Vol.2016-OS-136 No.12
2016/3/1

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

穐山 空道[1,a)]　小川 宏高[1]

概要：関係性抽出のためのグラフ処理や言語処理のための行列演算など、人工知能やビッグデータが注目されるに従いマルチコア CPU を活用した並列計算を手軽に行う需要が高まっている。こうしたユーザはアプリケーションに高い性能を要求する一方、OS やハードウェアには必ずしも精通しないため性能のチューニングは困難である。例えばユーザがマシンスペックから予測される最大までスクリプト言語レベルで並列化し以降のチューニングを行わないと、下層で呼び出されるライブラリも独立に並列化を行っており並列化度が大きすぎて性能が劣化することが起こり得る。本稿ではこの問題の具体例、解決のための課題及び OS のスケジューラ改変による解決手法の初期検討について述べる。

1. 序論

人工知能やビッグデータが注目されるに従い、機械学習などの計算パワーを要求するワークロードを手軽に実行する需要が高まっている。たとえば数値解析を行うための Python ライブラリである numpy や scipy では高速フーリエ変換や線形代数演算がサポートされ、その上で scikit-learn、NLTK、gensim など数多くの機械学習プラットフォームが提供されている。

これらの数値解析ライブラリや機械学習プラットフォームのユーザは人工知能やビッグデータに関連する分野の研究者や開発者であり、作成するアプリケーションの高速な実行速度を要求する [*1]。一方、これらのユーザにとって実行速度のチューニングを行うことは、それが一義的な目的ではないことおよび OS やハードウェアに必ずしも精通しないことから非常に困難である。本稿では**ユーザ**を下記のような特性を持つ人であると定義する。

(1) 対象とする処理について、できるだけ速く実行が完了することを望む。（ただし Stocastic Computing のような処理結果の一意性を犠牲にして高速化するケースまでは考慮しない。）
(2) 最上位の言語（Python や R がよく用いられる）のレベルで並列化や遅延評価などの効率化を行う知識を有するが、下位のライブラリの実装まで立ち入って高速化を行う経験も動機もない。

[1] 国立研究開発法人 産業技術総合研究所 人工知能研究センター
Artificial Intelligence Research Center, National Institute of Advanced Industrial Science and Technology (AIST)
[a)] s.akiyama@aist.go.jp

[*1] 人工知能やビッグデータの領域では**性能**は認識等の精度を意味することがあるため、本稿では明示的に**実行速度**と称する。

このようなユーザにとって実行速度のチューニングが困難な原因に、**多重並列化**と**処理対象の非一様性**があげられる。多重並列化とは、複数のレイヤやコンポーネントで並列化が独立かつ多重に行われることを指す。例えば、スクリプト言語レベルでユーザが指定した並列化と、その下で呼ばれる C 言語で記述されたライブラリが行う並列化が多重に起こり得る（ 2.2 節に具体例を示す）。処理対象の非一様性とは、人工知能やビッグデータ処理で対象とする実世界から得られたデータは一様な構造を示さないことを指す。例えば、Twitter 上のユーザのつながりを表したグラフでは各ノードの次数は指数的に分布することがよく知られている [5]。また巨大な疎行列を小さな密行列に分解する処理では、ライブラリの処理対象となる行列の次数が指数的に分布する（ 2.3 節に詳細を示す）。

多重並列化と処理対象の非一様性によって、次の二点がアプリケーションの性能を劣化させる。

(1) 多重並列化のためにユーザが求める並列度の割り当てを実現できない
(2) 処理対象の非一様性により、一つのデータセットの処理中に有効な並列度が一定ではない

多重並列化は下位のライブラリがユーザの指定とは独立に行い、前述のユーザに関する仮定により本稿で対象とするユーザには制御できない。その結果、例えばユーザが最適な並列度の配分（例えば Python レベルで 4、下位のライブラリレベルで各 2 の計 8）をデータ間の依存関係やアルゴリズムの性質から知っていても、それを実現することできない。また処理対象のグラフや行列が一様でないことはその処理に有効な最大並列度も一様ではないことを意味する。例えば全体を 4 並列、データを分割した各部分を 2 並列で記述

すると、一部のデータは並列化が有効でなく無駄な CPU 時間が発生しうる。しかしユーザはアプリケーションの精度や正しさに興味があるため、実装するアルゴリズムや数式を直感的に記述できることを望む（例えば行列演算を直感的に記述できるライブラリ [4] が人気である）。したがって高性能計算（HPC）で行われるような細粒度のループ展開、ブロック化、帯領域二重計算などの効率化手法を本稿で対象とするユーザが適用するのは不可能であり、その結果並列度が大きすぎて性能が劣化しうる。

そこで本稿では、OS のスケジューラを改変し多重並列化によって並列度が上がりすぎた場合に実行を自動的に調停することで、最適な実行速度を得ることを目指す。本稿の対象ユーザは慣れ親しんだ言語から任意のライブラリの組み合わせを呼び出すため、特定の自動チューニングコンパイラ、ランタイム、フレームワーク等の利用は難しい。一方、利用する言語やライブラリにかかわらず OS からはスレッドやプロセスとして同一に観測、制御可能であるため、スケジューラレベルでの解決が適すると考える。

本稿の構成を以下に示す。第 2 章で多重並列化、処理対象の非一様性について、具体的な機械学習アルゴリズムを用いて詳細に説明する。第 3 章では課題の解決手法に求められる要求、提案手法の一検討およびその実現のための技術的課題を述べる。本稿では研究の初期検討であるため提案手法の内容よりも技術的課題に重点を置いて説明する。第 4 章で関連研究について説明し、第 5 章で本稿をまとめる。

2. 多重並列化と処理対象の非一様性

2.1 具体的アプリケーション：Matrix Factorization

多重並列化と処理対象の非一様性が発生する具体例として、Matrix Factorization を考える。Matrix Factorization は人工知能やビッグデータ処理でよく用いられ、巨大な疎行列を小さな密行列二つの積で近似し巨大行列の特徴を抽出する手法である。

Matrix Factorization の概念図を**図 1** に示す。いま顧客 i のつけたムービー j の評価の集合を行列 $R = (r_{ij})$ で表現する。顧客とムービーはそれぞれ多数ある一方、ある顧客が評価を付けるムービーの数およびあるムービーに付けられる評価の数は少数、すなわち R は巨大かつ疎である。顧客とムービーの数をそれぞれ N_u, N_m で表し、R は $N_u \times N_m$ 次行列である。このとき、R を特徴次元数 f で Matrix Factorization するとは、$R' = U \times M$ と R の誤差が最小となる $N_u \times f$ 次元密行列 U と $f \times N_m$ 次元密行列 M の組を見つけることである。ただし、R' と R の誤差を R に評価が存在する i, j の組についての $|r_{ij} - r'_{ij}|$ の合計と定める。

Matrix Factorization のよく知られた応用は情報推薦（協調フィルタリング）である。いま $i = k, j = l$ を R 上に評価が存在しないユーザとムービーの組、すなわちユーザ k はムービー l を評価していないとする。U と M は密行列であるから、r_{kl} は存在せずとも r'_{kl} は何らかの値を持つ。U と M の全ての要素は R' と R の誤差を最小化するよう定めたので、これらは R に含まれる特徴を抽出し低次元へ圧縮したものと言える。具体的には、各行列はユーザの好みの特徴やムービーの評価の特徴を表現し、r'_{kl} はユーザ k のムービ l に対する評価の推定値になっている。実際に、ムービー配信事業者である Netflix のデータセットにおいて既知の評価値の行列を Matrix Factorization することで未知の評価を推定できると示されている [12]。協調フィルタリングではこの推定値を用いてユーザがまだ視聴していないが興味を持ちそうなムービーを推薦する。

2.2 多重並列化

本節では Matrix Factorization の実装において多重並列化が発生することを説明する。本稿で対象とするユーザはアルゴリズムの記述容易性や可読性を重視するため、Python を用いた実装を考える。実際、データマイニングやビッグデータ分析に関する情報サイトである KDnuggets におけるユーザ投票では、“What programming/statistics languages you used for an analytics / data mining / data science work in 2013?” という質問に対し Python が R に次いで 2 位であった [8]。

Matrix Factorization の実装では、(1) 求める行列の要素ベクトルごとの並列計算と (2) 各要素ベクトルの計算内で現れる一般的な行列計算での並列化が多重で可能である。求める行列の要素ベクトルごとの並列化は、Matrix Factorization が Bulk Synchronous Parallel モデルのステップ解法で解けることによる。あるステップ n での U の推定値 U^n のうち i 番目のユーザに関する（長さ f の）ベクトル $\mathbf{u}_i^n$ は、R と前ステップの M の推定値 M_{n-1} のみに依存して計算可能であり、他のユーザに関するベクトル $\mathbf{u}_k^n$ $(k \neq i)$ には依存しない。従って $\mathbf{u}_i^n$ は i について並列に計算可能である。M_n の各要素ベクトル $\mathbf{m}_j^n$ についても同様である。各要素ベクトルの計算内で現れる行列計算の並列化は、アルゴリズムの本質とは関わりが少ない行列の一般的な演算でも並列化が可能なことによる。例えば行列の LU 分解は文献 [12] の Matrix Factorization アルゴリズムでも多数現れるが [*2]、LU 分解は並列化手法が広く知られている。

図 2 に Matrix Factorization を Python と BLAS ライブラリで実装した際の並列度と実行速度の関係を示す。分解した行列は実世界から得られた約 8000 次元の行列で、実行に用いたマシンは Intel Xeon E5-2603（4 コア）を 2 ソケット分で 8 コア、32GB メモリ、Debian GNU/Linux 8.2

[*2] LU 分解は逆行列計算を数値的安定に求める手法であるため式中の逆行列の数だけ LU 分解が発生する。

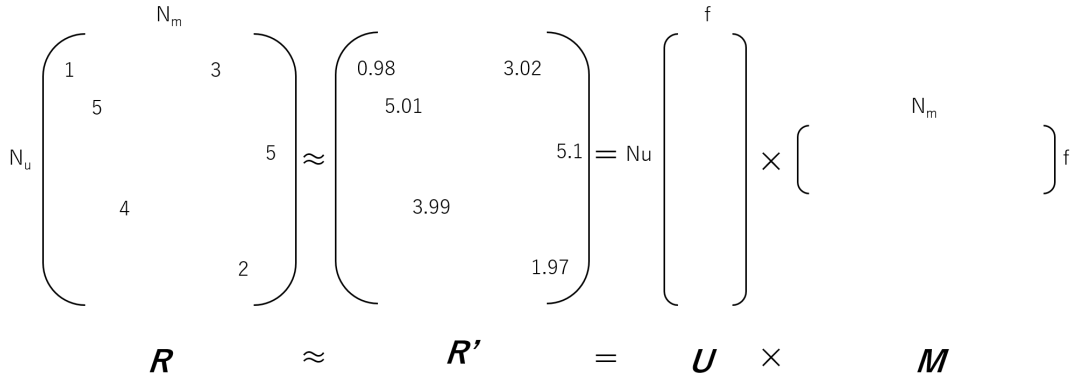

図 1 Matrix Factorizatio の概念図

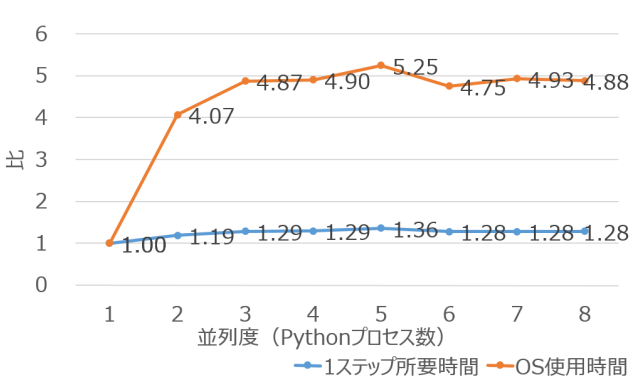

図 2 Matrix Factorization の並列度と実行速度

である。また Python 処理系および BLAS ライブラリは Intel 社の提供する高速化版 [6] を用いた。横軸は Python レベルでの並列度すなわち `multiprocessing` で起動するプロセス数、縦軸はプロセス数 1 の場合を基準とした場合の Matrix Factorization の 1 ステップに要した時間とそのうち OS が使用した時間である。なお `multiprocessing` はプロセスベース並列化で、ワーカ間でジャイアントロックを共有しないため CPU-intensive なタスクでも高速化が可能である。（一方 `threading` はスレッド間の共有メモリがジャイアントロックで過剰に保護されているため IO 待ちが多発するようなアプリケーションでしか高速化できない。）

マシンのコア数が 8 でありかつプロセス間の依存性がない（i ごとに独立）にも関わらず、図 2 では Python プロセス数 1 のときに実行時間と OS 使用時間ともに最短である。特に並列度を 2 にした時の OS 使用時間の伸びが顕著であり、多重並列化で並列度が上がりすぎていることが分かる。実際に `ps -eo comm,pid,tid,pcpu -M` によってスレッドの表示を行うと、**図 3** のように Python プロセス 1 つに対して複数のスレッドが起動して多重に並列化が行われている。カラムの内容は左からプロセス名、プロセス ID、スレッド ID、CPU 使用率である（プロセス名とプロセス ID はプロセスのみ、スレッド ID はスレッドのみに表示されている）。

```
python3.5       30330     - 11.6
-                   - 30330  4.6
-                   - 30339  0.0
-                   - 30340  7.2
-                   - 30341  0.0
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
```

図 3 並列化 Matrix Factorization 実行中の ps コマンド出力の一部

2.3 処理対象の非一様性

処理対象の非一様性は、人工知能やビッグデータ処理で現れるような実世界から得られたデータが一様な分布を示さないことが原因であり、結果として当該データ上で愚直なアルゴリズムの効率を大きく下げる。例えば web ページ間のリンク関係や Twitter 上のユーザ間のつながりなどの実データから得られた大規模グラフはノードの次数が exponential に変化することがよく知られている。このようなグラフ上でページランクを計算をすると、多くのノードは少ないステップ数で収束するが次数が多いノードは収束に数十から数百ステップを要する。よってすでに収束したノードを再計算しないなどの高速化を施さなければ無

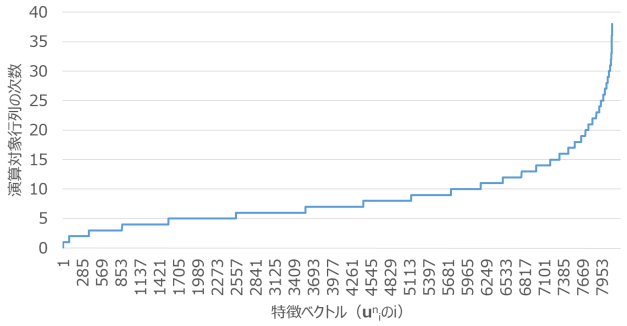

図 4 処理対象の非一様性：LU 分解される行列の次数

駄な計算が非常に多くなる。このように非一様な大規模グラフを効率よく処理する手法は近年盛んに研究されている [2], [3], [5]。

Matrix Factorization のような行列アルゴリズムにおいても処理対象の非一様性は顕著である。例えばユーザとムービーの例では、一部のユーザは沢山のムービを評価しているが多くのユーザは 1 種類のムービーしか評価しないといったことが起こり得る。この時近似される行列 R は非一様、すなわち一部の行には沢山の値が存在するが、値が 1 つしかない行も多く存在する。

処理対象の行列が非一様である結果、Matrix Factorization の計算もグラフ処理と同様に効率化を要する。具体的には、各ステップで演算の対象となる行列（例えば $\mathbf{u}^n_i$ の計算の途中で現れる行列）の次数が大きく変化する結果、常に一様な並列度で計算すると最適な並列度が得られない。ステップ n での $\mathbf{u}^n_i$ の計算は、$\mathbf{m}^{n-1}_j$ のうちユーザ i が評価したムービー j の関するベクトルを結合した行列を元に和、積、LU 分解などの一般的な行列演算を組み合わせて行う（詳しい計算式は文献 [12] を参照）。従ってムービーをたくさん評価しているユーザに関する計算では LU 分解を高い並列度で行うことができ、一方 1 つしかムービーを評価していないユーザに関する計算ではまったく並列化を行うことができない。

図 4 に大規模疎行列を Matrix Factorization した際の、1 ステップ内で BLAS ライブラリによって LU 分解される行列の次数を示す。分解した行列およびソフトウェア、ハードウェア環境は図 2 の実験と同様である。縦軸は LU 分解される行列の次数、横軸は求める行列中の特徴ベクトルの番号（$\mathbf{u}^n_i$ の i）を縦軸の値が小さい順に並び替えたものである。図より、約半分の特徴ベクトルに関する計算では LU 分解される行列の次数が 10 以下であるが、一部の特徴ベクトルに関する計算では 35 次以上の行列を LU 分解しているという処理対象の非一様性が確認できる。従って、BLAS ライブラリ内の並列度を一定の値にする（例えばスレッド数を 8 に固定する）とステップ中の多くの時間で最適な並列度が得られないことが分かる。BLAS ライブラリ内の並列度を細粒度に調整することは本稿で対象とするようなユーザには不可能であるため、この課題を自動的に解決する必要がある。

3. 提案手法と技術的課題

3.1 手法への要求

本節では多重並列化と処理対象の非一様性を解決する手法に求められる要求について論じる。提案手法の最重要な非機能要求は、「手軽に高速処理をしたい（HPC の専門家ではない）ユーザが利用できる手法であること」である。これを機能要求に分解すると次ようになる。

(1) 広く一般の（あるいは任意の）言語とライブラリの組み合わせに対して使える。
(2) ユーザに求める負担は最上位の言語（例えば Python）で簡易なものに留める。

要求 (1) は人工知能やビッグデータ処理をサポートする言語やライブラリは多岐にわたりまた日々新しいものが登場するためである。この要求により、例えば「ユーザレベルスレッドと work stealing で動的な並列度をサポートする BLAS ライブラリ」を作ることは解決にならない。要求 (2) はユーザの主たる関心が実装されるアルゴリズムやアプリケーションの正しさや精度にあるからである。従って例えば HPC で行われるように直感的な記述を崩して高速実行や最適な並列度を求めることはできない。

3.2 スケジューラによるアプローチと技術的課題

本稿では OS のタスクスケジューラの改変や新たなユーザレベルスレッドライブラリの実装など、スケジューラによる実行速度の改善を要求 (1) と (2) を共に満たす解決策として検討する。目標とするスケジューラは多重並列化による過大、過小な並列度を自動的に検知し調停することで実行速度の高速化を実現する。本スケジューラを実現する上での技術的課題は下記のようになる。

有効な並列度の検出 目標とする最大の有効な並列度、すなわち何スレッド/プロセスまで高速化が見込めるかを検知する必要がある。例えば Matrix Factorization の例では有効な並列度は LU 分解している行列の次数によって定まるが、OS 側からの自動検知は単純ではない。有効な並列度の検出のため、最初のステップでプロファイルした結果を static に用いることは可能である。これは機械学習のアルゴリズムにステップ式のものが多いことによる。さらにユーザに最上位言語レベルの簡易な変更を許す仮定からステップやステップ内の各処理の開始、終了を知ることもできる。

スレッドのグループ分け 求まった有効な並列度に対して、その並列度までスレッドをグループ分けする。同一のグループに属するスレッドは同一のデータに対する同一の処理の一部であることが望ましい。これは直感的にはどのスレッドとどのスレッドを合成して

よいかを表す。技術的にはスレッドが作られた際の `pthread_create` の引数、スレッドのスタックの内容、アクセスするデータ位置等から同一グループに属するべきスレッドを発見できる可能性がある。システムコール呼び出しの引数を解析することでタスク間の依存関係を抽出しスケジューリングに活用する研究 [11] 等が参考になる。

スレッドの合成 有効な並列度とそれを実現するスレッドグループができた時、次にグループ分けされたスレッドを合成し並列度を下げることが必要である。ユーザに求め得るのは最上位の言語レベルの簡易な変更のみであるため、下位で用いられるライブラリが過剰な数のスレッドを作成しようとすることは抑止できない。例えば図 3 の例では Python レベルのプロセス数に関わりなく BLAS ライブラリがコア数分のスレッドを作成している。従って一度作られたスレッドを物理的、あるいは見かけ上合成して並列度が少ない際と同程度の実行速度を実現する。例えば、同一のデータを分割処理する二つのスレッドを同一のスレッドグループであると検地できれば、それらを連続してスケジューリングすることでコンテキストスイッチの増加によるキャッシュミス等の増加を削減するできる（見かけ上のスレッド合成）。

4. 関連研究

本稿で分析した処理対象の非一様性による適切な並列化の難しさは、Matrix Factorization における LU 分解以外にも発生する。文献 [10] では、XeonPhi における疎行列とベクトルの効率的な乗算を提案する。当該文献では背景調査において疎行列とベクトルの積の演算において有効に使われているベクトルユニットの数が行列によって異なるという図が示されている（Figure 10）。これは本稿で述べた処理大量の非一様性による非効率な並列化そのものである。また文献 [10] は自動チューニングの研究であるという点でも本稿と類似する。しかし一般に自動チューニングでは (1) チューニングし得る要素がパラメータとして用意され、(2) チューニング方法が明確であることが必要であり、本稿で対象とするユーザや環境では既存の自動チューニングの適用は難しい。

文献 [7] は OpenCL の計算モデルを CPU で実行する際にスケジューリングを変更することでアクセスの局所性を向上させ高速な実行速度を得る研究である。本文献は本稿とスケジューリングの変更のみでオーバーヘッドを削減するという点で類似するが、OpenCL の計算モデルではどのスレッドとどのスレッドが独立/依存かコンパイル時に決定される点が異なる。

OS レベルで実際に作られるスレッド数をユーザの作成する論理スレッドから分離しコンテキストスイッチのオーバーヘッドを削減する手法として、Cilk [1] や MassiveThreads [9] などの軽量スレッドと Work Stealing を組み合わせた手法が広く研究されている。軽量スレッドはユーザ空間で実装され作成などの操作にシステムコールが必要なく、コンテキストスイッチ一回あたりのコストが低い。また Work Stealing はタスクが終了したワーカーが別のワーカーからタスクを移譲されることで実行時間が予測できないタスクに対しても適切な並列度を実現する（例えばあるワーカーだけ時間がかかり全体が遅くなるといった課題を解決する）。しかし、Work Stealing はタスクの移譲のオーバーヘッドが大きいため、時間のかかる大きなタスクを移譲する場合にのみ有効である。本稿で分析したような、短い行列演算で並列度を少なく抑えるという目的には適さない。

5. 結論

本稿では人工知能やビッグデータ処理を行うユーザが高速な実行速度を求める場合にユーザとライブラリによる多重並列化と実世界データの処理対象の非一様性が課題となることを指摘し、その具体例と解決のための要求、技術的課題を分析した。今後は本稿で述べた技術的課題をさらに要素技術的な課題に分割し考察と実装を進める。

参考文献

[1] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H. and Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System, *SIGPLAN Notice*, Vol. 30, No. 8, pp. 207–216 (1995).

[2] Chen, R., Ding, X., Wang, P., Chen, H., Zang, B. and Guan, H.: Computation and Communication Efficient Graph Processing with Distributed Immutable View, *International Symposium on High-performance Parallel and Distributed Computing*, pp. 215–226 (2014).

[3] Chen, R., Shi, J., Chen, Y. and Chen, H.: PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs, *European Conference on Computer Systems (EuroSys)*, pp. 1:1–1:15 (2015).

[4] Eigen: `http://eigen.tuxfamily.org/`.

[5] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D. and Guestrin, C.: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs, *USENIX Symposium on Operating Systems Design and Implementation*, pp. 17–30 (2012).

[6] Intel Distribution for Python Technical Preview: `https://software.intel.com/en-us/python-distribution`.

[7] Kim, H.-S., El Hajj, I., Stratton, J., Lumetta, S. and Hwu, W.-M.: Locality-centric Thread Scheduling for Bulk-synchronous Programming Models on CPU Architectures, *International Symposium on Code Generation and Optimization*, pp. 257–268 (2015).

[8] Languages used for analytics / data mining / data science: `http://www.kdnuggets.com/polls/2013/languages-analytics-data-mining-data-science.html`.

[9] Nakashima, J. and Taura, K.: MassiveThreads: A Thread Library for High Productivity Languages, *Concurrent Objects and Beyond*, Springer, pp. 222–238 (2014).
[10] Tang, W. T., Zhao, R., Lu, M., Liang, Y., Huynh, H. P., Li, X. and Goh, R. S. M.: Optimizing and Auto-tuning Scale-free Sparse Matrix-vector Multiplication on Intel Xeon Phi, *International Symposium on Code Generation and Optimization*, pp. 136–145 (2015).
[11] Zheng, H. and Nieh, J.: SWAP: A Scheduler with Automatic Process Dependency Detection, *Symposium on Networked Systems Design and Implementation*, pp. 1–14 (2004).
[12] Zhou, Y., Wilkinson, D., Schreiber, R. and Pan, R.: Large-Scale Parallel Collaborative Filtering for the Netflix Prize, *International Conference on Algorithmic Aspects in Information and Management*, pp. 337–348 (2008).