で collaborative filtering の一種である。図 3.1 はコンテストの最終的なスコアボードだが、優勝チームは Cinematch の精度を 10%以上改良している [14]。

このコンテストを遠して様々な手法が提案された [22, 33, 20, 36]。中でも、latent-model-based CF に属する行列補完（matrix factorization）というアルゴリズムを用いたチームが概して高い成績をおさめたことから注目度が増すに至った。行列補完アルゴリズムの詳細は次の賞で述べることとする。

第4章　行列補完アルゴリズム

4.1　概要

行列補完アルゴリズム（matrix factorization, matrix completion）[15] とは、collaborative filtering に属するアルゴリズムの 1 種である。本章で用いるパラメータと記号の対応表は表 4.1 の通りである。

表 4.1. パラメータと記号の対応表

パラメータ	記号
ユーザ数	m
アイテム数	n
モデル行列の次元	k
入力行列	$\boldsymbol{R} \in \mathbb{R}^{m \times n}$
モデル行列（ユーザ）	$\boldsymbol{P} \in \mathbb{R}^{k \times m}$
モデル行列（アイテム）	$\boldsymbol{Q} \in \mathbb{R}^{k \times n}$

まず、行に m 人のユーザ、列に n 個のアイテム、各エントリーにレーティングの点数をとった $m \times n$ の入力行列 $\boldsymbol{R}$ を考える。通常ユーザはごく少数のアイテムに対してのみ評価をするため、$\boldsymbol{R}$ は疎行列だとする。

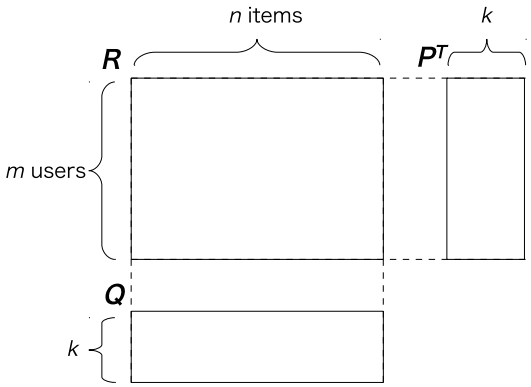


図 4.1. matrix factorization の概要図 1

行列補完では、図 4.1 のように 2 つの低ランク行列 $\boldsymbol{P} \in \mathbb{R}^{k \times m}$、$\boldsymbol{Q} \in \mathbb{R}^{k \times n}$ の内積で入力行列 R を近似する。つまり、

$$\boldsymbol{R} \approx \boldsymbol{P}^T \boldsymbol{Q} \tag{4.1}$$

と表すことができる。ここで、$\boldsymbol{P}$ はユーザをマッピングしたモデル行列、$\boldsymbol{Q}$ はアイテムをマッピングしたモデル行列であり、ユーザ u は $\boldsymbol{p}_u \in \mathbb{R}^k$、アイテム v は $\boldsymbol{q}_v \in \mathbb{R}^k$ という k 次元のベクトルで表すことができる。$\boldsymbol{p}_u$ はユーザ u があるアイテムに対して高い評価をつけるのか低い評価をつけるのかを潜在的に表した生成モデルということができる。$\boldsymbol{q}_v$ についても、アイテム v があるユーザから高い評価をつけられるのか低い評価をつけられるのかを表した生成モデルだということができる。

行列補完では、ユーザ u のアイテム v に対する評価予測 $\hat{r}_{u,v}$ を $\boldsymbol{p}_u$ と $\boldsymbol{q}_v$ の内積でモデリングする。

$$\hat{r}_{u,v} \approx \boldsymbol{p}_u^T \boldsymbol{q}_v \tag{4.2}$$

と表すことができる。

さて、行列補完の目標は、$\boldsymbol{P}^T\boldsymbol{Q}$ で形成される $\hat{\boldsymbol{R}}$ と実データである入力行列 $\boldsymbol{R}$ との差を可能な限り小さくすることである。それにより、入力行列中の欠損値、つまりあるユーザがまだレーティングを与えていないアイテムに対してどれほどの評価をつけるかをなるべく正確に予測し、レコメンドに活かすことができる。差を小さくするとは、入力行列中の非零要素 $r_{u,i}$ とモデルの予測値 $\boldsymbol{p}_u^T\boldsymbol{q}_v$ の差を小さくすることに等しい。つまり、もっとも単純な目的関数は以下のように表される。

$$\min_{\boldsymbol{P},\boldsymbol{Q}} \sum_{(u,v)\in\boldsymbol{R}} \left(r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v\right)^2 \tag{4.3}$$

4.1.1　正規化項

機械学習の一般的な問題として、訓練データに対しては非常によい精度を示すもののテストデータに対しては精度が悪くなることがある。これは「過学習」と呼ばれ、正則化項をつけて過学習を防ぐ試みが他の機械学習アルゴリズムでもなされる。

正則化項付きの目的関数は以下のようになる。

$$\min_{\boldsymbol{P},\boldsymbol{Q}} \sum_{(u,v)\in\boldsymbol{R}} \left(r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v\right)^2 + \lambda_P ||\boldsymbol{P}||_F^2 + \lambda_Q ||\boldsymbol{Q}||_F^2 \tag{4.4}$$

λ_P、λ_Q はモデルの過学習を防ぐための正則化係数であり、$||\cdot||_F$ はフロベニウスノルムを表す。

行列補完における過学習とは、入力行列 $\boldsymbol{R}$ の非零要素 $r_{u,i}$ については予測値 $\boldsymbol{p}_u^T\boldsymbol{q}_v$ が非常に近い値となるが、$\boldsymbol{R}$ 中の欠損値を予測すると大きく値がずれてしまうということを意味する。レコメンドにおいて欠損値を予測することこそが重要であるため、過学習を起こすことはレコメンドの精度の悪化と直結している。そのため、正則化項をつけることで過学習を防ぐことは非常に重要である。

4.1.2　バイアス項

式 4.3 はユーザとアイテムの相関からレーティングを一様に推定しているが、そもそもユーザごと、アイテムごとにレーティングの値に違いがあるのが普通であるユーザによってレーティングに星 5 つを与える基準は異なっているため、アイテムに与えるレーティングの平均値によって異なりうる。また、レーティングの平均値もアイテムごとに異なりうる（あの映画はこの映画よりも面白いなど）ので、ユーザごと、アイテムごとにバイアス項を設定すると精度に良い影響をもたらすことが知られている。[15]

レーティング $r_{u,v}$ におけるバイアス $b_{u,v}$ は以下の式で表すことができる。

$$b_{u,v} = avg + \boldsymbol{ub}_u + \boldsymbol{ib}_v \tag{4.5}$$

すると、評価予測 $\hat{r}_{u,v}$ は以下の式で表される。

$$\hat{r}_{u,v} = b_{u,v} + \boldsymbol{p}_u^T \boldsymbol{q}_v \tag{4.6}$$

avg は全レーティングの平均値を表し、$\boldsymbol{ub}_u$ と $\boldsymbol{ib}_v$ はそれぞれユーザ u、アイテム v における平均からのバイアスである。例えば、全レーティングの平均値 avg が 3.2 であり、スターウォーズのエピソード 2 は評価が高く、ユーザからの平均が 3.9 であるとする。一方、山田くんは辛口な映画評論家で、全映画に対するレーティングの平均が 2.8 であるとすると、山田くんがエピソード 2 に与えるであろうレーティングの点数は $3.2 + 0.7 - 0.4 = 3.5$ になると予想される。

バイアス項を反映した目的関数を以下に示す。

$$\begin{aligned} \min_{\boldsymbol{P},\boldsymbol{Q},\boldsymbol{a},\boldsymbol{b}} & \sum_{(u,v)\in\boldsymbol{R}} \left(r_{u,v} - avg - \boldsymbol{ub}_u - \boldsymbol{ib}_v - \boldsymbol{p}_u^T \boldsymbol{q}_v\right)^2 \\ & + \lambda_P ||\boldsymbol{P}||_F^2 + \lambda_Q ||\boldsymbol{Q}||_F^2 + \lambda_{ub} ||\boldsymbol{ub}||_F^2 + \lambda_{ib} ||\boldsymbol{ib}||_F^2 \end{aligned} \tag{4.7}$$

4.1.3　implicit なフィードバックの利用

3.1.1 章で述べた通り、レーティングというユーザが与えた明示的な評価情報のみを扱う collaborative filtering は、新しいユーザや新しいアイテムが入ってきたときに入力行列のスパース性に起因する「コールドスタート問題」に遭遇する。そこで、レーティングのような explicit なデータだけではなく、ユーザのアイテム閲覧履歴や購買履歴などの implicit な情報を利用して入力行列の欠損値を埋めることで、この問題を回避することができる [12, 11]。

4.1.4　時間によるユーザの嗜好変化の考慮

これまでに述べてきた学習モデルの扱い方は静的、つまり時間経過による変化は考慮していない。しかし現実には、時が経つにつれてアイテムへの評価が変わる、あるいはユーザの嗜好が変化するという動的な要素が加わる。例えば、公開直後は人々に受け入れられなかったが何年か後に再評価されるような映画もあれば、同じ映画でも昔は星 4 つとして評価したとしても今では目

が肥えて星 3 つとしか評価しなくなるユーザがいる可能性もある。具体的には、ユーザバイアス $\boldsymbol{ub}_u(t)$、アイテムバイアス $\boldsymbol{ib}_v(t)$、ユーザの生成モデル $\boldsymbol{p}_u(t)$ は時間によって変わりうるものである。

時間変化を含めた評価予測値 $\hat{r}_{u,v}$ は以下の式で表される。

$$\hat{r}_{u,v} = avg + \boldsymbol{ub}_u(t) + \boldsymbol{ib}_v(t) + \boldsymbol{p}_u(t)^T \boldsymbol{q}_v \tag{4.8}$$

4.2　最適化

式 4.3 で表される最適化問題の解法はいくつか考えられるが、ここでは

- gradient descent 法
- stochastic gradient descent 法

4.2.1　gradient descent 法

まず、gradient descent 法（以下 GD 法）について説明する。GD 法は、全データ点における予測誤差の描く多次元平面の勾配を計算し、勾配と逆方向に特徴ベクトルを更新する手法である。

サンプル数を N、学習モデルのベクトルを w、各サンプルの特徴ベクトルを x_i、サンプルの予測誤差を $f(i)$ とすると、全サンプルにおける予測誤差の総和は、

$$\sum_{i=0}^{N} f(i)$$

と表される。すると、w は以下のようにして更新される。

$$w \leftarrow w - \nabla \sum_{i=0}^{N} f(i) \tag{4.9}$$

GD 法を行列補完に適用する。目的関数には正則化項つきの式 4.4 を用いるとすると、
$u(0 < u < m)$ において、

$$\boldsymbol{p}_u \leftarrow \boldsymbol{p}_u + \gamma \left(\sum_{(u,v)\in \boldsymbol{R}} (r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v)\boldsymbol{q}_v - \lambda_P \sum_{u=1}^{m} \boldsymbol{p}_u \right) \tag{4.10}$$

$v(0 < v < n)$ において、

$$\boldsymbol{q}_v \leftarrow \boldsymbol{q}_v + \gamma \left(\sum_{(u,v)\in \boldsymbol{R}} (r_{u,v} - \boldsymbol{p}_u^t \boldsymbol{q}_v)\boldsymbol{p}_u - \lambda_Q \sum_{v=1}^{n} \boldsymbol{q}_v \right) \tag{4.11}$$

とモデル行列を更新する。疑似コードは図 4.2 のようになる。

GD 法の問題点として、式 4.9 の勾配計算がコスト高であることに加え、全レーティングを処理した後でモデルを更新するというバッチ的な性質から、収束が遅いという点が挙げられる。

```
for every iteration:
  for r in all ratings in rating matrix R:
    compute loss
  end for
  update P and Q
end for
```

図 4.2. gradient descent 法の疑似コード

4.2.2　stochastic gradient descent 法

これを解決するのが stochastic gradient descent 法（以下 SGD）[16] である。SGD 法では、1 つのサンプルを処理するたびにモデルベクトルの更新をおこなう。更新式は以下のように表される。

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \nabla f(i) \tag{4.12}$$

式 4.9 と比較してもわかるように、GD 法では全サンプルにおける誤差を合計するバッチ的な処理だったが、SGD 法では 1 つのサンプルの誤差からオンライン的にモデルベクトルを更新している。

行列補完に SGD 法を適用すると、

$$\boldsymbol{p}_u \leftarrow \boldsymbol{p}_u + \gamma \left((r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v) \boldsymbol{q}_v - \lambda_P \boldsymbol{p}_u \right) \tag{4.13}$$

$$\boldsymbol{q}_v \leftarrow \boldsymbol{q}_v + \gamma \left((r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v) \boldsymbol{p}_u - \lambda_Q \boldsymbol{q}_v \right) \tag{4.14}$$

疑似コードは図 4.3 のようになる。

```
for every iteration:
  for r in all ratings in rating matrix R:
    update p[r.user_id] and q[r.item_id]
  end for
end for
```

図 4.3. stochastic gradient descent 法の疑似コード

SGD 法ではメモリ上に全レーティングを保持する必要がないため省メモリであるということ、特に大規模データにおいては GD と比較して収束が速いということから、近年特に注目を浴びている最適化アルゴリズムの 1 つである。

オンライン的に更新をおこなうという SGD の性質上、並列化が難しいという難点がある。扱うデータの量が肥大化している昨今では並列化、分散化は必須の流れであることから、並列 SGD を行列補完に適用する手法がいくつか提案されている [7, 26, 37]。次章でそれら関連研究について紹介する。

第5章　関連研究

本章では、行列補完に並列 SGD を適用した研究についていくつか言及する。

5.1　HogWild!

行列補完問題を共有メモリ環境で並列化するもっともナイーブな手法は、

- レーティング $r_{u,v}$ を入力行列の中からランダムに選択
- 対応する p_u, q_v を更新

の処理を全ワーカが独立におこなうことである。しかし、この手法では複数ワーカが同じ行あるいは同じ列上に存在するレーティングをほぼ同時に読み込み、一方のワーカの更新を他のワーカ更新が上書きすることが起こり得る。

並列アルゴリズムでは、このような共有リソースへの同時書き込みを防ぐためにロックを用いることが多い。しかし、ロックを用いると実行がシリアライズされ、パフォーマンスに悪影響を与えることが往々にしてある。Niu らは、入力行列が十分に疎であればロックなしで並列化しても収束することを証明し、ロックを用いた実装と彼らの提案する lock-free な実装では後者の方がより速く収束することを示した [7]。

しかし、ワーカ数が増えると同時書き込みが多発することが考えられるため、スケーラビリティという点では疑問である。また、各ワーカが入力行列の中からランダムにレーティングを選択するため、ハードウェアプリフェッチによるキャッシュ内データの再利用ができない。

5.2　DSGD

Gemulla らは、分散環境における行列補完の適用を提案した [26]。この手法では、以下の流れで処理がおこなわれる。

- 入力行列を $t \times t$ のブロックに分割（t はワーカ数）
- ワーカにブロックを割り当てる。なお、ワーカに割り当てられたブロック同士は同じ行・同じ列を共有しないようにする。
- 全ワーカが割り当てられたブロックの処理を終えると、全ワーカが同時に次のブ ロックへと移動する。

各ワーカは、HogWild!の場合と同じように、

- レーティング $r_{u,v}$ を入力行列の中からランダムに選択

- 対応する p_u, q_v を更新

この手法の特徴は、入力行列を多数のブロックに分割し、ワーカはブロック内のレーティングのみ処理する点である。割り当てられるブロック同士は同じ行・同じ列を共有していないため、同時書き込みを防ぐことができる。

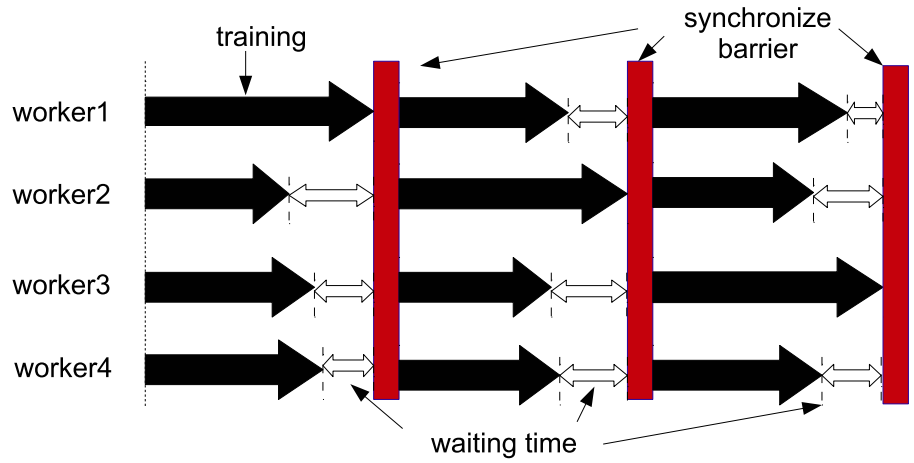


図 5.1. synchronous parallel による同期バリアの模式図

この手法の欠点はワーカのブロック移動が同期的におこなわれることである。入力行列における rating の分布が不均一であるなどの理由であるワーカのみ他のワーカよりブロックの処理に時間がかかった場合、他のワーカは遅れているワーカを待たなければならず、パフォーマンスの低下につながる。また、ワーカは割り当てられたブロック内のレーティングをランダムに選択するため、やはりプリフェッチを活かすことができないということも欠点として考えられる。

5.3　FPSGD

Zhuang らは、上記 2 つの手法の問題点を解決するアルゴリズムとして、FPSGD を提案した [37]。この手法では、

- conflict-free scheduling
- partial random method

という 2 つの手法を導入した。以下では上の 2 つの手法について説明する。

5.3.1　conflict-free scheduling

まず、FPSGD でも DSGD と同様に入力行列を多数のブロックに分割する。図 5.2 では、入力行列を 6×6 に分割した例を示している（ワーカ数は 4）。まず最初にワーカには、DSGD と同様にそれぞれ行と列を共有しないブロックがスケジューラにより割り当てられる。ここで、ワーカ T_0 が最初に処理を終えたとする。他のワーカ T_1〜T_3 が処理しているブロックの行と列は同じ

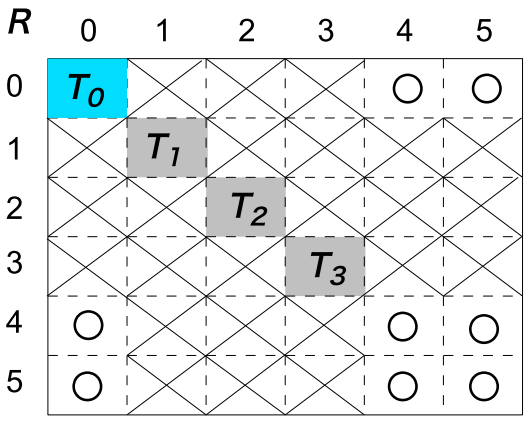


図 5.2. conflict-free scheduling の模式図

行あるいは同じ列への同時書き込みを防ぐためにブロックされているため、割り当てが可能なブロックは丸がついたブロックのみである。よってスケジューラは丸がついたブロックの中からランダムに 1 つブロックを選び出し、ワーカに割り当てる。この割り当て方式により、ワーカは

- HogWild!で起こりうる同時書き込みを防ぎつつ、
- DSGD の同期バリアを排除することで、

非同期的にブロックの処理をおこなう。

5.3.2　partial random method

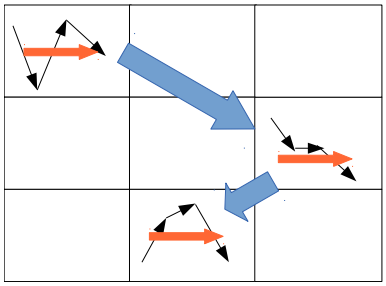


図 5.3. partial random method の模式図

HogWild!、DSGD ではワーカはランダムにレーティングを選択、処理している。ランダムなサンプル選択は収束に寄与することが知られているが、先述したようにハードウェアプリフェッチを活かすことができず、最悪の場合レーティング $r_{u,v}$ を読みこむたびにメモリからキャッシュ上に p_u、q_v を載せる操作が必要となる。そこで FPSGD では、図 5.3

- ブロックの選択はランダム
- レーティングの選択はユーザ ID 順あるいはアイテム ID 順におこなう

ことを提案した。

これにより、ユーザ ID 順にレーティングが処理されるとすると、少なくともブロック内においてはワーカはプリフェッチにより一度キャッシュに読み込んだ p_u もしくは q_v を再利用できる可能性が高くなると言える。

5.4　まとめ

本章にて述べた 3 つの先行研究を

- 並列実行方式（同期 or 非同期）
- キャッシュの再利用効率

の 2 つの観点でまとめると次の表 5.1 のようになる。

表 5.1. 先行研究の比較表

	parallel	キャッシュの再利用効率	
		ブロックの選択	レーティングの選択
HogWild!	非同期	-	ランダム
DSGD	同期	順番	ランダム
FPSGD	非同期	ランダム	順番

HogWild!は処理は非同期的に進められるもののキャッシュの再利用効率としては劣悪、DSGD では処理は同期的におこなわれるためパフォーマンスは遅いワーカに依存し、さらにレーティングの選択もランダムなのでキャッシュ再利用効率もあまり良いとは言えない。この中では並列実行方式、キャッシュの再利用効率ともに FPSGD が一番優れていると考えることができ、事実、彼らの論文では HogWild!、DSGD の性能を大きく上回っていることから、FPSGD が共有メモリ環境における行列補完の並列化の研究としては state-of-the-art だと言える。

第6章　FPSGDの問題点

前章で述べた通り、FPSGD は行列補完の並列化の研究としては state-of-the-art だと考えられるが、彼らの論文では 8 スレッドを用いた際の結果しか載っておらず、スケーラビリティの評価が無い。そこで我々は彼らのオープンソースの実装[1]を用いてスケーラビリティの測定をおこなった結果、データセットにもよるものの使用するワーカ数が多いとスケーラビリティが著しく低下することが分かった。スケーラビリティを阻害する原因は

- ロック待ち時間の増大
- キャッシュミスの増加

と 2 つあると考えられる。以下にそれぞれの詳細について述べる。

6.1　ロック待ち時間の増大

図 6.1 に FPSGD の疑似コードを示す。あるワーカがブロックを処理する流れは以下のようになる。

- Scheduler クラスの get_job メソッドでブロックを取得し、その行と列を他のワーカがアクセスできないようにブロックする
- update_by_sgd 関数でブロック内のレーティングを処理、モデルの更新をおこなう
- Scehduler クラスの put_job メソッドでブロックを解放すると同時に行と列のブロックを解除する

get_job メソッド、put_job メソッドともに変数への書き込みが必要となるため、Scheduler クラスのメンバ変数である lock を取得してアクセスする。全てのワーカが単一のロックを取得しようとするため、ワーカ数が増えた場合にボトルネックになることが予想される。

6.2　キャッシュミスの増加

FPSGD では、DSGD と異なりワーカへのブロックの割り当てはランダムにおこなわれる。データセットの大きさと分割するブロック数によるが、あるブロックに対応する P と Q のブロックはキャッシュサイズよりも大きいため、ワーカが次のブロックに移動するとキャッシュ内のデータが全て載せ替えられる、つまりメモリとキャッシュ間のデータ転送が起こる可能性が高いということを示している。

[1] http://www.csie.ntu.edu.tw/~cjlin/libmf/

```
t: number of threads

class Scheduler {
  Lock lock;
  Block get_job () {
    get the lock and search for a free block
    mark the free block as 'being processed'
    return the free block
  }
  void put_job (Block block) {
    get the lock and increment the counter of the block
    unmark the block as 'free'
  }
};

void update_by_sgd (Block b, Model m) {
  for each r in b do
    update P[r.user_id] and Q[r.item_id]
  end for
}

void train () {
  divide R into at least (t+1) * (t+1) blocks
  for each iteration do
    for each thread do (in parallel)
      while (true)
        // the scheduler assigns a new block
        // スケジューラが
        block = scheduler.get_job()
        // update learning model
        update_by_sgd(block, model);
        // increments the processed count
        scheduler.put_job(block)
        break if all blocks are processed
      end while
    end for
  end for
}
```

図 6.1. FPSGD の疑似コード

ここで、入力行列全体を処理するために発生するメモリ-キャッシュ間のトラフィックを理論的に見積もると式 6.1 のようになる。用いる記号は表 6.1 に示す。

$$\begin{aligned} T_{fpsgd} &= \left(\frac{n_r}{b^2} + \frac{kn_u}{b} + \frac{kn_i}{b}\right) \times b^2 \\ &= (n_r + bk(n_u + n_i)) \quad \text{words.} \end{aligned} \tag{6.1}$$

表 6.1. パラメータと記号の対応表

パラメータ	記号
ユーザ数	n_r
アイテム数	n_i
レーティング数	n_u
ブロックの分割数	b
モデル行列の次元	k

式 6.1 の左辺第 1 項は 1 ブロックに含まれるレーティング数（単純のため全てのブロックに均一にレーティングが分布していると仮定している）、左辺第 2 項、第 3 項はそれぞれブロックに対応する P、Q のサイズ、1 word は単精度浮動小数点なので 4 バイトである。

第 7 章　分割統治法による行列補完の並列化

本研究では分割統治法による行列補完アルゴリズムの並列化を提案する。本章では、提案手法の概要とタスク並列モデルを用いた並列化とそれにより得られる利点について言及する。

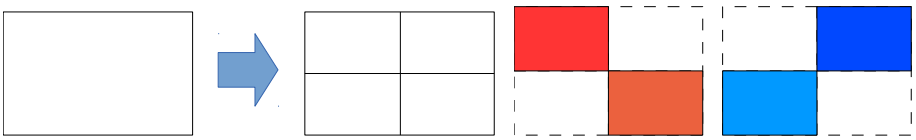
7.1　概要

提案手法では、目的の大きさのブロックが得られるまで再帰的に入力行列を分割する手法を提案する。各再帰において、

ステップ 1:　ブロックを 2×2 のサブブロックに分割する（図 7.1a）

ステップ 2:　同一対角線上のブロックを処理する（図 7.1b）

ステップ 3:　もう一方の対角線上のブロックを処理する（図 7.1c）



(a) ステップ 1　(b) ステップ 2　(c) ステップ 3

図 7.1. dcMF の各ステップの概要図

ステップ 2、3 においてそれぞれ対角線上のブロックのみ処理することで、これらのブロックは行・列ともに共有していないためワーカはデータ競合なしに独立に処理することができる。

7.2　タスク並列モデルを用いた並列化

7.2.1　タスク並列プログラムの例

タスク並列モデルでは、並列処理は

- *create_task*
- *sync_task*

の 2 つの命令で表現される。*create_task* はタスクを生成し、*sync_task* でタスクの終了を待ち合わせる。生成されたタスクは実行時システムによって動的にワーカに分散されるため、ユーザは負荷分散のためのコードを書かずとも簡単に並列プログラムを実装することができる。

```
1 int fib(int n) {
2   if (n > 2) return 1;
3   else {
4     int x, y;
5     x = fib(n-1);
6     y = fib(n-2);
7     return x + y;
8   }
9 }
```

(a) タスク並列なし

```
1  int task_parallel_fib(int n) {
2    if (n > 2) return 1;
3    else {
4      int x, y;
5      x = create_task fib(n-1);
6      y = create_task fib(n-2);
7      sync_task;
8      return x + y;
9    }
10 }
```

(b) タスク並列あり

図 7.2. フィボナッチ数列問題の疑似コード

図 7.2 はフィボナッチ数列問題の疑似コードを示しており、図 7.2a は逐次のプログラム、図 7.2b はタスク並列のプログラムの疑似コードである。図 7.2b より、*create_task* を関数呼び出しの前につけることで、関数呼び出しをタスクとみなし、その実行をワーカに割り当てることで並列実行をおこなうことができる。

7.2.2　行列補完の並列化

```
1  void dcMF(Block b, Model m) {
2    if b is smaller than a user-defined threshold
3      update_by_sgd(b, m);
4      return;
5    end if
6    // ブロックを 2 x 2のサブブロックに分割する(ステップ 1)
7    Block children[2][2]
8    divide_blocks(b, children);
9    // 同一対角線上のブロックを処理する(ステップ 2)
10   create_task dcMF(children[0][0], m);
11   create_task dcMF(children[1][1], m);
12   sync_task;
13   // もう一方の対角線上のブロックを処理する(ステップ 3)
14   create_task dcMF(children[0][1], m);
15   create_task dcMF(children[1][0], m);
16   sync_task;
17 }
18
19 void train () {
20   for each iteration do
21     create_task dcMF(R, model);
22   end for
23 }
```

図 7.3. dcMF の疑似コード

図 7.3 は dcMF の疑似コードを示している。先の概要にも述べたように、まず 7〜8 行目においてブロックを 2×2 のサブブロックに分割する。次に、10〜11 行目において同一対角線上のブ

ロックを処理する。このとき、再帰関数の呼び出し元を親とすると、*create_task* をつけて子タスクが生成され、親は 12 行目の *sync_task* で全ての子タスクの終了を待つ。そのあと、もう一方の対角線上にブロックについても同様に処理される。ブロックがユーザの指定した大きさに到達した場合は再帰呼び出しはおこなわず、レーティングの処理がおこなわれる。

7.3　タスク並列モデルを用いる利点

分割倒置法とタスク並列モデルを組み合わせることで得られる利点として、

- ロックの排除
- キャッシュミスの削減

の 2 つがあると考えられる。本項では、これらの利点について言及する。

7.3.1　ロックの排除

FPSGD では、ブロック処理の前後で単一ロックを取得して共有リソースへの書き込みをおこなうことがボトルネックになっていると考えられるが、dcMF では生成されたタスクの負荷分散は実行時システムに一任することができ、ロックのような中央集権的なデータ構造をもつ必要がない。これにより、ロック待ちの時間の削減につながると考えられる。

7.3.2　キャッシュミスの削減

ワーカへのタスクの割り当ては、このタスク木の根元付近から木構造を分割しておこなわれるという性質がある。図 7.4 を例にとって考える。そのため、例えばノード A から下の部分木は全てワーカ 1 が処理する、ノード B から下の部分木は全てワーカ 2 が処理するというようにワーカはある程度の塊でブロックを処理することになる。これはワーカごとに負荷をなるべく均一にしようという目的でおこなわれるが、それでもタスクごとの大きさに差がある場合は他のワーカよりも処理が早く終わって暇になるワーカが出現することがある。

我々が実装のなかで用いたタスク並列モデルは、そのようなタスクが割り当てられていないワーカが出現すると、実行時システムが他のワーカからタスクを取ってきてそのワーカに割り当てるという動作をする。これはワークスチーリングと呼ばれるもので、ワーカ間の負荷分散を動的におこなうための仕組みの 1 つである。

さて、ここで行列補完アルゴリズムの場合について考えてみる。行列補完に分割統治法とタスク並列を適用した場合、生成されるタスクはやはり図 7.4 のような木構造を形成する。ここでは入力行列は 4×4 で分割されているとする。この図ではタスクは丸いノードで表され、ノードの中の数字はそのタスクが動作したブロックの番号である。末端ではないノードはまず実線で繋がれた 2 つの子タスクを生成し（ステップ 2）、それらの実行が終わるのを待ってから破線で繋がれた 2 つの子タスクを生成する（ステップ 3）。

先に述べたように、ワーカに対するタスクの割り当てはタスク木の根元付近から分割しておこなわれるため、ワーカが 2 つ存在したとすると、一方の対角線上にあるブロックについてはノー

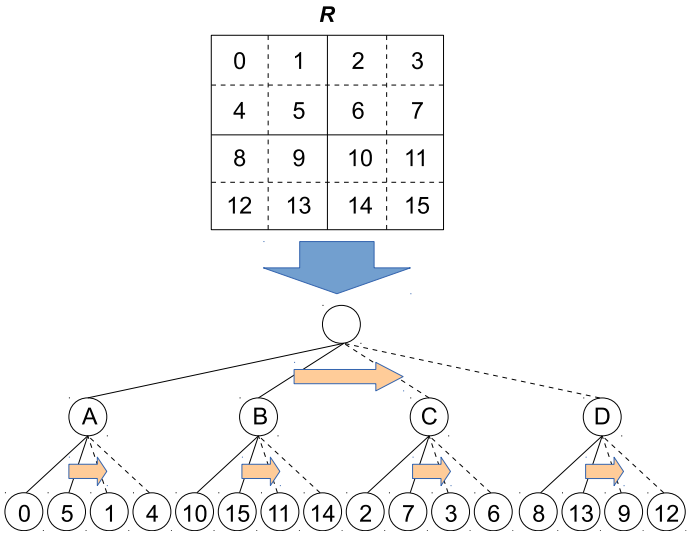


図 7.4. タスクツリーの模式図

ド A から下の部分木はワーカ 1 に、ノード B から下の部分木はワーカ 2 に割り当てられる（ワーカの番号を入れ替えても同様）。もう一方の対角線上にあるブロックについても、ノード C から下の部分木はワーカ 1 に、ノード D から下の部分木はワーカ 2 に割り当てられる。

そのため、例えばノード A から下の部分木は全てワーカ 1 が処理する、ノード B から下の部分木は全てワーカ 2 が処理する、という形になる。ここで、1 つのブロックを処理するのにかかる時間が全部のブロックで一定、つまりブロックに含まれるレーティングの数が全く均一であるとすると、各ワーカが処理するブロックは

ワーカ 1:　0,1,2,3,4,5,6,7

ワーカ 2:　8,9,10,11,12,13,14,15,16

というように、ある程度のブロックの集合がまるまる 1 つのワーカに処理されることになる。FPSGD でブロックの割り当てが完全にランダムにおこなわれていたのに対し、dcMF ではブロックの割り当てがある程度の規則性をもっておこなわれるので、ブロック間のキャッシュ上のデータを再利用できる可能性が高くなるということを示唆している。

表 7.1. パラメータと記号の対応表

パラメータ	記号
ユーザ数	n_r
アイテム数	n_i
レーティング数	n_u
ブロックの分割数	b
モデル行列の次元	k
スレッド数	t

ここで、dcMF において、入力行列全体を処理するために発生するメモリ-キャッシュ間のトラフィックを理論的に見積もると、

$$T_{dcmf} \quad = \quad (n_r + tk(n_u + n_i)) \quad \text{words}, \tag{7.1}$$

となる。FPSGD のトラフィックを示す 6.1 との違いは、左辺第二項の係数の b が t に変わったことである。これはつまり、$b > t$ であれば dcMF のトラフィックは FPSGD のそれより少なくなるということを意味する。

7.4　まとめ

表 7.2. 提案手法と先行研究の比較表

	parallel	キャッシュの再利用効率	
		ブロックの選択	レーティングの選択
HogWild!	非同期	-	ランダム
DSGD	同期	順番	ランダム
FPSGD	非同期	ランダム	順番
dcMF	非同期	ある程度順番	順番

表 7.2 は提案手法と先行研究との比較を示している dcMF ではワーカへのブロックの割り当てはある程度決まった順番でおこなわれるため、FPSGD と比較してブロック間のデータローカリティという点では優れていると考えられる。

第8章　評価

本章では、提案手法の評価をおこなう。比較実験では、先行研究の中では FPSGD が state-of-the-art であるため、FPSGD との比較実験をおこなった。パラメータの変化による性能評価では、学習率などのパラメータの変化が収束性能に与える影響を調べた。

8.1　評価環境

まず、我々は FPSGD のオープンソース実装である *libmf*[1] というライブラリを使って実装をおこなった。このライブラリは C++で実装されている。提案手法の実装に用いられたタスク並列ライブラリは当研究室で開発された MassiveThreads[13] を使用した。

実験環境は、AMD Opteron 8354 2.50GHz ソケットを 4 枚載せた NUMA（non-uniform memory access）アーキテクチャである。各ソケットには 2 コアが載ったモジュールが 8 つ存在するので、総コア数は $4 \times 8 \times 2 = 64$ コアである。

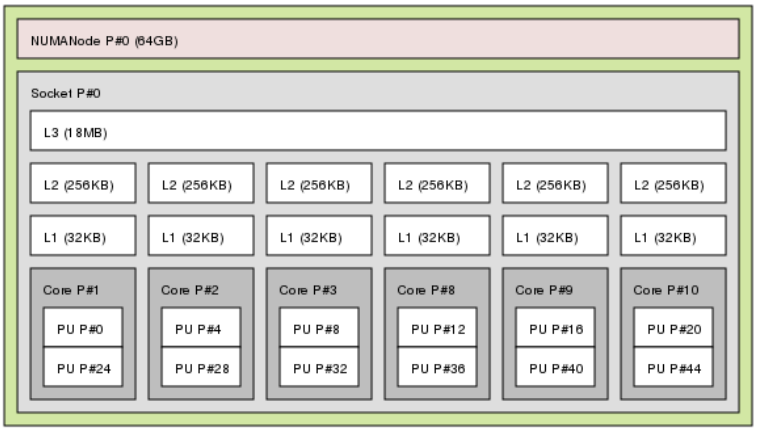


図 8.1. AMD Opteron 8354 におけるコアとキャッシュの関係

図 8.1 は Opteron 8354 ソケットのコアとキャッシュの関係を示している。この図から、まず L3 キャッシュ（18MB）はソケット上の全てのモジュールで共有されており、L1 キャッシュ、L2 キャッシュはともに 2 コアで共有されている。1 つのモジュールにスレッドを 2 つマッピングして L2 キャッシュを共有しないようにするため、1 スレッドが使用できるキャッシュ領域を可能な限り大きくとるため、使用するスレッド数が 32 以下の場合は numactl コマンドの physcpubind オプションを使用して 1 つのモジュールにつきスレッドを 1 つマッピングした。

[1] `http://www.csie.ntu.edu.tw/~cjlin/libmf/`

また、4 枚のソケット間でのメモリ割付けを均一にするため、numactl コマンドの interleave オプションを all にして実験をおこなった。

```
1 numactl --interleave=all --physcpubind=0,2,4,6, ...
```

表 8.1. 使用したデータセットとその統計的特性、パラメータの値

パラメータ/データセット	MovieLens10M	Netflix	Yahoo!Music
m (ユーザ数)	71567	2649429	1000990
n (アイテム数)	65133	17770	624961
学習に使用したレーティング数	9301274	99072112	252800275
テストに使用したレーティング数	698780	1408395	4003960

我々の実験では 3 種類のデータセットを使用した。それぞれのデータセットの特性を表 8.1 に示す。

8.2　比較実験

我々が FPSGD との比較実験をおこなったのは次の 3 項目である。

- スケーラビリティ
- 収束速度
- L2 キャッシュミス数

これら全ての実験に共通する点として、以下の点が挙げられる。

- FPSGD と同様に入力行列 R を $2t \times 2t$ に分割した（t はスレッド数）
- 入力行列内のレーティングの分布の偏りを是正するために、FPSGD でも使用されている random shuffling method を適用した
- SGD は本来全サンプルの中からランダムにサンプルを選んで更新するが、FPSGD にならい全レーティングを更新した

比較実験において使用したパラメータの値は、公平性のため FPSGD と同じ値を使用した。パラメータの値を表 8.1 に示す。

表 8.2. 使用したデータセットとその統計的特性、パラメータの値

パラメータ/データセット	MovieLens10M	Netflix	Yahoo!Music
k (モデル行列の次元)	40	40	100
γ (学習率)	0.003	0.002	0.0001
λ_p (P の正則化係数)	0.05	0.05	1
λ_q (Q の正則化係数)	0.05	0.05	1

以下でそれぞれの比較実験の詳細について述べる。

8.2.1　スケーラビリティ

本実験では、1 イテレーションにかかる時間、すなわち全てのブロックを 1 度処理するのにかかる時間を使用するスレッド数を変えて計測、スケーラビリティを比較した。25 イテレーションする 1 回の計測を 10 回繰り返し、それらの平均値を 95%信頼区間付きで使用した。また、FPSGD のスケーラビリティを阻害する原因を調べるために、FPSGD の実行時間に占めるロック待ち時間の割合も計測した。ロック待ち時間とは、あるワーカがロックを所有してクリティカルセクションを実行中は他のワーカはそのワーカがロックを解放するのを待つ時間のことである。実験では以下の計算式にしたがってロック待ち時間の割合を算出した。

$$\text{ロック待ち時間の割合} = \frac{\text{全スレッドのロック待ちにかかる時間の合計}}{\text{全スレッドの実行時間の合計}} \tag{8.1}$$

結果を図 8.3 に示す。左のカラムはデータセットごとのスケーラビリティの比較を表しており、横軸は使用したスレッド数、縦軸はスケーラビリティ（逐次での性能で割った値）である。右のカラムはデータセットごとの FPSGD のロック待ちが実行時間に占める割合を表しており、横軸は使用したスレッド数、縦軸は実行時間に占めるロック待ち時間の割合である。

まず、スケーラビリティ比較の図から、特に MovieLens10M と Netflix において、FPSGD は使用するスレッド数が多い場合にスケーラビリティが著しく低下していることがわかる。右のカラムを見ると、MovieLens10M では 24 スレッド、Netflix では 44 スレッドから FPSGD のロック待ち時間が増大していることがわかる。これは使用スレッド数の増加により、あるスレッドがロックを所有している間は他のスレッドはロックが解放されるのを待つため、ロック待ち時間が増えていると考えることができる。一方、Yahoo!Music では最大 64 スレッドでも FPSGD のロック待ち時間の割合は約 3%とそこまで大きなウェイトを占めてはいない。これは Yahoo!Music が使用したデータセットの中では最も大きなサイズのため、ブロック処理時間に比べロックの取得から解放までの時間が比較的小さいためと考えられる。そのため、Yahoo!Music での FPSGD のスケーラビリティで見てもスケーラビリティは維持できている。

それに対して我々の提案手法である dcMF はどのデータセットにおいてもスケーラビリティを維持することができている。

8.2.2　収束速度

次に、収束までにかかる時間を比較した。学習モデルの精度評価指標としては RMSE を用いた。

なお、使用したスレッド数は FPSGD の性能が最も高くなるスレッド数でおこない、MovieLens10M では 24 スレッド、Netflix では 32 スレッド、Yahoo!Music では 60 スレッドを使用した。

結果を図 8.4 と表 8.3 に示す。図 8.4 は収束までの RMSE の遷移を示しており、横軸は経過時間、縦軸は RMSE の値である。表 8.3 は収束した時の RMSE の値と、それまでに必要としたイテレーション数と経過時間である。まず図 8.4 より、全てのデータセットにおいて dcMF の方がより早く収束していることがわかる。一方 FPSGD において、Netflix においては dcMF と収束速度にそれほど大きな差はないように見受けられるが、FPSGD は全てのデータセットにおいて RMSE の値が小さくになるにつれ値が前後する現象が見られた。

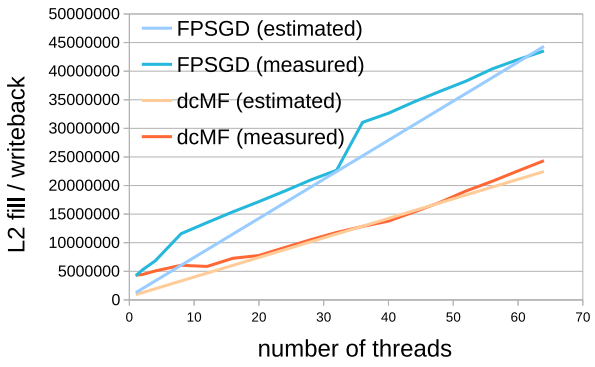
表 8.3. 収束した時の RMSE の値とそれまでに必要としたイテレーション数と経過時間

データセット	手法	イテレーション回数	時間（秒）	RMSE
MovieLesn10M	FPSGD dcMF	144 146	9.63 7.25	0.835
Netflix	FPSGD dcMF	165 166	113.08 92.79	0.918
Yahoo!Music	FPSGD dcMF	138 139	1240.29 339.32	21.820

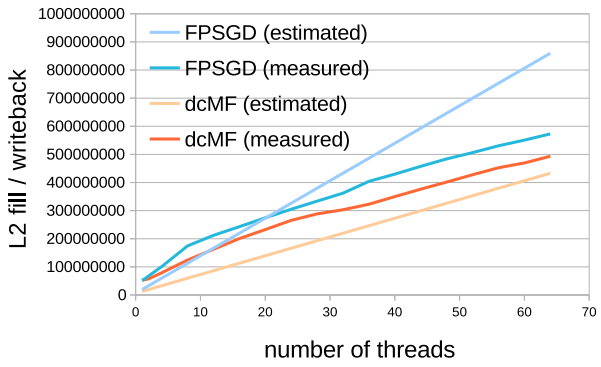
また、表 8.3 からも、dcMF の方が FPSGD よりも早く収束していることがわかる。イテレーション数に大きな開きはないため、この差は 1 回のイテレーション時間の削減により得られたものだと思われる。

8.2.3　キャッシュミス数

次に、*update_by_sgd* 関数内での p_u と q_v の内積計算におけるキャッシュミス数を比較した。ここで、我々は単純なキャッシュミス数ではなく、L2 キャッシュの fill/writeback 数を比較した。fill/writeback 数は、データがキャッシュライン単位でメモリからキャッシュへ読み込まれた回数のことである。計測には *perf* コマンドと AMD CPU のハードウェアイベント *r037F* を用いた [6]。



(a) MovieLens10M での L2 fill/writeback 数

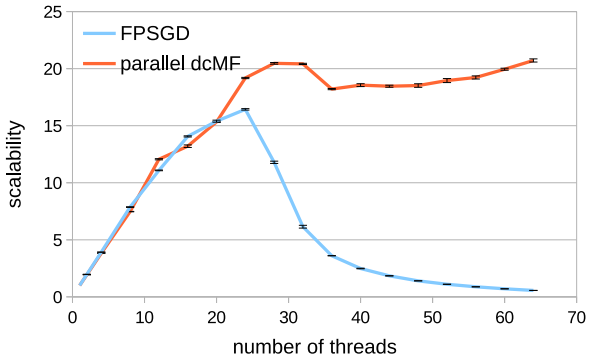


(b) Netflix での L2 fill/writeback 数

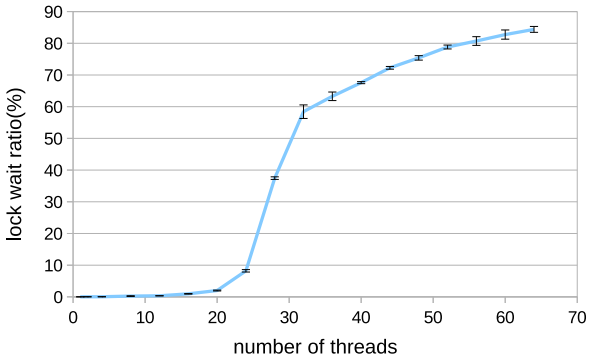
図 8.2. データセット毎の収束速度の比較

結果を図 8.2 に示す。図 8.2a に MovieLens10M を用いた結果、図 8.2b に Netflix を用いた結果である。両方の結果は式 6.1、式 7.1 に示される理論値と実測値を合わせて表示している。MovieLens10M では FPSGD、dcMF ともに理論値によく fit しており、dcMF では FPSGD よりも fill/writeback 数が減っていることがわかる。しかし、Netflix では FPSGD の実測値が理論値を大きく下回り、dcMF との差はそれほど顕著ではなくなっている。これは Netflix ではレーティングの分布に偏りがあり、スレッド間の負荷を均一にするためにワークスチーリングが頻繁に起こったことで、スレッドがデータローカリティを無視してブロックを処理したためではないかと思われる。

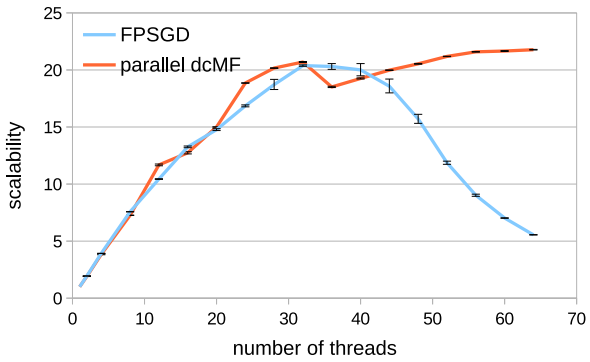
実際にスレッドがどのブロックを処理したのかを図 8.5 と図 8.6 に可視化した。各図の上側は FPSGD の、下側は dcMF の可視化結果を示し、ブロック分割数を変えて実験をおこなった。この際、スレッド数は 8 で固定した。全体の正方形が入力行列 R であり、スレッドは色を変えて表示している。FPSGD ではスレッドへのブロックの割り当てはランダムにおこなわれるため、スレッドがデータローカリティを無視してブロックを処理していることがこの図からわかる。一方、dcMF ではブロック分割数を増やしてもある程度ブロックが固まった状態で処理されていることがわかる。これは 7.3.2 で述べた通り、タスク木の根元付近からスチールされているからであると考えられる。しかし、MovieLens10M でも Netflix でも動的な負荷分散によるローカリティの無視はある程度起こっているようであり、図 8.2b で FPSGD と dcMF の差が小さくなった理由としては考えにくい。そのため、より定量的な調査が必要であると思われる。



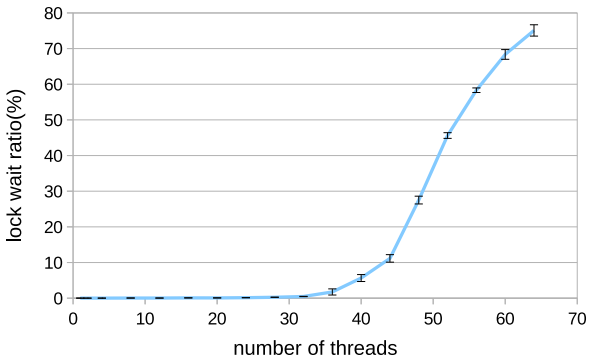
(a) MovieLens10M でのスケーラビリティ



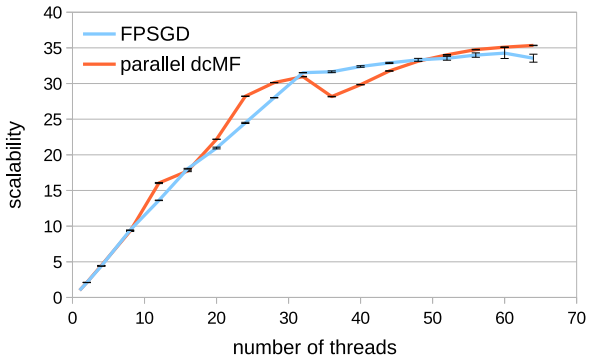
(b) MovieLens10M での FPSGD のロック待ち時間の割合



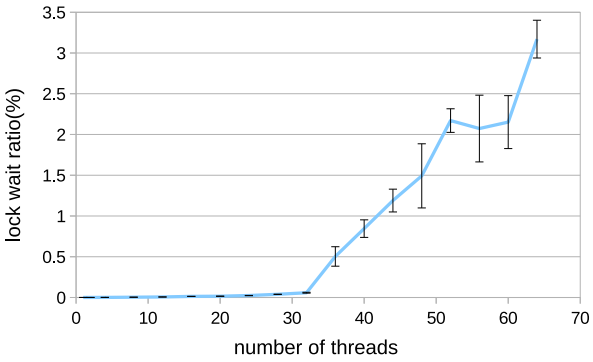
(c) Netflix でのスケーラビリティ



(d) Netflix での FPSGD のロック待ち時間の割合

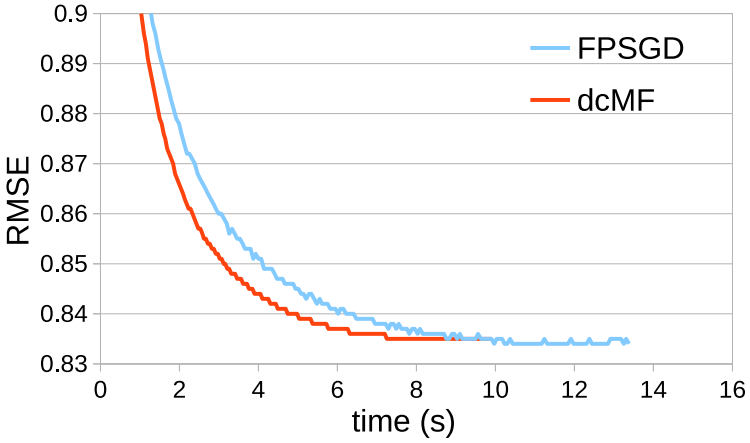


(e) Yahoo!Music でのスケーラビリティ

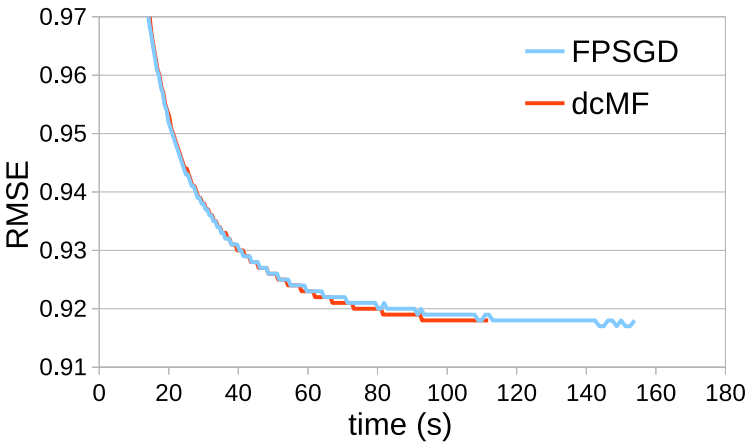


(f) Yahoo!Music での FPSGD のロック待ち時間の割合

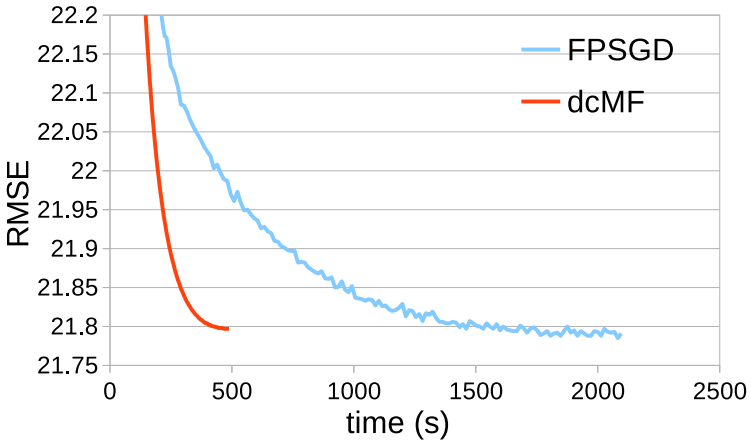
図 8.3. データセット毎のスケーラビリティの比較（左のカラム）と、FPSGD のロック待ちが占める割合（右のカラム）



(a) MovieLens10M での収束速度

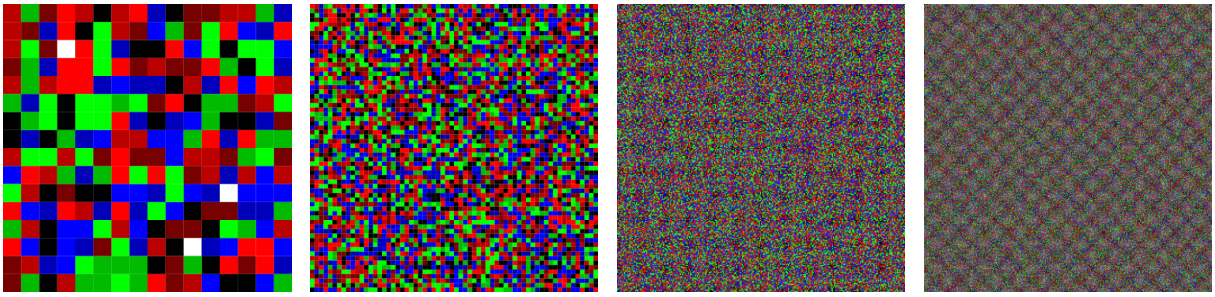


(b) Netflix での収束速度

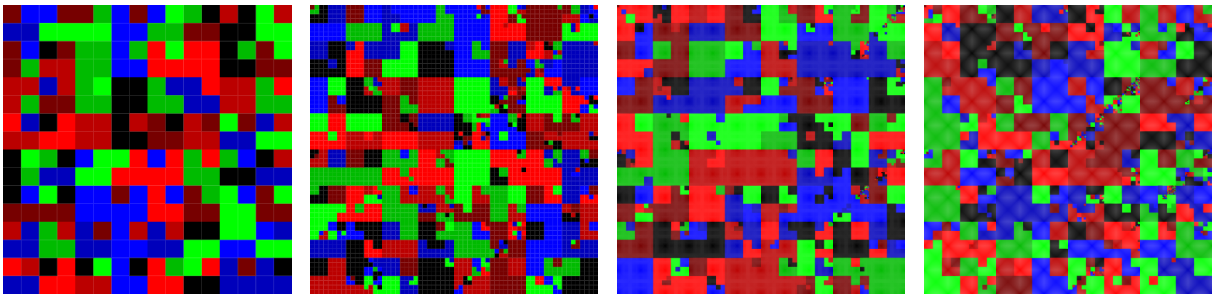


(c) Yahoo!Music での収束速度

図 8.4. データセット毎の収束速度の比較

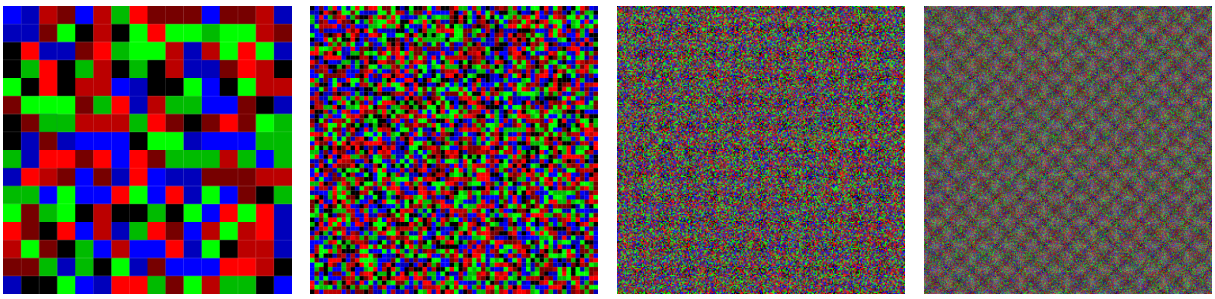


(a) FPSGD, 16 × 16 blocks (b) FPSGD, 64 × 64 blocks (c) FPSGD, 256 × 256 blocks (d) FPSGD, 512 × 512 blocks

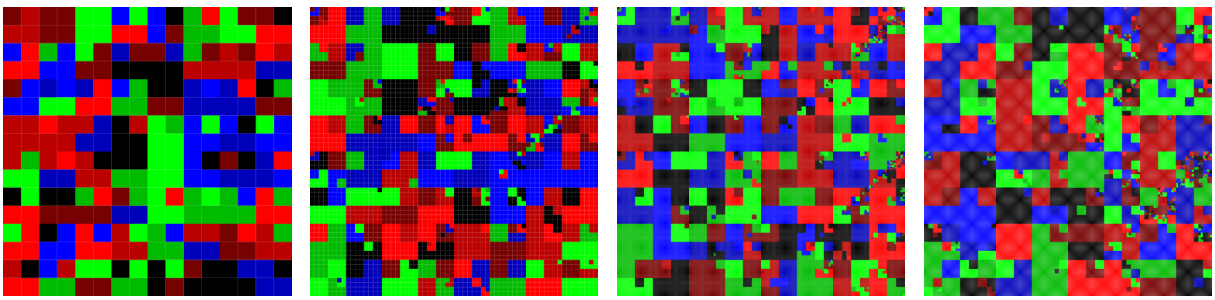


(e) dcMF, 16 × 16 blocks (f) dcMF, 64 × 64 blocks (g) dcMF, 256 × 256 blocks (h) dcMF, 512 × 512 blocks

図 8.5. Comparison of MovieLens10M



(a) FPSGD, 16 × 16 blocks (b) FPSGD, 64 × 64 blocks (c) FPSGD, 256 × 256 blocks (d) FPSGD, 512 × 512 blocks



(e) dcMF, 16 × 16 blocks (f) dcMF, 64 × 64 blocks (g) dcMF, 256 × 256 blocks (h) dcMF, 512 × 512 blocks

図 8.6. Comparison of Netflix

8.3　パラメータの変化による性能評価

本項では、タスク並列を用いた提案手法において、

- 学習率 γ の影響
- モデル行列の次元 k の影響

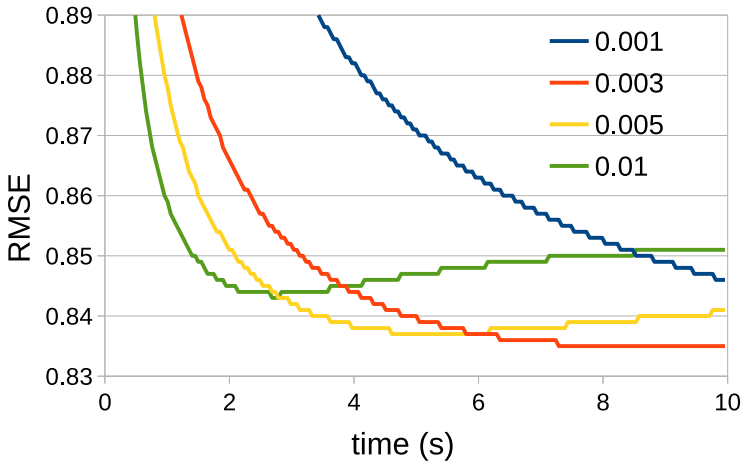
の収束性能への影響を調べた。なお、本項でのおこなった実験では、スレッド数は 32、ブロック分割数は 64×64 でおこなった。

8.3.1　学習率 γ の影響

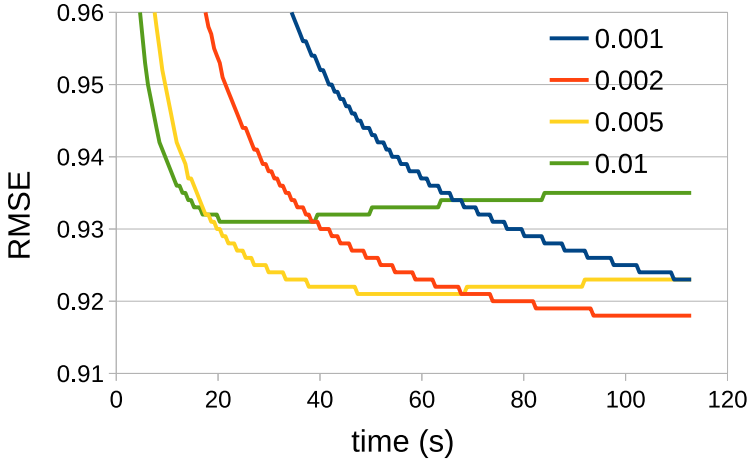
まず、学習率 k の収束性能への影響について調べた。結果を図 **??**に示す。各データセットにおいて、学習率の値を少しずつ変化させて学習時間に対する収束性能を計測した。この図より、学習率が小さいと収束までにかかる時間が長くなり、学習率が大きいとある特定の RMSE の値にたどり着くまでの時間は短くなるもののその後 RMSE が悪化するという挙動を見せる。この結果から、最適な学習率は表 8.2 に示した値であることがわかる。

8.3.2　モデル行列の次元 k の影響

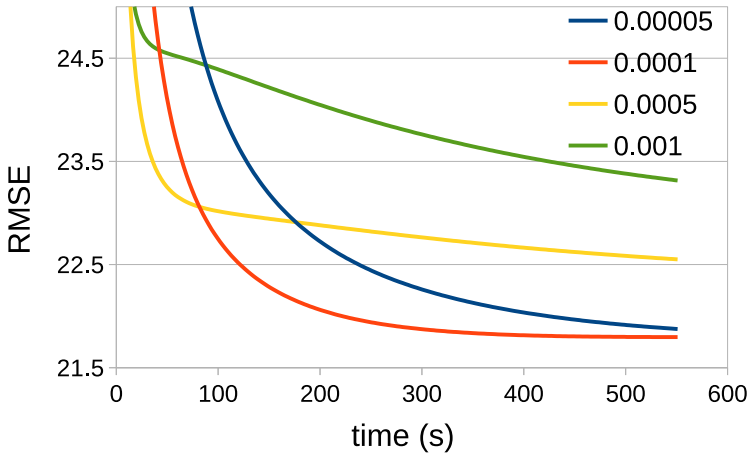
次に、モデル行列の次元 k の影響について調べた。結果を図 8.8 に示す。各データセットにおいて、k の値を少しずつ変化させて学習時間に対する収束性能を計測した。この図より、k が小さいと速く大きい RMSE へと収束し、k が大きいとゆっくりと小さい RMSE へと収束する様子が見て取れる。k の値を大きくする、つまりモデルの表現力を強くすると汎化性能が悪化する現象が一般には見られるが、正則化項を目的関数に付与していることからここでは最も大きい k でも汎化性能は下がっていない。この結果から、学習時間とモデルの精度はトレードオフの関係にあり、どちらをより追求するかは用いるアプリケーションとその用途によると言える。



(a) MovieLens10M での学習率による変化

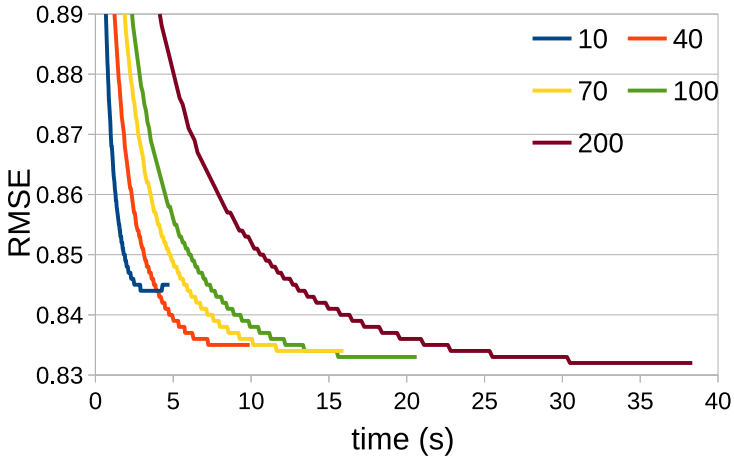


(b) Netflix での学習率による変化

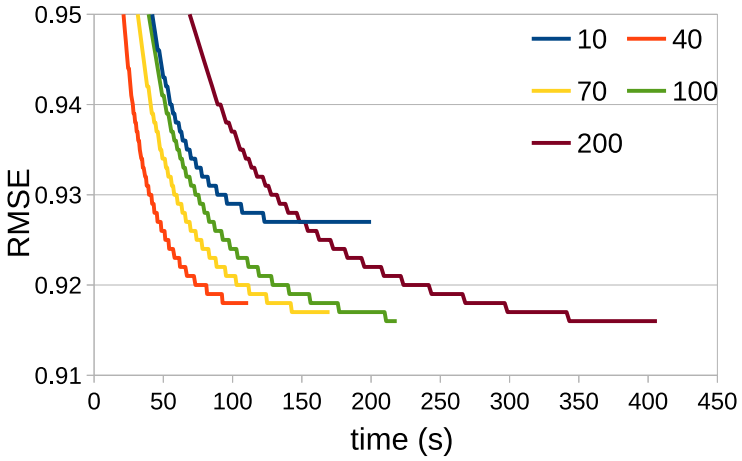


(c) Yahoo!Music での学習率による変化

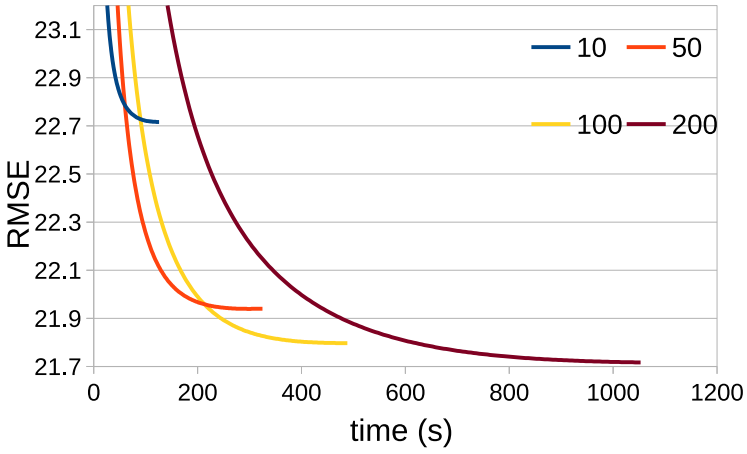
図 8.7. データセット毎の学習率による変化



(a) MovieLens10M での次元による変化



(b) Netflix での次元による変化



(c) Yahoo!Music での次元による変化

図 8.8. データセット毎の次元による変化

第9章　結論

9.1　まとめ

本研究ではタスク並列モデルと分割統治法を組み合わせて、最適化手法に SGD を適用した行列補完アルゴリズムを並列化する手法を提案した。行列を 2×2 のブロックに再帰的に分割し、各スレッドにブロックをあてがう。その際、同じ行・同じ列を持つブロックを同時にスレッドが処理しないようにタスク生成することで、更新の競合なしに学習モデルの更新をおこなうことができる。これにより、先行研究の FPSGD でロックの問題を解決することができ、理論的にキャッシュミス数の削減できることを示した。

評価実験として、FPSGD とスケーラビリティ、収束速度、L2 fill/writeback 数の 3 点において比較をおこなった。使用した全てのデータセットにおいて FPSGD のスケーラビリティを提案手法のそれが上回ったことを示し、さらにより早く収束することを示した。また、用いた 2 つのデータセットに関してキャッシュミス数は現象していることから、提案手法の有効性を示すことができた。

9.2　今後の展望

まず、スケーラビリティの評価では FPSGD の実験結果にロック待ち時間と L2 fill/writeback の影響が含まれており、スケーラビリティに及ぼす L2 fill/writeback の影響のみを論じることができていない。そのため、FPSGD のロック問題を改良した実装と提案手法を比較し、キャッシュミスの性能への影響を述べることが望ましい。

また、Netflix 使用時の fill/writeback 数の比較において、FPSGD と dcMF の間にそれほど顕著な差がなく理論値と一致しない結果となっている。これは Netflix データセットにおけるレーティング分布の偏りにより各スレッドに割り当てられる負荷にも偏りが生まれ、ワークスチーリングによりデータローカリティを意識したブロック処理ができなくなったからだと考えられるが、図 8.6 に示すタスクの処理状況はそれを説明できる結果とはなっていない。そのため、ブロックの分割数を変えた場合の各ブロックに含まれるレーティング数を調べるなどして説明力のある理由を示す必要がある。また、その理由を示した上で、提案手法のキャッシュミス数を可能な限り理論値に近づけるための方法についても考える必要がある。

また、本研究では共有メモリ環境における並列化について論じたが、レコメンドシステムで問題になるのは大量のユーザと大量のアイテムからなる大規模データであるため、分散環境への適用についても考えていきたい。

謝辞

まず、本研究を進めるにあたり研究の方向性、論文の添削など数多くのご指導とご助言を賜りました田浦健次朗准教授に深く感謝いたします。他大の他専攻から来たためにコンピュータサイエンスの知識に欠け理解力も非常に乏しい私に対し丁寧にご指導いただき、修論提出直前でしたが国内学会で発表することもできました。修士の 2 年間をこの研究室で過ごせたことを誇りにしたいと思います。

また、博士 4 年の中島潤先輩や博士 3 年の秋山茂樹先輩には発表練習の際に数々のご助言をいただき、発表の質を高めることができました。同期の早水君、フィン君とは日々の研究生活の大変さを分かち合いました。修士 1 年の島津君が茶軸キーボードの打鍵音と圧倒的タイピング速度の相乗効果を生み出す横で研究にいそしんだことも良い思い出です。同じく修士 1 年の石川君と山部君には円滑な研究室運営でお世話になりました。その他にも研究生活を送る上で多くの方々にお世話になりましたことを、この場を借りてお礼申し上げます。これまで支えてくださった家族にも感謝いたします。ありがとうございました。

発表文献

[1] Yusuke Nishioka, Kenjiro Taura. Scalable Task-Parallel SGD on Matrix Factorization in Multicore Architectures. *ACSI2015*, Tsukuba, 2015/1. (accepted)

[2] Yusuke Nishioka, Kenjiro Taura. Scalable Task-Parallel SGD on Matrix Factorization in Multicore Architectures. *4th International Workshop on Parallel and Distributed Computing for Large Scale Machine Learning and Big Data Analytics.(IPDPS2015)*, Hyderabad, 2015/5. (submitted)

参考文献

[1] Netflix Prize. http://www.netflixprize.com/rules.

[2] The Music Genome Project. http://www.pandora.com/about/mgp.

[3] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, pp. 43–52, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[4] Xi Chen, Xudong Liu, Zicheng Huang, and Hailong Sun. Regionknn: A scalable hybrid collaborative filtering algorithm for personalized web service recommendation. In *Web Services (ICWS), 2010 IEEE International Conference on*, pp. 9–16, July 2010.

[5] Mark Claypool, Anuja Gokhale, Tim Miranda, Pavel Murnikov, Dmitry Netes, and Matthew Sartin. Combining content-based and collaborative filters in an online newspaper. In *Proceedings of ACM SIGIR workshop on recommender systems*, Vol. 60. Citeseer, 1999.

[6] Paul J Drongowski and Boston Design Center. Basic performance measurements for amd athlon 64, amd opteron and amd phenom processors. *AMD whitepaper*, Vol. 25, , 2008.

[7] F.Niu, B.Recht, and C.Re. Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. *NIPS'11*, pp. 693–701, 2011.

[8] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, Vol. 35, No. 12, pp. 61–70, December 1992.

[9] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl. An algorithmic framework for performing collaborative filtering. In *Proceedings of the 22Nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '99, pp. 230–237, New York, NY, USA, 1999. ACM.

[10] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, Vol. 22, No. 1, pp. 5–53, January 2004.

[11] Yifan Hu, Y. Koren, and C. Volinsky. Collaborative filtering for implicit feedback datasets. In *Data Mining, 2008. ICDM '08. Eighth IEEE International Conference on*, pp. 263–272, Dec 2008.

[12] I.Pilaszy, D.Zibriczky, and D.Tikk. Fast ALS-based Matrix Factorization for Explicit and Implicit Feedback Datasets. *Proceedings of the 4th ACM conference on Recommender systems*, pp. 71–78, 2010.

[13] J.Nakashima. MassiveThreads: A Lightweight Thread Library for High Productivity Languages. http://code.google.com/p/massivethreads/, 2012.

[14] Yehuda Koren. The bellkor solution to the netflix grand prize. *Netflix prize documentation*, Vol. 81, , 2009.

[15] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, No. 8, pp. 30–37, 2009.

[16] L.Battou. Online Algorithms and Stochastic Approximations. *Online-learning in neural networks*, Vol. 34, No. 4, 1998.

[17] G. Linden, B. Smith, and J. York. Amazon.com recommendations: item-to-item collaborative filtering. *Internet Computing, IEEE*, Vol. 7, No. 1, pp. 76–80, Jan 2003.

[18] James MacQueen, et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1, pp. 281–297. Oakland, CA, USA., 1967.

[19] Koji Miyahara and Michael J Pazzani. Improvement of collaborative filtering with the simple bayesian classifier 1. 2002.

[20] Arvind Narayanan and Vitaly Shmatikov. How to break anonymity of the netflix prize dataset. *arXiv preprint cs/0610105*, 2006.

[21] MarkO 'Connor, Jon Herlocker. Clustering items for collaborative filtering. In *Proceedings of the ACM SIGIR workshop on recommender systems*, 第 128 巻. Citeseer, 1999.

[22] Arkadiusz Paterek. Improving regularized singular value decomposition for collaborative filtering. In *Proceedings of KDD cup and workshop*, Vol. 2007, pp. 5–8, 2007.

[23] Michael J Pazzani. A framework for collaborative, content-based and demographic filtering. *Artificial Intelligence Review*, Vol. 13, No. 5-6, pp. 393–408, 1999.

[24] Michael J Pazzani and Daniel Billsus. Content-based recommendation systems. In *The adaptive web*, pp. 325–341. Springer, 2007.

[25] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, CSCW '94, pp. 175–186, New York, NY, USA, 1994. ACM.

[26] R.Gemulla, P.J.Haas, E.Nijkamp, and Y.Sismanis. Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent. *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, KDD'11*, pp. 69–77, 2011.

[27] Gerard Salton and Michael J McGill. *Introduction to modern information retrieval*. 1983.

[28] Badrul M Sarwar, George Karypis, Joseph Konstan, and John Riedl. Recommender systems for large-scale e-commerce: Scalable neighborhood formation using clustering. In *Proceedings of the fifth international conference on computer and information technology*, Vol. 1. Citeseer, 2002.

[29] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pp. 285–295, New York, NY, USA, 2001. ACM.

[30] Upendra Shardanand and Pattie Maes. Social information filtering: Algorithms for automating “word of mouth”. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pp. 210–217, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[31] Xiaoyuan Su, Russell Greiner, Taghi M Khoshgoftaar, and Xingquan Zhu. Hybrid collaborative filtering algorithms using a mixture of experts. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, pp. 645–649. IEEE Computer Society, 2007.

[32] Xiaoyuan Su and Taghi M. Khoshgoftaar. A survey of collaborative filtering techniques. *Adv. in Artif. Intell.*, Vol. 2009, pp. 4:2–4:2, January 2009.

[33] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. Matrix factorization and neighbor based algorithms for the netflix prize problem. In *Proceedings of the 2008 ACM conference on Recommender systems*, pp. 267–274. ACM, 2008.

[34] K. Yu, A. Schwaighofer, V. Tresp, Xiaowei Xu, and H.-P. Kriegel. Probabilistic memory-based collaborative filtering. *Knowledge and Data Engineering, IEEE Transactions on*, Vol. 16, No. 1, pp. 56–69, Jan 2004.

[35] Zhi-Dan Zhao and Ming-Sheng Shang. User-based collaborative-filtering recommendation algorithms on hadoop. In *Knowledge Discovery and Data Mining, 2010. WKDD '10. Third International Conference on*, pp. 478–481, Jan 2010.

[36] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management*, pp. 337–348. Springer, 2008.

[37] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel sgd for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pp. 249–256, New York, NY, USA, 2013. ACM.