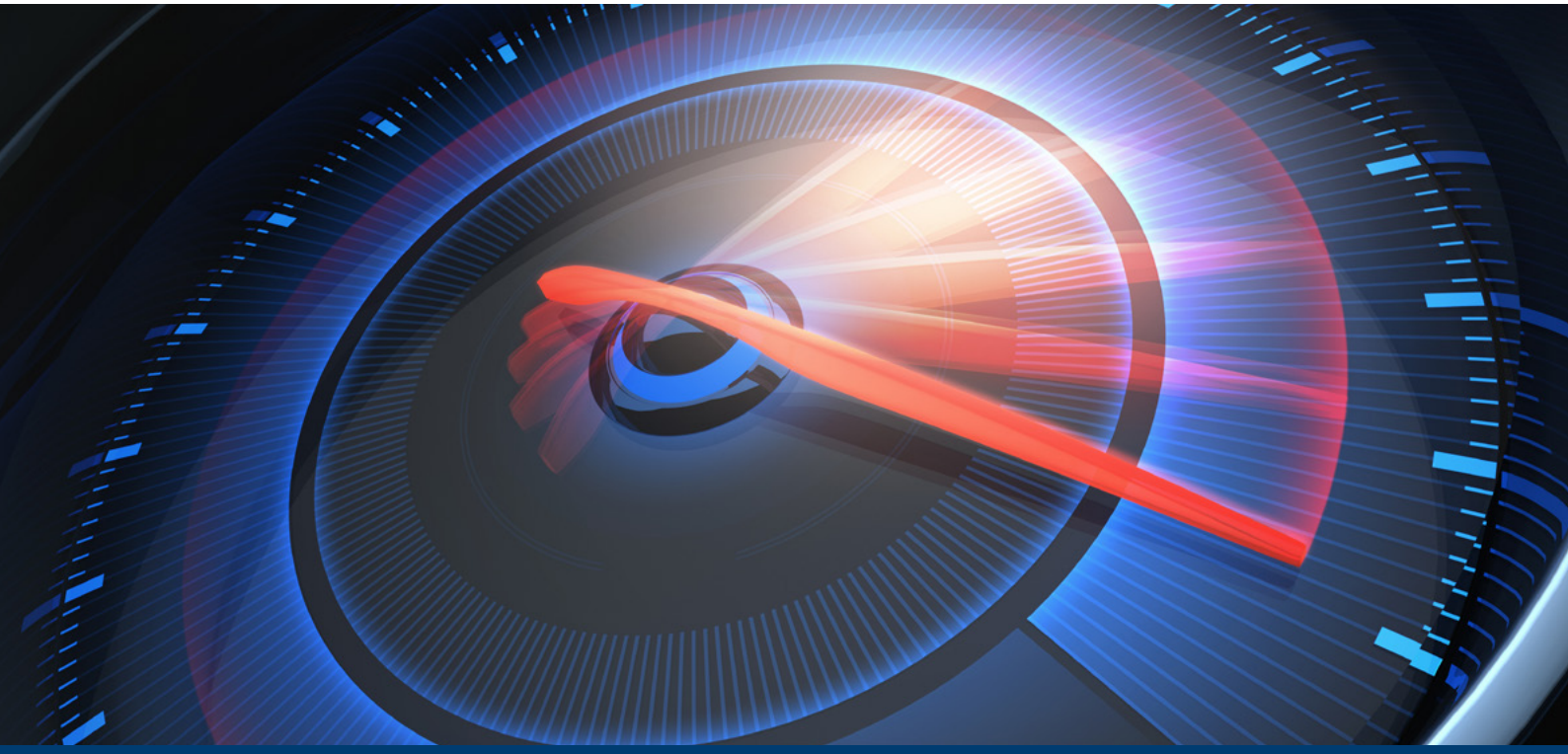


# マシンラーニングにおける PYTHON* パフォーマンスの壁を乗り越える

## Python* マシンラーニング・アプリケーションの高速化と最適化

**Vasily Litvinov、Viktoriya Fedotova、Anton Malakhov、Aleksei Fedotov、Ruslan Israfilov、Christopher Hogan インテル コーポレーション ソフトウェア・エンジニア**

### 協調フィルタリングによるパフォーマンスの高速化

インテルは、革新的なツール、手法、最適化により、Python* パフォーマンスを高速化するため、マルチコアおよび **SIMD** 並列処理向けのハイパフォーマンス・ライブラリー、プロファイラー、拡張サポートを提供します。ここでは、協調フィルタリングを使用して、実際のマシンラーニング・アプリケーションのパフォーマンスを高速化します。

結果は、これらの最適化を利用することで Python* でネイティブコードに近いパフォーマンスを達成できることを示しています。つまり、パフォーマンスを向上するため、C/C++ でコードを書き直す必要はありません。

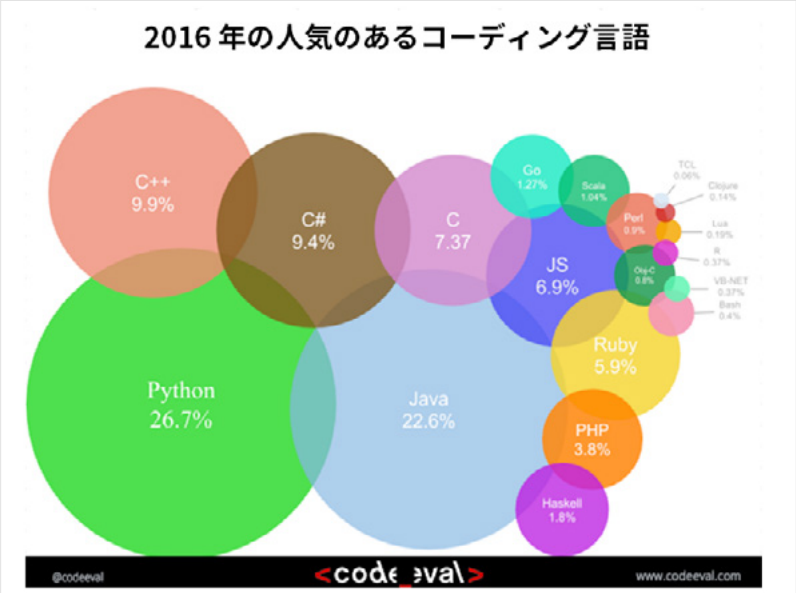
コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

# 問題を理解する

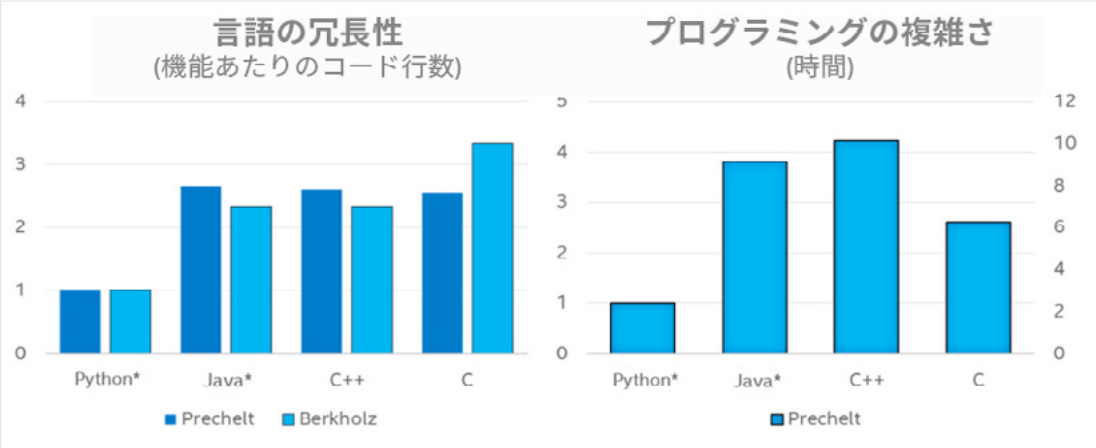
## Python* の人気

Python* は、その生産性の高さから、着実に採用が進んでいます。2016 年の **CodeEval.com** の投票では、Python* が開発者に最も人気のある言語に選ばれました。2 位は Java*、3 位は C++ でした (**図 1**)。[1] Python* の表現力とコーディングの容易さは、開発者にとってプロトタイプ環境として魅力的です (**図 2**)。[2,3]

Python* を使用することで、クオンツアナリストによる取引アルゴリズムの開発、データ・サイエンティストによる解析モデルの構築、研究者による数値シミュレーションのプロトタイプ生成が可能になります。




1 2016 年 2 月現在、最も人気があるコーディング言語 (出典: **www.codeeval.com** (英語))




2 プログラミング言語の生産性

## Python* パフォーマンスの問題

Python* のパフォーマンスとスケーラビリティーの問題により、通常プロトタイプ・コードをプロダクション・コードにスケーリングするため、開発者は C++ または Java* のような言語でアルゴリズムを書き直す必要があります。例えば、Python* では、評判が悪いグローバル・インタープリター・ロック (GIL) により、効率良い並列コードを記述することができません。アプリケーション・コードの書き直しは、労力と時間がかかるだけでなく、柔軟性を損ない、エラーにつながる可能性があります。

パフォーマンスの問題のほとんどは、Python* インタープリターの実装が現代のハードウェアに対応していないためです。インタープリターは、多種多様なハードウェア・プラットフォームで実行する必要があるため、多くの場合、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE) やインテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) のような CPU 固有の機能を効率良く利用するようにチューニングされていません。

さらに、Python* は、キャッシュの局所性を考慮して実装されていません。そのため、連続するデータがシーケンシャルに格納されず、データアクセスには多くのポインターの逆参照などが必要になります。これらはすべて、Python* インタープリターが最近のハードウェア・プラットフォーム上で優れたパフォーマンスを達成する妨げとなります。

# Python* の問題と制限に対する既知のソリューション

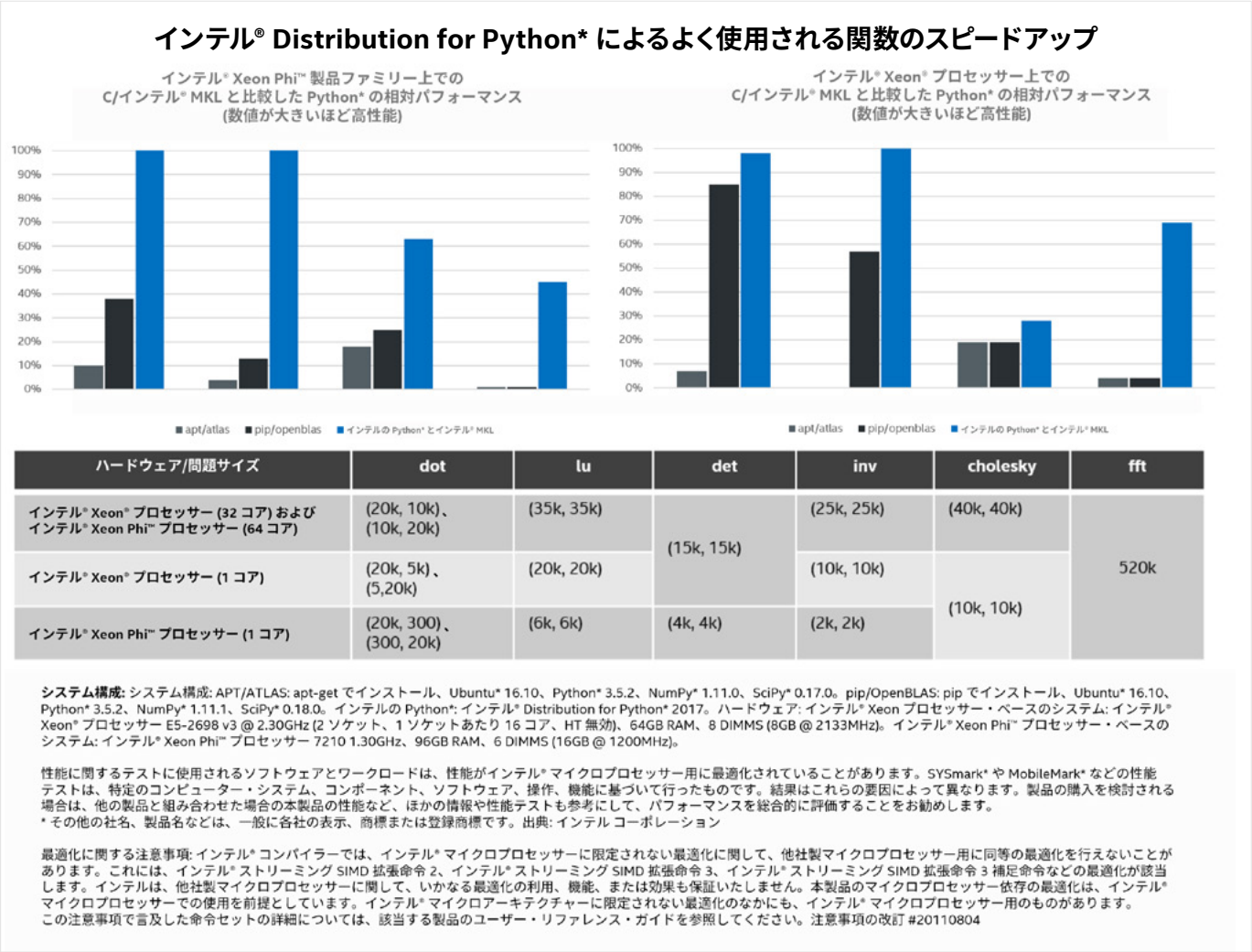
前述のパフォーマンスの問題は通常、Numba のような JIT コンパイラーを使用したり、NumPy*、SciPy*、scikit-learn パッケージの C 拡張を使用して対応できます。これらの Python* パッケージは、ハードウェアを最大限に利用できるように、スレッド対応で高度に最適化されており、パッケージを使用することで、Python* 開発者はアイデアを明確かつ簡潔に表現することができます。

ネイティブにコンパイルされるプログラミング言語を使用する場合、すべての CPU リソースを効率良く利用するのは、一般に開発者の責任です。しかし、多くの Python* 開発者は、アルゴリズムの最適化と並列化に多くの知識を持ち合わせていません (そして、それらの知識の習得は不要であるべきです)。そのため、ライブラリーを通してこれらの最適化を利用するのが簡単です。しかし、NumPy*、SciPy*、scikit-learn のようなライブラリーの標準実装は、汎用化されているため、通常対象とするハードウェアの機能を最大限に引き出せません。例えば、最近のベクトル命令をすべて使用することができません。[4]

## Python* コミュニティーへのインテルの貢献

インテルは、Python* のパフォーマンスの課題への取り組みとして、多くのツールと手法を通じて、Python* アプリケーションのパフォーマンスを高速化する、強力で簡単に使えるソリューションを開発者に提供しています。インテル® マス・カーネル・ライブラリー (**インテル® MKL**)、インテル® スレッディング・ビルディング・ブロック (インテル® TBB)、OpenMP* 標準の実装、MPI 標準の実装、インテル® Data Analytics Acceleration Library (インテル® DAAL) を含む、ハイパフォーマンス・アプリケーションの開発を支援する数々のライブラリーに加えて、インテル® VTune™ Amplifer XE などのプロファイリング・ツールがあります。しかし、最近まで、これらの言語とツールのほとんどは、C/C++/Fortran でのみ利用することができました。

インテルは、インテル® MKL を利用して NumPy* のようなパッケージの数値計算を最適化した、**インテル® Distribution for Python*** で Python* 開発者にこれらの強力なツールと手法の提供を開始しました。また、効率良いタスク・スケジューラーによりスレッド処理の調整を行うマルチスレッド・アプリケーションを可能にする Python* 向けのインテル® TBB モジュール [5] と、Python* でのビッグデータ解析を支援する Python* 向けのインテル® DAAL モジュール (pyDAAL) もリリースされました。人気のパフォーマンス・アナライザー **インテル® VTune™ Amplifer XE** では、Python*/C/C++/Fortran が混在するアプリケーションのプロファイルがサポートされ、アプリケーションのパフォーマンスを検証し、自明 / 非自明の対応可能なボトルネックを特定できるようになりました。




| ハードウェア/問題サイズ | dot | lu | det | inv | cholesky | fft |
|---|---|---|---|---|---|---|
| インテル® Xeon® プロセッサー (32 コア) および インテル® Xeon Phi™ プロセッサー (64 コア) | (20k, 10k)、(10k, 20k) | (35k, 35k) | (15k, 15k) | (25k, 25k) | (40k, 40k) | 520k |
| インテル® Xeon® プロセッサー (1 コア) | (20k, 5k)、(5,20k) | (20k, 20k) | | (10k, 10k) | (10k, 10k) | |
| インテル® Xeon Phi™ プロセッサー (1 コア) | (20k, 300)、(300, 20k) | (6k, 6k) | (4k, 4k) | (2k, 2k) | | |

**システム構成:** システム構成: APT/ATLAS: apt-get でインストール、Ubuntu* 16.10、Python* 3.5.2、NumPy* 1.11.0、SciPy* 0.17.0。pip/OpenBLAS: pip でインストール、Ubuntu* 16.10、Python* 3.5.2、NumPy* 1.11.1、SciPy* 0.18.0。インテルの Python*: インテル® Distribution for Python* 2017。ハードウェア: インテル® Xeon プロセッサー・ベースのシステム: インテル® Xeon® プロセッサー E5-2698 v3 @ 2.30GHz (2 ソケット、1 ソケットあたり 16 コア、HT 無効)、64GB RAM、8 DIMMS (8GB @ 2133MHz)。インテル® Xeon Phi™ プロセッサー・ベースのシステム: インテル® Xeon Phi™ プロセッサー 7210 1.30GHz、96GB RAM、6 DIMMS (16GB @ 1200MHz)。

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。
* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。出典: インテル コーポレーション

最適化に関する注意事項: インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。
この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。注意事項の改訂 #20110804

3 インテル® Distribution for Python* 2017 の Python* パフォーマンス

## 実際の例へのライブラリーとツールの適応

レコメンデーション・エンジンの構築のような一般的な問題について考えてみます。入力データは、購入商品とその商品に対するユーザーの評価と仮定します。構築されるレコメンデーション・システムは、次の 2 つの独立したタスクを行います : 1) レコメンデーション・ルールの構築、2) 特定のユーザーに対するレコメンデーションの計算。タスク 2 では、システムは Web サービスとして動作し、一度に複数のユーザーからのレコメンデーション要求が発生する可能性があります。

## レコメンデーション・ルールの構築

レコメンデーション・システムでモデルの作成に使用されるアルゴリズムは数種類あります。ここでは、商品ベースのレコメンデーションに注目します。モデルは、類似商品の行列で表されます。行列 $s_{ij}$ の各要素は、商品 **i** と **j** の類似性を表します。この実装では、類似性の評価にコサイン距離を使用します。

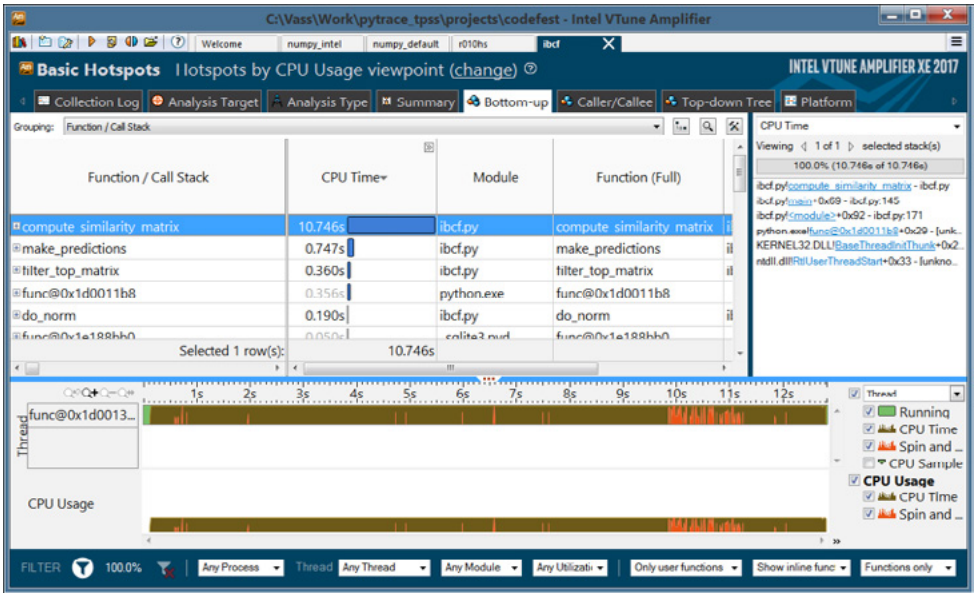
モデル生成アルゴリズムは、次のステップで構成されます。

1. ユーザーの評価をデータベースから行列にロードします。行列の行は個々の商品を、列は個々のユーザーを表します。
2. コサイン距離を使用して商品の類似性を計算します。
   a) 各行をそのユークリッド距離で割って、ユーザーの評価で行列を正規化します。
   b) 行列の行のドット積を計算します。
3. 各商品で、**i** は最も類似性の高い商品を **k** 個選択し、それ以外のすべての商品の類似性をゼロに設定します。

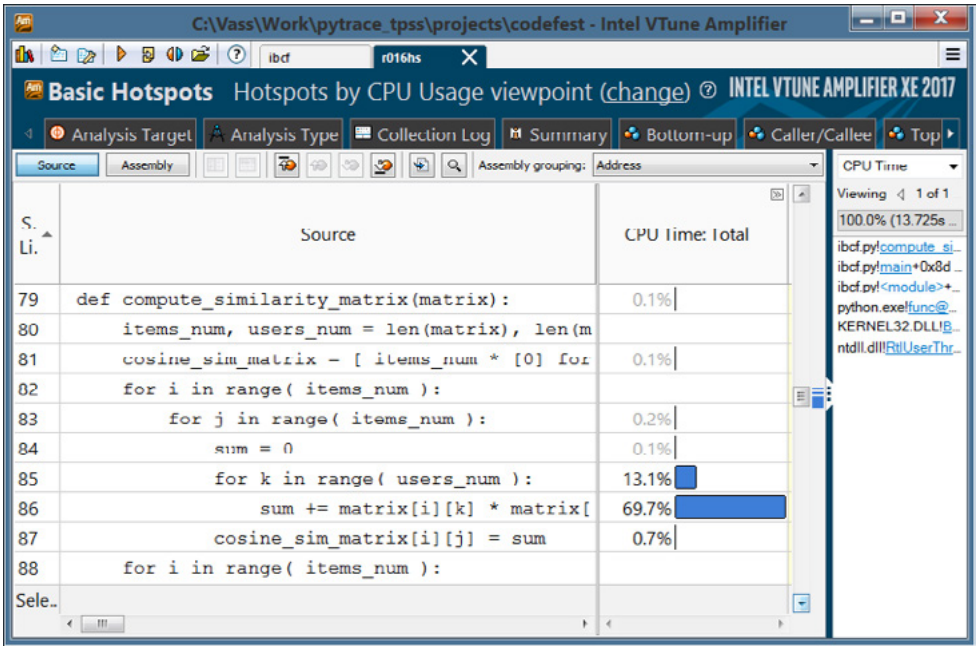
以下は、追加のパッケージを使用せずに、標準の Python* だけでステップ 2b を実装するコードです。

```
def compute_similarity_matrix(matrix):
items_num, users_num = len(matrix), len(matrix[0])
cosine_sim_matrix = [ items_num * [0] for i in range( items_num ) ]
for i in range( items_num ):
for j in range( items_num ):
sum = 0
for k in range( users_num ):
sum += matrix[i][k] * matrix[j][k]
cosine_sim_matrix[i][j] = sum
for i in range( items_num ):
cosine_sim_matrix[i][i] = 0
return cosine_sim_matrix
```

**grouplens.org** (英語) から、6,040 人のユーザーが 3,260 本の映画に対して行った 1,000,000 件の評価データを使用してこの実装のパフォーマンスを評価したところ、標準の Python* 実装では、データをロードし、類似性行列を作成するのに 338 分かかりました。また、主なボトルネックを理解するため、インテル® VTune™ Amplifer XE の Python* プロファイル機能を利用しました。その際、プロファイル時間を大幅に短縮するため、評価データの件数を 1,000,000 件から 20,000 件に減らしました。**図 4** と**図 5** にプロファイル結果を示します。




4 インテル® VTune™ Amplifer XE の [Bottom-up] ビューから計算負荷が最も高いのは商品の類似性計算であることが分かる

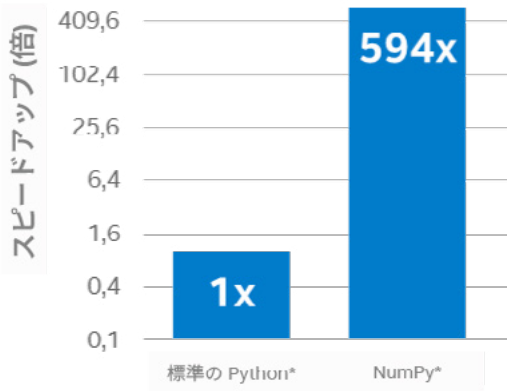



5 インテル® VTune™ Amplifer XE から実行時間の大半が 86 行目で費やされていることが分かる

プロファイル結果から、アルゴリズムで最も実行時間を費やしているのはドット積の計算であることが分かりました。Python* は、この処理を C/C++ や Fortran などのコンパイル言語のように効率良く行うことができません。これは、前述の理由によるものです。幸い、Python* には、追加パッケージとして多数の拡張が用意されており、それらを利用することで、ここで触れたほとんどの問題に対応することができます。NumPy* は、線形代数計算用の業界標準パッケージで、ndarray と呼ばれる追加のデータ型を提供します。ndarray は、連続するメモリーブロックとして表される任意の次元の配列です。NumPy* では、OpenBLAS、Atlas、Netlib のような一般的な数学ライブラリーを利用して線形代数処理を実装しています。

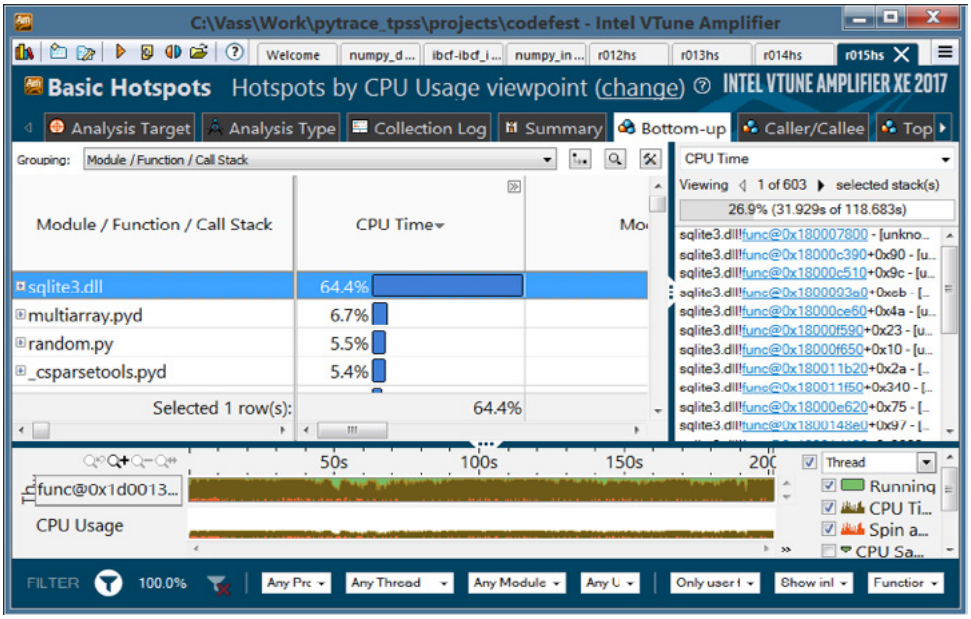
**図 6** は、最初の類似性行列計算を次のように変更した後のパフォーマンスの向上を示します。

```
import numpy as np
matrix = np.array(shape=(len(items), len(users)))
...
cosine_sim_matrix = matrix.dot(matrix.T)
```




システム構成 :
Fedora* に同梱の Python*: Python* 2.7.10 (default, Sep 8 2015)、NumPy* 1.9.2、SciPy* 0.14.1、gcc 5.1.1 でビルドされたマルチプロセシング 0.70a1。ハードウェア : 96 CPU (HT 有効 )、4 ソケット (1 ソケットあたり 12 コア )、1 NUMA ノード、インテル® Xeon® プロセッサー E5-4657L v2 @ 2.40GHz、RAM 64GB、オペレーティング・システム : Fedora* 23 (twenty-three)。

6 標準の Python* 実装と比較した NumPy* 計算のスピードアップ




7 インテル® VTune™ Amplifer XE の [Bottom-up] ビューからアプリケーションの実行時間の 64.4% がデータベースからのデータのロードに費やされていることが分かる

**図 7** のプロファイルから、データのロードが新しいボトルネックであることが分かります。つまり、これ以上計算部分を最適化しても意味がありません。

## ユーザーに対するレコメンデーションの生成

協調フィルタリングの 2 つ目のタスクは、ユーザーのこれまでの評価データを基に未評価商品に対する好みを予測します。商品 **i** のユーザーの好みを計算するため、商品 - 商品の類似性行列の **i** 行目とユーザー評価のベクトルの乗算を行います。類似性行列の各行に対してこの処理を行うことで、すべての商品に対するユーザーの好みを予測します。そして、ユーザーがすでに評価済みの商品を除外後、好みベクトルをソートして、上位 **k** 個の要素を選択します。ここでは、タスクを簡略化するため、上位 1 つのレコメンデーションのみ取得しています。以下のコードは、NumPy* と Numba を使用して、アルゴリズムのこの部分を実装します。

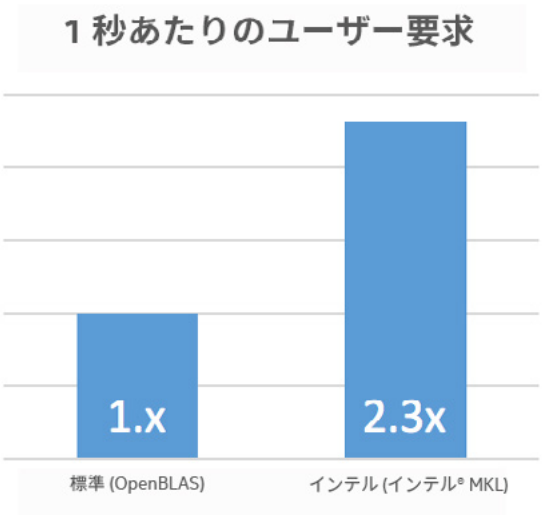
```
@numba.guvectorize('(f8[:],f8[:])', '(),()', target="parallel")
def masking(x, u):
    if u[0]:
       x[0] = 0

x = topk_matrix.dot(user_vector)
masking(x, user_vector)
recommendation_ids = x.argmax(axis=0)
```

関数マスクをネイティブコードにコンパイルし、引数で渡された行列の各要素に並列に適用して、Python* コードを最適化するため、ここでは Numba デコレーターを使用しています。

Web サービスとして動作するアプリケーションでは、レコメンデーションを得るために毎秒数十万の要求を処理する必要がありますが、類似性行列を計算するタスクは、頻繁に実行する必要はありません (例えば、1 日 1 回で十分です)。特定のユーザーに対するレコメンデーション生成タスクは、いくつかの要求をバッチ処理する、大きなタスクのサブセットとして見ることができます。この状況をテストするため、500,000 ユーザーに対するレコメンデーションを一度に計算します。`user_vector` 行列の各次元は、個々のユーザーの評価を表します。

**図 8** は、インテル® Xeon Phi™ プロセッサー・ベースのシステム上で、インテル® Distribution for Python* と標準の NumPy* パッケージを使用してレコメンデーションを生成した場合のパフォーマンスです。




システム構成 :
ハードウェア : インテル® Xeon Phi™ プロセッサー 7250、68 コア @1.40GHz、96GB DDR4-2400

バージョン :
インテル® Distribution for Python* 2017 Gold、インテル® MKL バージョン 2017.0.0、libopenblasp-r0-39a31c03.2.18.so、Python* 3.5.2、NumPy* 1.11.1、SciPy* 0.18.0、Red Hat* Enterprise Linux* Server 7.2 (Maipo)

8 標準の Python* + NumPy* パッケージ (OpenBLAS を使用) とインテル® Distribution for Python* (インテル® Xeon Phi™ プロセッサー・ベースのシステム上でインテル® MKL を使用) のユーザーレコメンデーション生成パフォーマンスの比較

標準の NumPy* パッケージは最近のリリースで、インテル® Xeon® プロセッサー上でより高速に実行する OpenBLAS 数学ライブラリーを使用するように変更されました。これにより、標準の NumPy* パッケージの以前のバージョン (OpenBLAS を使用しない) とインテル® Distribution for Python* の間にあったパフォーマンス・ギャップが小さくなりました。標準の NumPy* の新しいバージョンとインテル® Distribution for Python* は、どちらも並列化とベクトル化を最適化しています。しかし、これだけでは、インテル® Xeon Phi™ プロセッサー・ファミリーの新しいインテル® AVX-512 命令と、通常のプロセッサー・アーキテクチャーとは大きく異なるインテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャーの性能を引き出すことができません。

インテル® Xeon® プロセッサー・ベースのシステムでは、処理をさらにスピードアップできる可能性がもう 1 つあります。ユーザーの好みの計算をアプリケーション・レベルでマルチスレッド化することです。ここでは、NumPy* に似たインターフェイスを備え、NumPy* を呼び出す並列タスクにワークを分割できる Dask ライブラリーを使用します。

Dask を使用するには、例えば、NumPy* 配列の代わりに Dask 配列を宣言する必要があります。

```
chunks=(int(topk_matrix.shape[0]), 5000) # 1 タスクあたり 5000 ユーザー
t = dask.array.from_array(topk_matrix, chunks)
u = dask.array.from_array(user_vector, chunks)
```

この例では、以前に宣言した行列をソースデータとして使用していますが、実際のアプリケーションは最初から Dask 配列を扱うことができるため、変換オーバーヘッドは発生しません。上記の NumPy* コードは、次のように変更できます。変更後のコードは、上記のコードで宣言されたものと同じ **`masking`** 関数を使用します。

```
x = t.dot(u)
dask.array.map_blocks(masking, x, u)
recommendation_ids = x.argmax(axis=0).compute()
```

このコードは、複数のスレッドを使用して、配列の異なるチャンクを並列に処理するタスクを生成します。実行時間の大半がグローバル・インタープリター・ロック (GIL) を解放する NumPy* 内で費やされているため、GIL は大きな問題にはなりません。そのため、複数の計算を並列に実行できます。しかし、NumPy* 計算の並列実行は、オーバーサブスクリプションと呼ばれる別の問題を引き起こす可能性があります。[6] この問題は、2 つの入れ子構造レベル (1 つは Dask から、もう 1 つは**インテル® MKL** 内) で同時にワークが並列実行されると発生します。どちらの並列レベルも、システム全体がアイドル状態で、直ちに並列処理を行うことができるかのように振る舞います。最悪の場合、作成されるスレッド数はコア数の 2 乗になり、アプリケーションは有用なワークの代わりに、コンテキスト・スイッチやキャッシュミスに大半の時間を費やします。オーバーサブスクリプションの問題に対応するため、ここでは優れた入れ子構造の並列処理を提供するインテル® TBB ライブラリーを使用します。インテル® TBB の Python* ラッパーは、Dask で使用される標準の Python* の ThreadPool クラスを置換できます。Dask やアプリケーションのソースコードを変更する必要はありません。次のコマンドライン・オプションを指定すると、インテル® TBB の Python* ラッパーが有効になります。

```
$ python -m TBB <your>.py
```

このオプションは、インテル® MKL でインテル® TBB のスレッドレイヤーも有効にするため、両方の並列レベルで、インテル® TBB のタスク・スケジューラーによってタスクと利用可能な処理ユニットの調整が行われます。

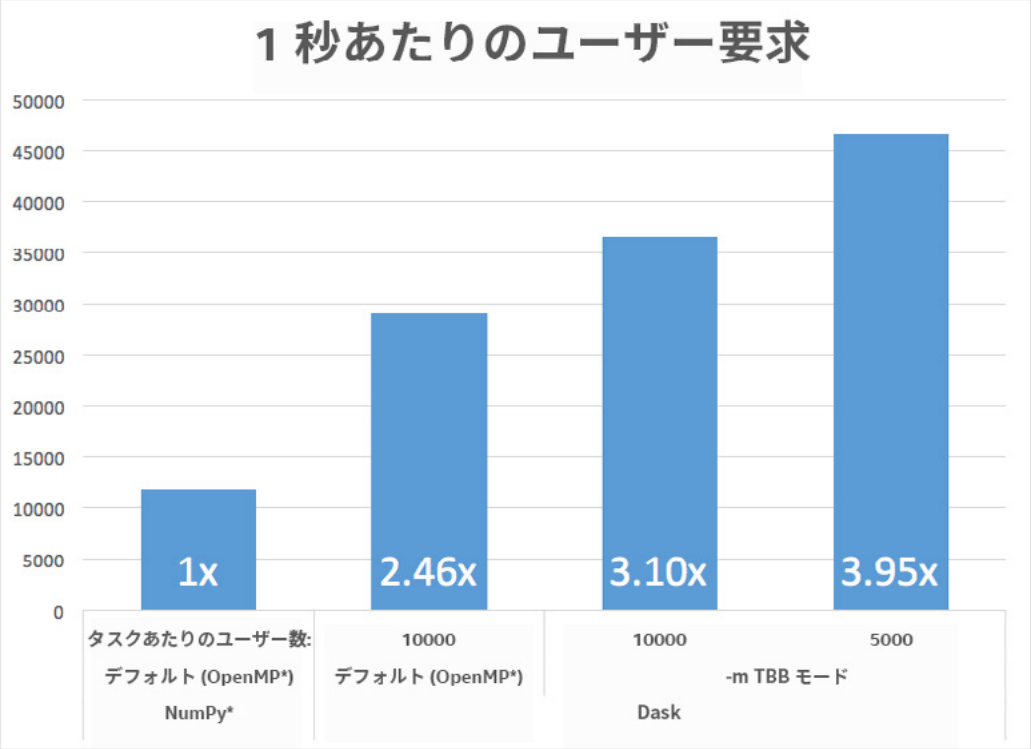
オーバーサブスクリプションは、パフォーマンスに影響するだけではありません。多数のプロセッサー・コアが搭載された大規模システムでは、オペレーティング・システムが多数のスレッドの作成を拒否し、次のようなメッセージを出力します。

```
OMP: エラー #34: システムは OMP スレッドに必要なリソースを割り当てることができません:
OMP: システムエラー #11: Resource temporarily unavailable
OMP: ヒント: OMP_NUM_THREADS の値を減らしてみてください。
```

(OpenBLAS で高速化された) 標準の NumPy* を実行しているシステムとアプリケーションは、追加の情報やヒントを表示することなく失敗します。セグメンテーション違反が発生する可能性もあります。インテル® TBB の Python* モジュールは、この問題が発生した場合にアプリケーション側で対応するための簡単な方法を提供します。OMP_NUM_THREADS=1 を設定してプログラムを続行することもできますが、プロセス全体で並列処理が制限されます。インテル® TBB モードで実行中、プログラムは動的にスケーリングし、バランスを取ります。

**図 9** は、上記のコードベースのベンチマーク・アプリケーションを実行した結果です。

デフォルトの NumPy* バージョンは、どの Dask (マルチスレッド) バージョンよりも遅くなります。デフォルト (OpenMP*) モードの Dask バージョンは、1 タスクあたり 10,000 ユーザーでのみ実行できました。1 タスクあたり 5,000 では、常に前述の “OMP: エラー #34” で失敗しました。最良の結果は、インテル® TBB モードで 1 タスクあたり 5,000 ユーザーのチャンク (合計 500,000 ユーザー) を実行した際に得られました。




システム構成 :
ハードウェア : インテル® Xeon® プロセッサー E7-8890 v4、4 x 24 コア @ 2.20GHz ( 最大 4GHz)、HT 無効、768GB DDR4。
バージョン : インテル® Distribution for Python* 2017 Gold、インテル® MKL バージョン 2017.0.0、Python* 3.5.2、NumPy* 1.11.1、SciPy* 0.18.0、Numba 0.26.0、llvmlite 0.11.0、Dask 0.11.0、CentOS* Linux* 7.2.1511 (Core)。

9 異なるパラメーター設定による NumPy* と Dask バージョンのレコメンデーション生成ベンチマークのパフォーマンスの比較

## 分散プログラミング

実際のアプリケーションからのデータは膨大な量に上ることがあります。Netflix* のコンテストでは、480,000 メンバーの 18,000 本の映画の評価が行列に格納され、行列サイズは 65GB でした。この膨大な量の計算とデータ割り当てには、協調フィルタリング・アルゴリズムの分散バージョンを使用できます。

インテル® DAAL は、データの前処理からデータマイニングおよびマシンラーニングまで、すべてのデータ解析段階に対応したアルゴリズムを提供します。テキストファイル、データベース、分散ファイルシステムを含む外部データソースからデータをロードし、実際の計算を行うまでの計算フロー全体を最適化するように設計されています。インテル® DAAL は C++ で記述されていますが、インテル® Distribution for Python* の pyDAAL コンポーネントで Python* ラッパーが用意されています。

インテル® DAAL は、いくつかの処理モードをサポートします。

- **バッチ :** このモードは、すべてのデータが 1 つのデバイス上にあり、一度に処理可能な場合に使用します。
- **オンライン :** このモードは、データをブロック単位で受け取る場合、またはデータが大きすぎて一度にデバイスのメモリーに収まらない場合に使用します。この場合、部分結果を段階的に更新して、データをデバイスメモリーにストリームし、次のデータブロックを処理する前にファイナライズします。
- **分散 :** このモードは、データセットが大きすぎて 1 つのデバイスのメモリーに収まらない場合、または複数のデバイスで実行することでアルゴリズムの処理時間を短縮できる場合に使用します。この場合、アルゴリズムは各ノードで部分結果を生成し、最後にマスターノードで最終結果にマージされます。

それぞれの処理モードで使用される API は似ており、必要に応じて、あるモードから別のモードへ比較的簡単に切り替えることができます。例えば、プロトタイプの生成段階ではバッチ処理モードと比較的小さなデータセットを使用し、プロダクション段階では分散処理モードと大きなデータセットに切り替えることができます。

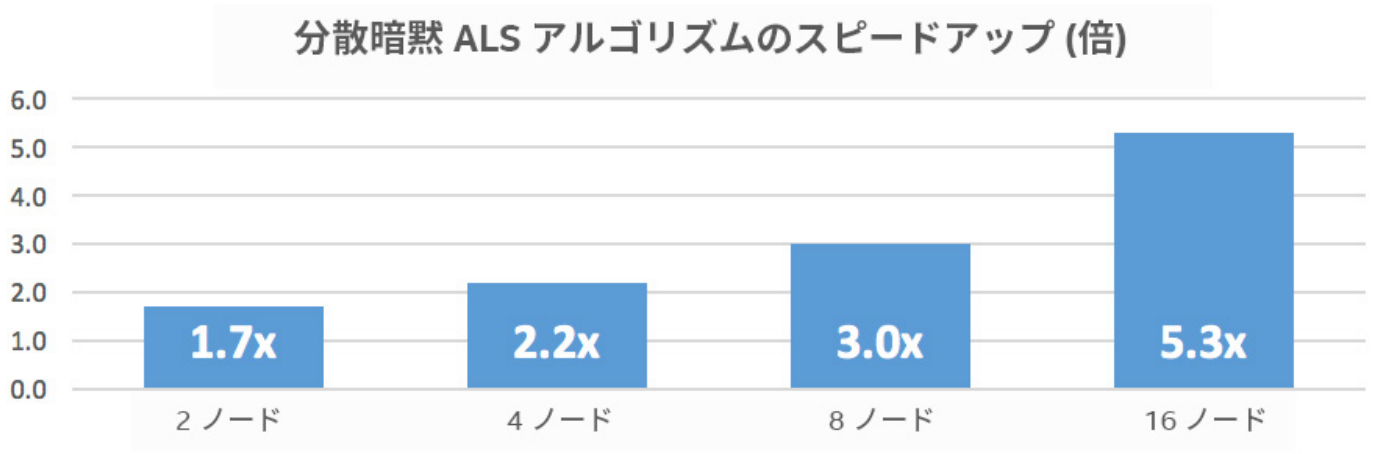
インテル® DAAL の分散処理モードは、通信レイヤーに依存せず、mpi4py や PySpark とともに利用できます。特定のハードウェア向けに最適な通信スキームを選択するのはユーザーの責任ですが、インテルのパッケージには、さまざまな通信技術を利用して分散アルゴリズムを実行する方法を示すサンプルが含まれています。

インテル® DAAL は、バッチおよび分散処理モードの協調フィルタリング向けに暗黙の交互最小 2 乗 (暗黙 ALS) アルゴリズムを提供します。mpi4py と pyDAAL を利用して分散協調フィルタリングを実装する方法を示すソースコードは、**インテル® DAAL のサンプルコード・ページ**から入手できます。

インテル® DAAL の暗黙 ALS のパフォーマンスを示すため、次のベンチマークを用意しました。このベンチマークは、分散暗黙 ALS アルゴリズムを、人為的に生成された 10,000,000 ユーザーの 10,000,000 商品に対する評価を含むデータセットに適用します。データセットはスパースで、すべてのユーザーと商品に一様に分布された 100,000,000 件の既存の評価が含まれており、それ以外の評価はゼロに設定されています。インテル® DAAL の実装は、CSR (Compressed Sparse Row) 形式のスパースデータを扱うため、ベンチマーク・データはこの形式で格納されています。暗黙 ALS アルゴリズムのパラメーターのうち、計算の複雑さに大きく影響する因数の数と反復数は、ぞれぞれ 10 と 15 に設定されています。

このベンチマークは、通信レイヤーに **MPI** を使用し、1 から 16 ノードの範囲でスケーリングします。各ノードで 1 つの MPI ランクが開始されます。データセットは、ノード数に応じていくつかのパートに分割されます。

**図 10** は、それぞれのクラスターノード数での暗黙 ALS トレーニング・アルゴリズムのパフォーマンス・スケーリングを示したものです。シングルノードでは、計算に 192.8 秒かかります。グラフでは、この値をベースに相対的なパフォーマンスの向上を示しています。




システム構成:
ハードウェア (各ノード): インテル® Xeon® プロセッサー E5-2697 v4 @ 2.30GHz、2 x 18 コア、HT 有効、128GB RAM。
バージョン: Oracle* Linux* Server 6.6、インテル® DAAL 2017 Gold、インテル® MPI ライブラリー 5.1.3。
インターコネクト: 1GB Ethernet。

10 異なるクラスターノード数での分散暗黙 ALS アルゴリズムのパフォーマンス・スケーリング

## まとめ

複数のツールと手法により、Python* アプリケーションのパフォーマンスを大幅に向上し、ネイティブコードのスピードに近づけることができます。協調フィルタリング・アプリケーションの例から、インテル® Distribution for Python* とインテル® VTune™ Amplifer XE に含まれる Python* ツールは、インテル® プロセッサー上で有用であることが分かります。標準の Python* アプリケーションのプロファイルでは、ドット積関数でパフォーマンス・ボトルネックが正確に検出されます。

インテル® Distribution for Python* に含まれるインテル® MKL により最適化された NumPy* を使用することで、インテル® Xeon Phi™ プロセッサー・ベースのシステム上で、標準の NumPy* バージョンよりもパフォーマンスを 2.3 倍向上できます。

スレッドのオーバーサブスクリプションの問題は、インテル® TBB パッケージにより対応できます。効率良いスレッド・スケジューリングによって入れ子構造の並列処理のパフォーマンスを向上し、シングルスレッド・アプリケーションと比べて最大 4 倍のスピードアップを達成できます。

インテル® DAAL の暗黙 ALS アルゴリズムは、パフォーマンスを高速化するため、インテル® Xeon® プロセッサー・ベースのノード上で分散処理モードで実行する大規模データセット向けに実装されています。

Python* は、誰もがコンピューティングを利用できるようにしました。Python* 向けの強力なパフォーマンス・ツールは、開発者とユーザーが、パフォーマンスを諦めることなく、Python* を使用できるようにします。

**インテル® DAAL の詳細 >**

**インテル® DISTRIBUTION FOR PYTHON* の詳細 >**

## 参考資料 (英語)


1. Studies on popularity of coding languages by **www.codeeval.com**, February 2016, **blog.codeeval.com/codeevalblog/2016/2/2/most-popular-coding-languages-of-2016**.
2. An empirical comparison of seven programming languages, L. Prechelt, IEEE Computer, 2000, Vol. 33, Issue 10, pp. 23–29.
3. Programming languages ranked by expressiveness, D. Berkholz, RedMonk, March 2013, **redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness**.
4. HPC Trends and What They Mean for Me, J. Cownie, Imperial College, Oct. 2015.
5. Unleash parallel performance of Python programs, A. Malakhov, April 2016, **software.intel.com/en-us/blogs/2016/04/04/unleash-parallel-performance-of-python-programs**.
6. Collaborative Filtering for Implicit Feedback Datasets, Y. Hu, Y. Koren, and C. Volinsky, Eighth IEEE International Conference on Data Mining, Pisa, 2008, pp. 263–272.


## BLOG HIGHLIGHTS

### Wind River* Simics で Simics を実行してカーネル 1-2-3 バグを発見

**JAKOB ENGBLOM >**

私は、バグ、ソフトウェアの不可解な / 予想外の動作、デバッグに関する話が大好きです。これらは、ソフトウェア開発というドライな世界に、笑いと騒動をもたらします。バグによっては、解決してみれば大したことがないものであったり、人に話すのが恥ずかしいものもあります。あるいは、伝説のイエティのように捉えどころがなく、見つからないものもあります。さらに、勇敢なプログラマーが素晴らしいひらめきと粘り強さにより、ドラゴン (デバッグ) を制した英雄的物語のようなものもあります。これはそんなバグの 1 つで、Linux* カーネルで見つけた非常に困難なバグに関する話です。Wind River* Simics (Simics) で Simics を実行して見つけました。このバグを再現するには、1 つのネットワーク、2 台のマシン、少なくとも 3 つのプロセッサー・コア ( タイトルの 1-2-3 はこれに由来します ) が必要です。このバグは、シングル・プロセッサー上でマルチタスクだけでなく、コンカレンシーを必要とします。そして、バグを見つけるため、ユーザー空間と Linux* カーネルコードの両方を延々とたどっていく必要があります。

**この記事の続きはこちら (英語) でご覧になれます。>**