複数レイヤにおける多重並列化の自動調停に関する検討

産業技術総合研究所
人工知能研究センター

2016年3月1日

穐山 空道, 小川 宏高

背景

- 人工知能, ビッグデータ処理の隆盛
  - 画像/音声認識, 情報推薦, 言語理解, 自動運転, …
- データ, モデルの巨大化
  - 行列演算: 100万次
  - グラフ処理: 1億ノード
  - ニューラルネット: 100層

負荷の高いワークロードを
手軽に実行する要求が高まる

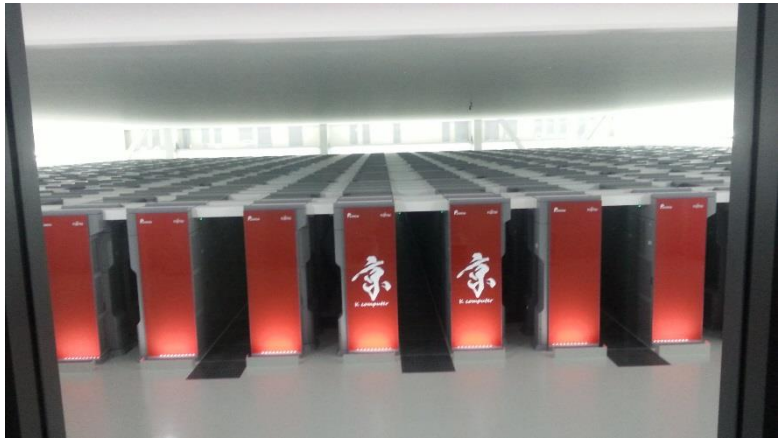
「手軽」とは

HPCユーザ

- 下層のシステムを熟知
- 高速化の技能・余裕がある

→ 細粒度の高速化が可能

  - ループアンローリング
  - ブロック化



「手軽」なユーザ

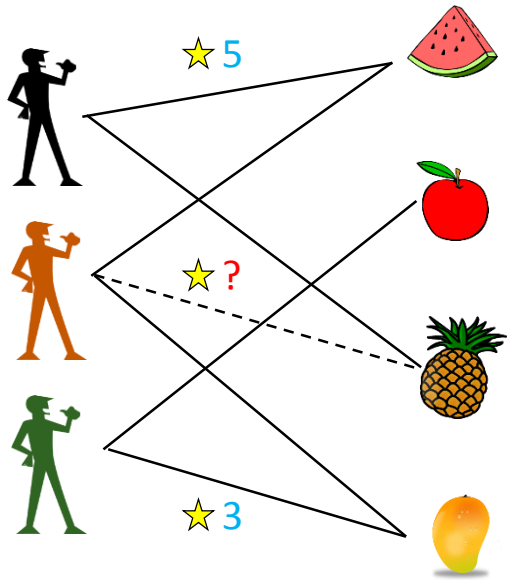
- 例：AI研究・開発者
- 計算機やOSの知識がない
- アルゴリズムの正しさや精度にのみ関心

→ 直感的記述が望ましい

$$\frac{1}{2}\frac{\partial f}{\partial u_{ki}} = 0, \quad \forall i,k$$
$$\Rightarrow \sum_{j \in I_i} (\mathbf{u}_i^T \mathbf{m}_j - r_{ij}) m_{kj} + \lambda n_{u_i} u_{ki} = 0, \quad \forall i,k$$
$$\Rightarrow \sum_{j \in I_i} m_{kj} \mathbf{m}_j^T \mathbf{u}_i + \lambda n_{u_i} u_{ki} = \sum_{j \in I_i} m_{kj} r_{ij}, \quad \forall i,k$$
$$\Rightarrow \left(M_{I_i} M_{I_i}^T + \lambda n_{u_i} E\right) \mathbf{u}_i = M_{I_i} \; R^T(i, I_i), \quad \forall i$$
$$\Rightarrow \mathbf{u}_i = A_i^{-1} V_i, \quad \forall i$$

Zhou *et al.*: “Large-scale Parallel Collaborative Filtering for the Netflix Prize”

具体例: 情報推薦



ユーザの嗜好

$$\begin{pmatrix} 5 & ? & 3 & ? \\ 4 & ? & ? & 2 \\ ? & 5 & ? & 3 \end{pmatrix}$$

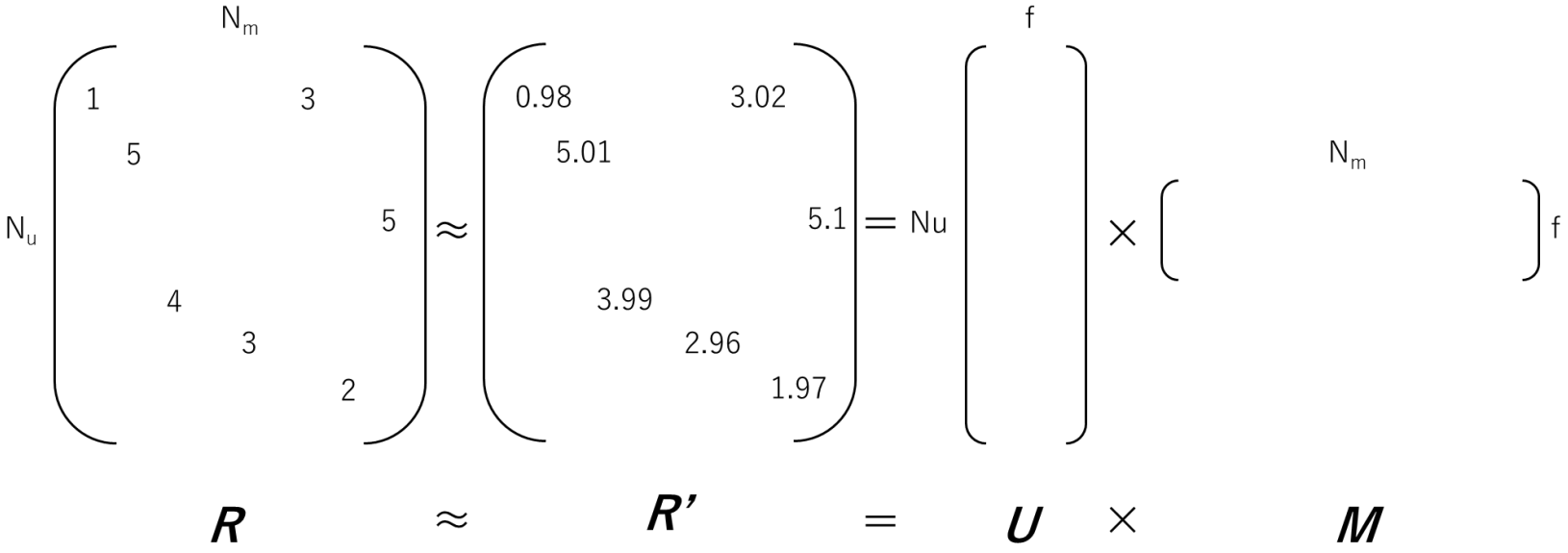
行列表現

- 既知のデータから “?” を推定
- Netflix data setでは 17K users × 480K movies

→ 巨大行列の手軽・高速な処理が必要

Matrix Factorization (MF)

- 大きな行列を小さな行列に分解し近似
- $R[N_u, N_m] \approx U[N_u, f] \times M[f, N_m]$
  - fは特徴数, $f << N_u$, $f << N_m$
  - Rはsparse（値がない要素が多数）

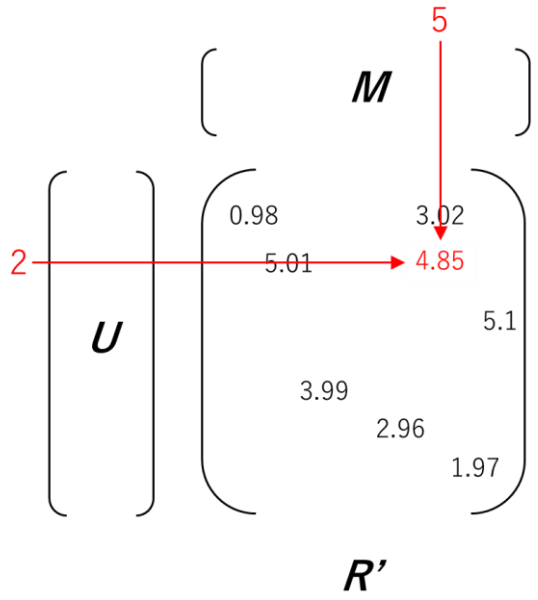


MFによる情報推薦

- R=(r_{ij}): ユーザiのムービーjに関する評価

→ U, Mはユーザ、ムービに関する特徴を抽出

- R’ (= U×M)のうちRに含まれない要素は評価の推測値（R’はRよりも要素数が多い）



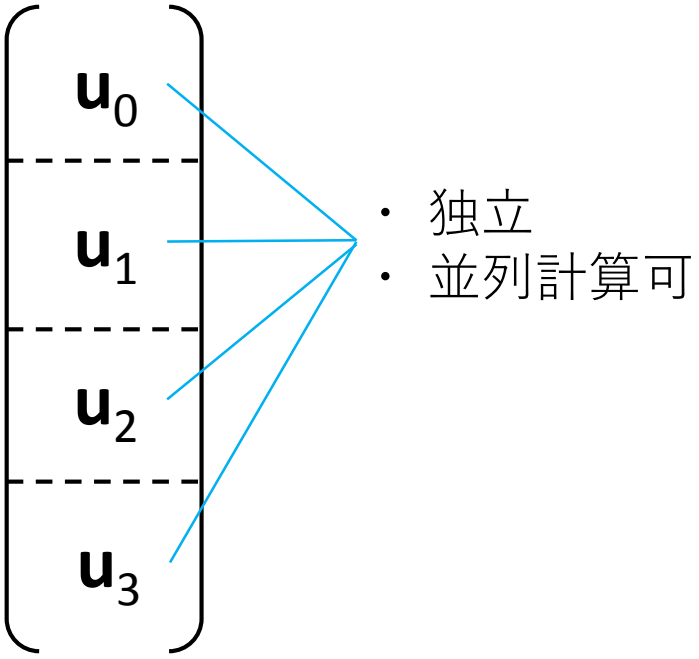
ユーザ2のムービー5に関する評価の推測値は4.85

→ ムービー5をユーザ2に推薦

並列化: 行単位とLU分解

行単位並列化

Uの各行（$\mathbf{u}_i$）を並列計算可
（***M***も同様, 詳細は文献[12]）



- 独立
- 並列計算可

各行内での並列化

$\mathbf{u}_i$の計算は主に行列のLU分解
LU分解は並列計算可

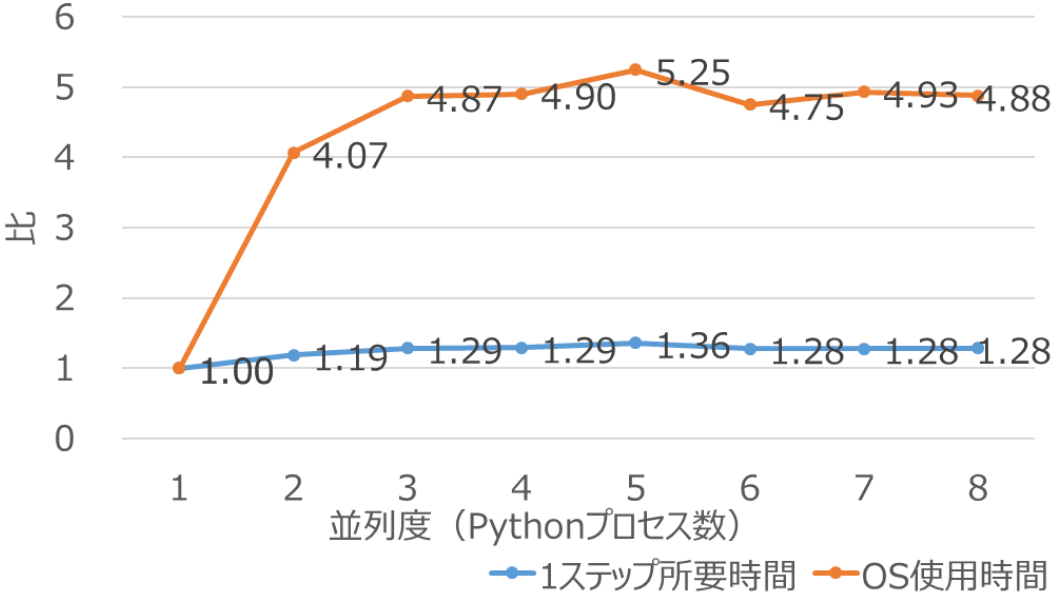
$$\mathbf{u}_i = \boldsymbol{A}_i^{-1}\boldsymbol{V}_i$$

*1 A, VはMとRから定まる
*2 A^{-1}の計算にLU分解が必要

行単位と各行内での多重並列化が発生

多重並列化の観測

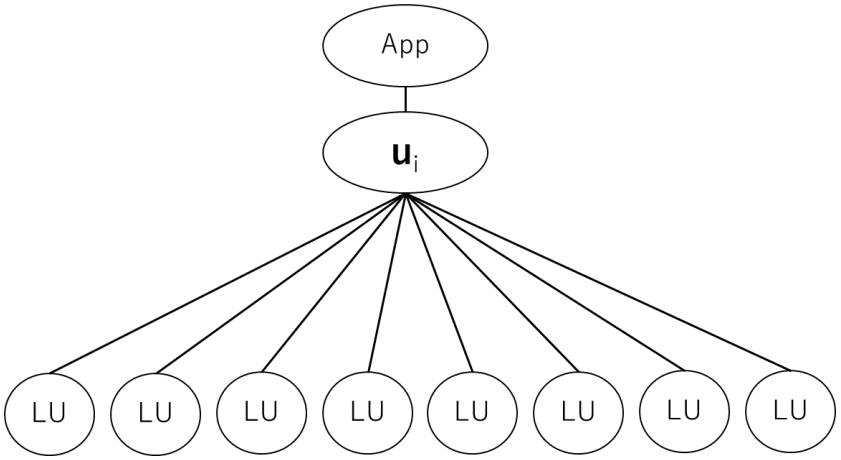
- Python, Intel MKL を用いMFを実装
  - 行単位並列化には `multiprocessing` を使用
  - 行内の並列化はIntel MKLが自動で行う
  - 実データから得た8000次の行列
  - 8コア（4コア Xeon ×2），16 GBメモリ



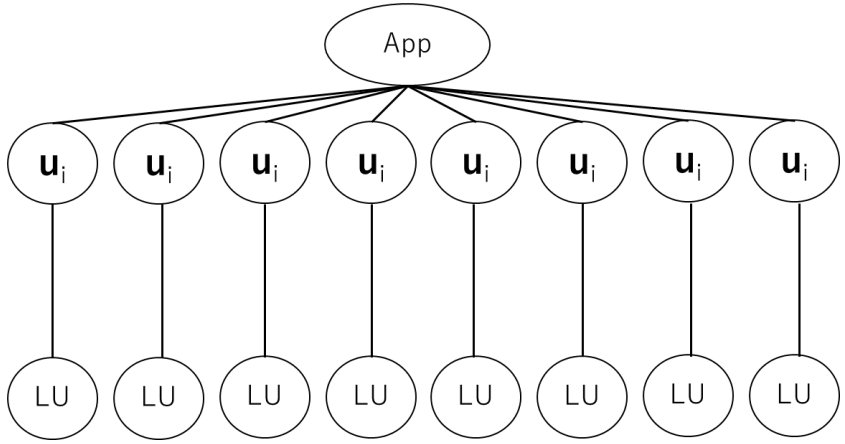

```
python3.5       30331      -  123
-                    - 30331 15.1
-                    - 30342  0.0
-                    - 30343 14.1
-                    - 30344 14.5
-                    - 30345 14.9
-                    - 30346 14.0
-                    - 30347 16.4
-                    - 30348 16.9
-                    - 30349 16.6
python3.5       30332      -  113
-                    - 30332 13.3
-                    - 30350  0.0
-                    - 30351 14.6
-                    - 30352 16.0
-                    - 30353 13.6
-                    - 30354 14.7
-                    - 30355 13.2
-                    - 30356 16.0
-                    - 30357 14.8
```

1プロセスにつき8スレッド

静的割り当ての問題点



行ごとの並列度: 1
LU分解の並列度: 8
→ スパースな行に対し非効率
（データの非一様性）



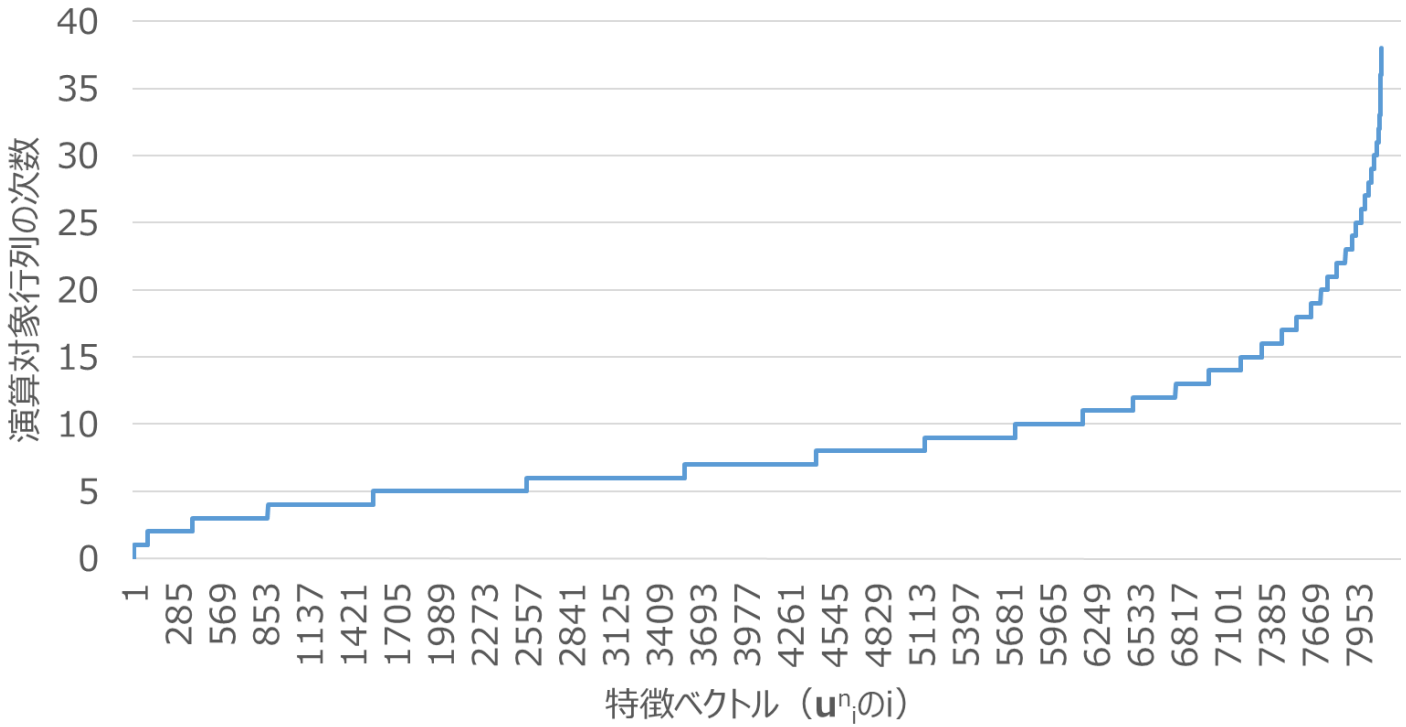
行ごとの並列度: 8
LU分解の並列度: 1
→ 行間の依存関係に弱い

処理対象データの非一様性

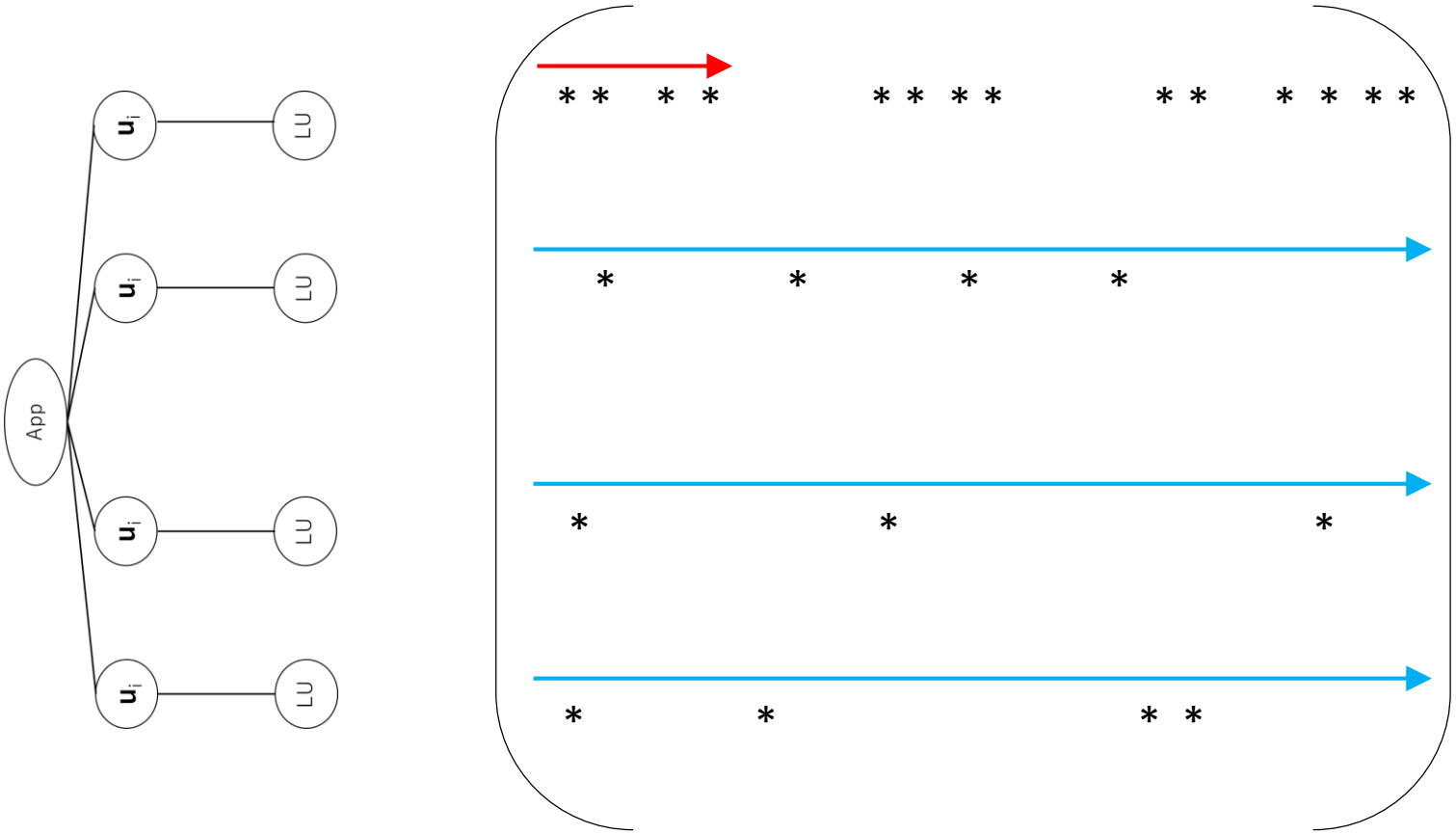
- 実世界データは高度に非一様
  - Twitterのフォロー数
  - Webページのリンク数

指数的分布
少数のノードは入/出次数が非常に多く
ほとんどのノードは次数ほぼゼロ

8000次実データをMFする際のLU分解される行列サイズ



行間の依存関係



アプリ上行間依存性があると非効率 **[検討中]**

解決手法への要求

「手軽な」ユーザが使えること

1. 幅広い言語とライブラリの組に適用可
   - Python + BLAS, ...
   - Ruby + ...

   → 自動並列化ライブラリを作ってもダメ
2. ユーザが触れるのは最上位レイヤのみ
   - アルゴリズムの可読性を崩さない
   - アノテーション: PythonはOK?ライブラリは不可能

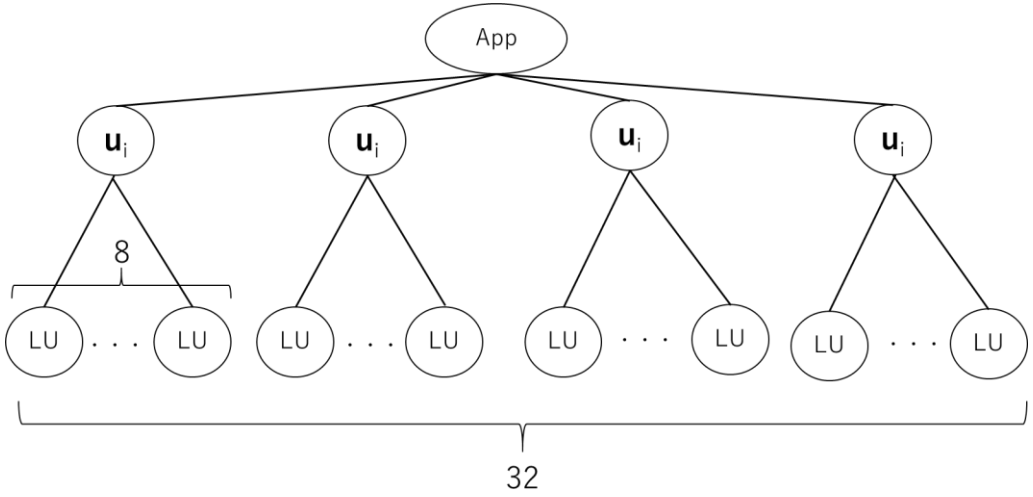
OS/ランタイムで自動的に調停するべき

スケジューラによるアプローチ

多めに並列化しスケジューラでオーバーヘッド削減

- オーバーヘッドがなければオーバーコミットはOK
- Under-utilizationはしたくない

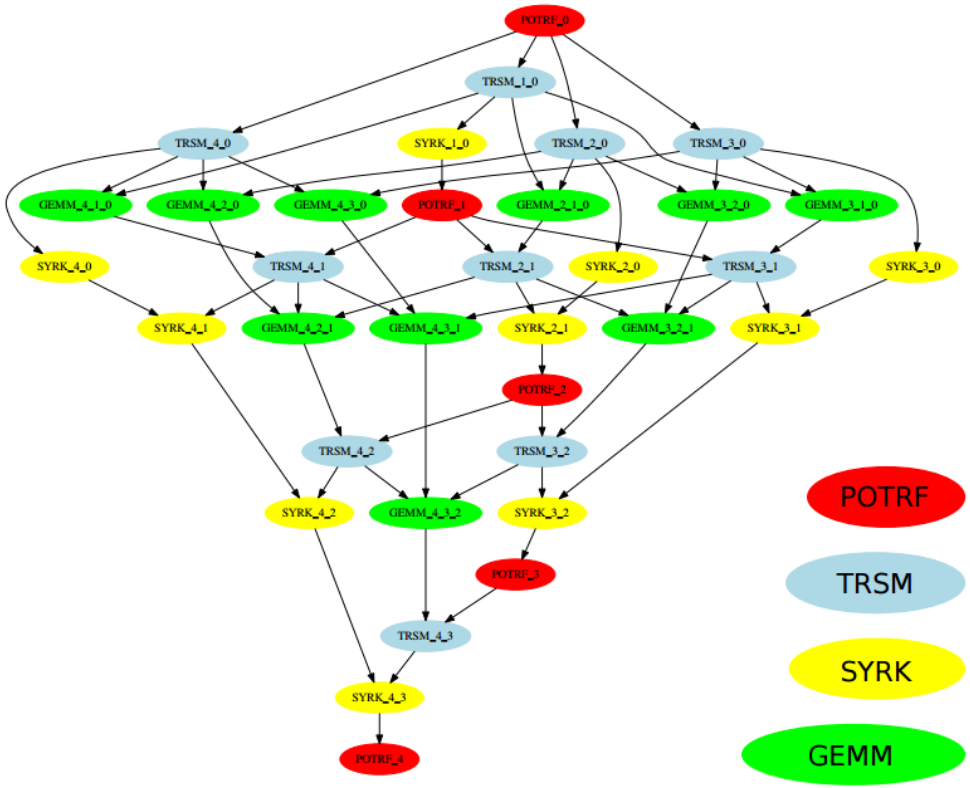
1. 多めに並列化(行ごと:4, LU分解:8)



2. 有効な並列度を動的/静的に検出
3. スレッドのグルーピング（と合成?）

有効な並列度の検出 (1/2)

行列を5x5のタイルごとにコレスキー分割する際の計算カーネル間の依存関係
“Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms”, *IPDPS workshop*, 2015



計算カーネル間依存性により、行列を処理中の有効な並列度は動的に変化

→ N次行列だからといって常にN並列可能ではない

有効な並列度の検出 (2/2)

- 前ページのように計算カーネルが見えていれば比較的容易
  - 計算カーネルをどうスケジュールするかという研究は多く存在
- 今回の条件（ライブラリspecificでない）では計算カーネルを見つけるのが困難
  - 既存のスケジューリング研究でデータ依存関係の抽出はよく行われている

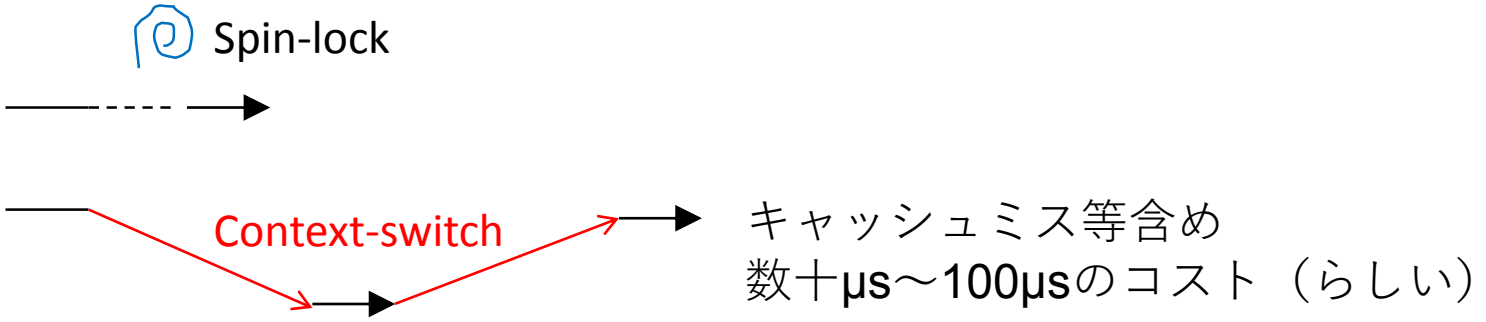
スレッドのグルーピング (1/3)

- 単純に少しだけオーバーコミットすれば待っているスレッドは寝ていてくれるのでは？

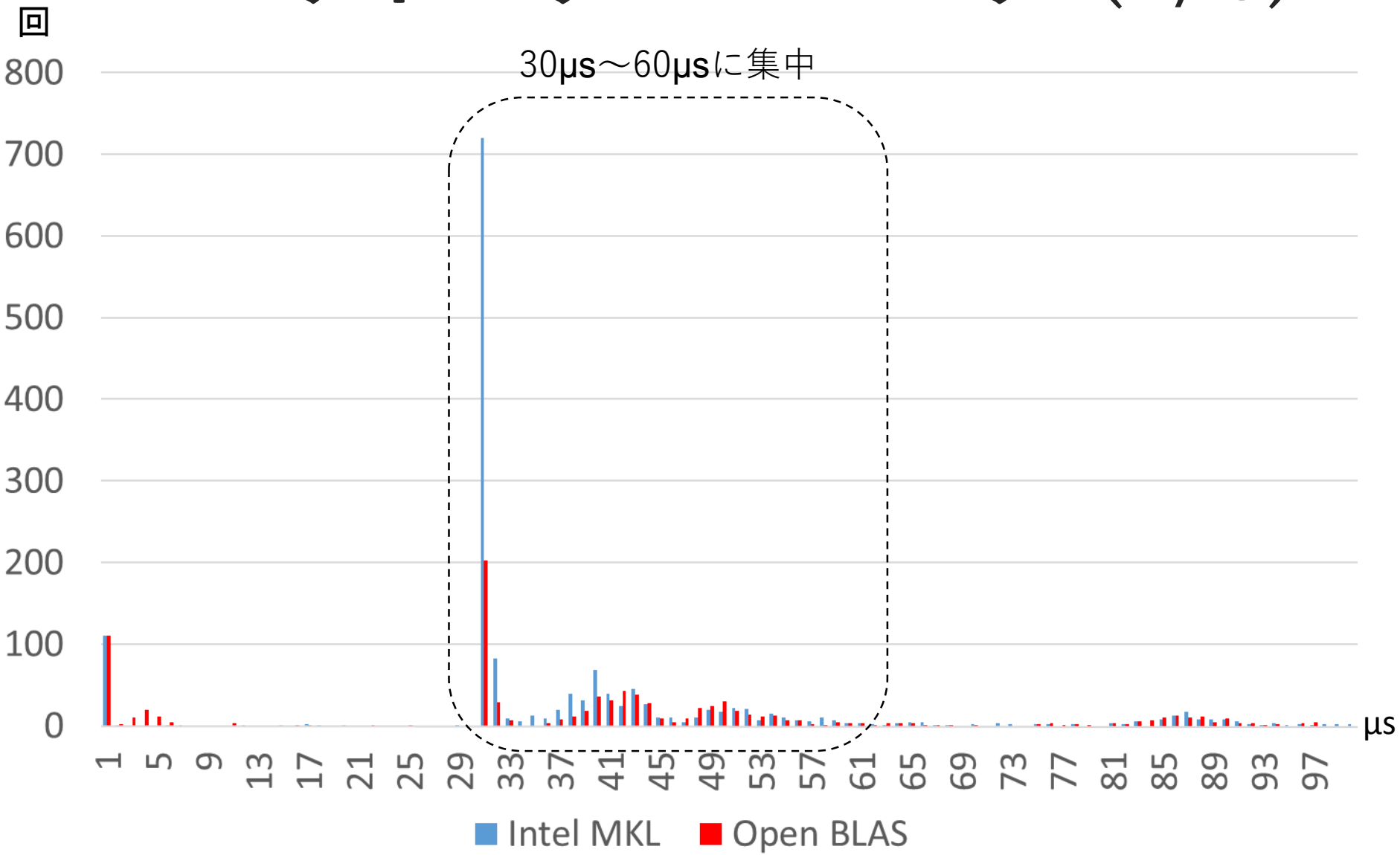
→ 計算の特性によりそうはならず

- Intel MKL, OpenBLASでは**スピンロック**を用いて待ち合わせ（sleepしない）

∵ごく短い待ちではスピンロックの方が効率的



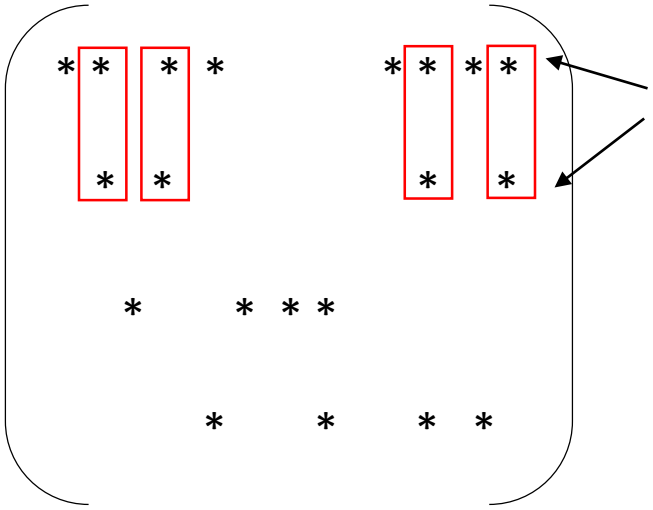
スレッドのグルーピング (2/3)



スレッドのグルーピング (3/3)

- 「スピンロック中のスレッドを後回しにする」は単純には損
  - コンテキストスイッチのオーバーヘッド（**Cs**）が平均待ち時間を上回るため

→ **Cs**が減るようなスケジューリング



同じ列に対するアクセスが多い
→ この二つはコンテキストスイッチのコストが小さいかも？ **[検討中]**

まとめ

- AI/ビッグデータの隆盛により高負荷ワークロードを手軽に実行する要求が高まる

- 課題
  - 「手軽」なユーザには細粒度チューニング不能
  - データの非一様性と多重並列化により、並列度を動的に調停する必要がある

- 「オーバーヘッドの少ないオーバーコミットをし、under utilizationしない」手法を検討中