Vol.2018-HPC-164 No.5
2018/5/7

# Xeon Phi のためのハイブリッドソート

土田翔馬[†1]　中村真輔[†1]　廣田千明[†1]

**概要**：近年，並列計算で用いられている機器の一つに Xeon Phi Knights Landing がある．Knights Landing では AVX-512 命令がサポートされており，これによって SIMD の性能が大幅に向上されている．この命令を用いた高速なソートアルゴリズムとして Bramas（2017）は AVX-512 命令を活用してクイックソートを元にしたアルゴリズムを提案している．このアルゴリズムは従来のソートよりも高速な処理を実現しているが，入力列が大きい場合に AVX-512 命令を活用するために無駄な処理を多く含む．したがって本稿では入力列を分割して Bramas のアルゴリズムを分割された小さな領域に適用し，その後 AVX-512 命令を活用したバイトニックソートを行うハイブリッドソートアルゴリズムを提案・実装し，Knights Landing 上で評価を行うことで Bramas のアルゴリズムよりも高速であることを示す．

**キーワード**：Xeon Phi，クイックソート，バイトニックソート，SIMD，AVX-512

## 1. はじめに

ソートアルゴリズムは幅広く使われている基本的なアルゴリズムである．また，データベースサーバなどの特定のアプリケーションではソートが中心的な操作となっている．したがって，新しいアーキテクチャ上で実行可能な効率的なソートライブラリを開発することは幅広いアプリケーションの効率の向上につながる．

よく知られている逐次ソートアルゴリズムとしてクイックソート[1]がある．このアルゴリズムは要素数を$N$として平均で$O(N\log_2 N)$の計算量となり，逐次ソートアルゴリズムでは最も高速であると言われている．それに対し，並列ソートアルゴリズムとしてはバイトニックソート[2]が知られている．このアルゴリズムではソートする要素数が少ないと並列性が活用できずクイックソートと比べると処理に時間がかかってしまうが，要素数が多い場合は並列性を十分に活用でき，クイックソートよりも速くソートすることができる．

近年，並列計算で用いられている機器の一つにインテル社の Xeon Phi[3]がある．第 1 世代の Xeon Phi Knights Corner（KNC）[4]はおよそ 60 個のコアを搭載しており，第 2 世代の Xeon Phi Knights Landing（KNL）[5]は KNC よりも多くのコアを搭載している．各コアはインテルハイパースレッディング・テクノロジーによって 4 スレッドで実行が可能である．また Xeon Phi は 512 ビットの SIMD ベクトルを持ち，KNL ではインテル AVX-512 命令[6]と呼ばれる 512 ビットの SIMD 命令をサポートしている．KNL では基本的な命令であるインテル AVX-512F など 4 つの AVX-512 命令をサポートしている．これにより SIMD の性能が従来の AVX 命令よりも大幅に向上されている．これらのことより多数のコアに加え，AVX-512 命令を活用するためのアルゴリズムが求められている．

AVX-512 命令を活用した最新のソートアルゴリズムとして，Bramas はクイックソートを元にベクトル化したアルゴリズムを考案している[7]．このアルゴリズムでは従来のクイックソートの手順を元に AVX-512 命令を活用した分割を行い，分割して得られた列が特定の値より小さくなった時に小さな領域向けのバイトニックソートを行っている．このアルゴリズムによって GNU C++ STL のソートよりも 4 倍の高速化を実現したという成果が得られている．

†1 秋田県立大学
Akita Prefectural University

## 2. 基本的なソートアルゴリズム

一般によく知られているクイックソートは逐次処理で最も高速とされている．それに対しバイトニックソートは並列処理をした場合に高速にソートができるソートアルゴリズムとして知られている．

### 2.1 クイックソート

Hoare が考案したクイックソート[1]は分割統治法を用いたソートアルゴリズムである．分割の方法は，まずピボットと呼ばれるある値を選び，列の前半にその値以下の値をまとめ，後半にその値より大きい値をまとめる．クイックソートはこの分割を分割後の各列のサイズが 1 になるまで再帰的に行うことでソートを行う．クイックソートは最悪の場合で$O(N^2)$の計算量となるが，実際には平均計算量の$O(N\log_2 N)$となる．図 1 にクイックソートの例を示す．破線で囲われた値はピボットを表し，塗りつぶしている値は位置が確定した値を表している．

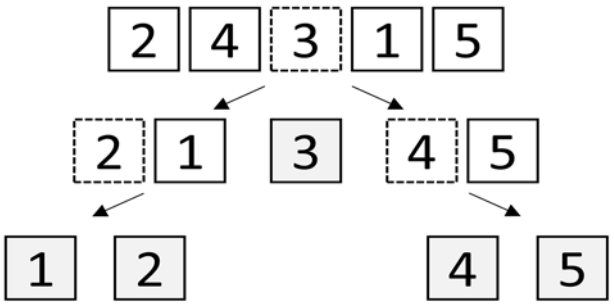


図 1　クイックソートの例

ピボットがソートする列に含まれている値のうちの最小値もしくは最大値のとき計算量は最悪となり，中央値の

ときに計算量は最良となる．このようにクイックソートの効率はピボットの選択に大きく影響される．したがって，計算量が最悪とならないように一般的に適当な 3 つの値を選び，そのうちの中央値をピボットとする手法が用いられている．

### 2.2 バイトニックソート

Batcher が考案したバイトニックソート[2]はソーティングネットワークを用いたソートアルゴリズムであり，複数の比較器を並列に処理できることが特徴である．バイトニックソートは単調増加と単調減少からなるバイトニック列と呼ばれる列を元にソートを行う．

ソートの手順はまず入力列を要素 2 つずつに分割する．そして各 2 要素のブロックを交互に昇順，降順になるように入れ替えを行うことで要素数が 4 のバイトニック列を作る．その後は隣り合ったバイトニック列を交互に昇順，降順にソートし，2 倍の長さのバイトニック列を再帰的に作る．最後にバイトニック列と入力列の長さが同じになったときにそれを昇順にソートすることでソートが完了する．バイトニックソートはこの特性から，入力列は$m$を自然数として$2^m$であることが望ましい．図 2 に入力列の要素数が 16 の場合のバイトニックソートの例を示す. 矢印は比較器を表し，矢印の始端が小さくなり，終端が大きくなるように入れ替えを行う．実線で囲われている部分はそれぞれバイトニック列のソートを表しており，赤色の部分は昇順のソートを表し, 青色の部分は降順のソートを表している．破線で囲われている部分は連続的に同じ比較・交換の処理を適用できる部分を表している．この破線で囲われている部分をスワップブロップと呼ぶこととする．また，バイトニックソートの計算量は$O(N(\log_2 N)^2)$であり，並列化した場合スレッド数を$T$とすると計算時間は$O(\frac{N}{T}(\log_2 N)^2)$となることが期待される．

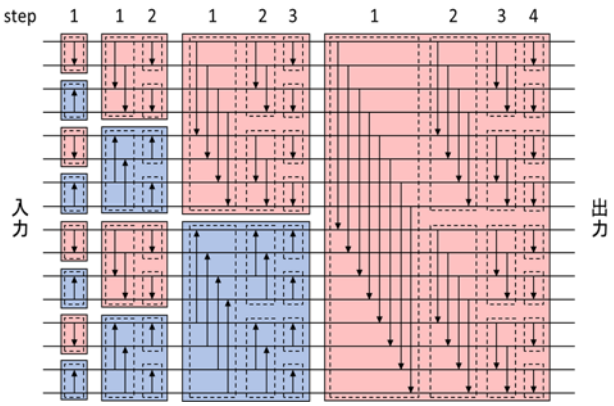


図 2　16 要素のバイトニックソート

## 3. Bramas のソートアルゴリズム

文献[7]のアルゴリズムは大きく分けて 2 つのステップで構成されている．文献[7]では後述する 2 つのステップを AVX-512 命令で実装することによって GNU C++ STL のソートよりも 4 倍の高速化を実現している．

最初のステップは SIMD パーティショニングと呼ばれる, AVX-512 命令を用いた分割である．SIMD パーティショニングの図を図 3 に示す．これは基本的に通常のクイックソートと同じである．手順はまず，入力列の左端，中央，右端の 3 つの値の中央値をピボットとして選択し，それを右端の値と交換する．選択したピボットからはピボットのみで構成されたピボットベクトルを作成する．次にピボット以外の列を左右から 512 ビット毎に区切る．512 ビットでちょうど区切れないときは中央のブロックが余りの長さとなる．次に図 3 の 2 段目に示すように，区切られたブロックのうち左端と右端のブロックをそれぞれロードし退避させ，その後左から 2 番目のブロックをロードする．その後ロードしたベクトルをピボットベクトルと比較し，3 段目に示すようにピボット以下の値を左に，ピボットより大きい値を右にストアする．次に左右のロード済みかつストアされていない領域を調べ, 小さい方のブロックをロードし，同様に比較しストアする．これを繰り返し行い，最後に最初にロードした左端と右端のベクトルを比較しストアした後に，図 3 の下部に示すようにピボットより大きい列の左端とピボットを交換することで SIMD パーティショニングは完了する．この処理を分割されたサイズが 512 ビットの 16 倍以下になるまで再帰的に行う．

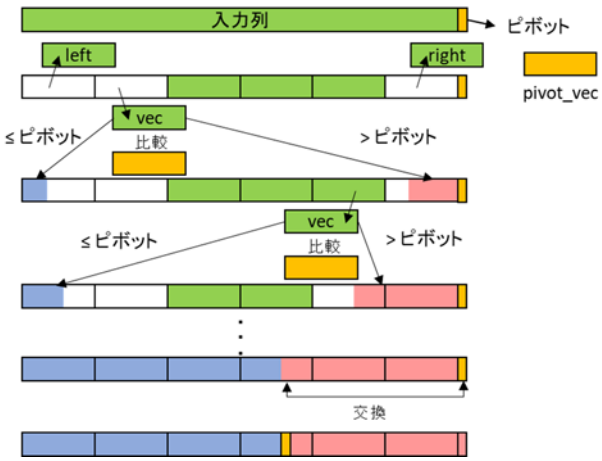


図 3　SIMD パーティショニング

次のステップとして，SIMD パーティショニングによって分割されたサイズが 512 ビットの 16 倍以下のときにバイトニックソートを行う．このバイトニックソートはベクトルの数が何本になるかを判定し，それぞれに対応した小領域向けのバイトニックソートを行う．図 4 に概要図を示す.

破線の矢印は SIMD パーティショニングによる分割を表しており，実線の矢印は小領域向けのバイトニックソートを表している．

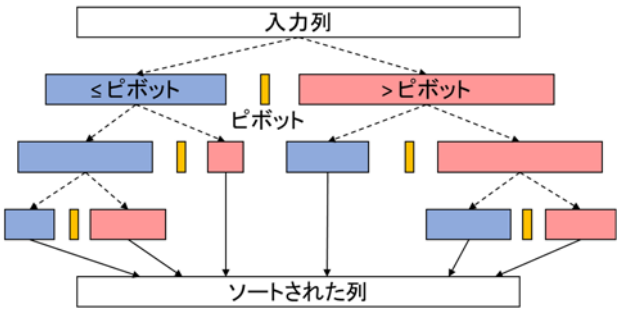


図 4　Bramas のアルゴリズムの概要

## 4. ハイブリッドソート

文献[7]のアルゴリズムはクイックソートが主要なアルゴリズムとなっているため，分割によって得られた列が 512 ビットの倍数になっているとは限らない．したがって，AVX-512 命令を活用する際に 512 ビットで区切れずに余ってしまった部分は 512 ビットにあわせるために不要な要素を含んで処理をする必要がある．入力列が大きくなった場合，クイックソートによる分割は多くなるため，それに伴い無駄な処理も増えると考えられる．したがって我々はまず入力列を分割することで複数の小さな列を作り，それらに対して文献[7]のアルゴリズムを適用してソートすることでバイトニック列を作る．その後 AVX-512 命令を活用したバイトニックソートによって全体をソートすることで文献[7]のアルゴリズムによる無駄を削減し，入力列が大きい場合に効率の良いソートアルゴリズムを提案する．

我々が提案するハイブリッドソートを整数型で実装する．整数型のデータサイズは 32 ビットであるため，512 ビットの SIMD ベクトル 1 つに対して 16 個の要素を格納することができる．

### 4.1 処理の概要

ベクトル演算は基本的に 2 つのベクトルを用いて処理を行うため，バイトニックソートを行う際に AVX-512 命令を効率良く活用するためには，バイトニック列のサイズが 512 ビットの$2^m$倍となっていることが望ましい．そこで，本アルゴリズムでは，まず入力列を 512 ビットの$2^m$倍ごとに分割する．分割された入力列のそれぞれを基本列と呼ぶこととする．次に基本列が交互に昇順，降順となるように各基本列を文献[7]のアルゴリズムを用いてソートする．これによって基本列の 2 倍のサイズのバイトニック列ができる．最後にこのバイトニック列をバイトニックソートすることでソートが完了する．このアルゴリズムの概要図を図 5 に示す．赤の矢印は昇順にソートすることを表しており，青の矢印は降順にソートすることを表している．また，このアルゴリズムをアルゴリズム 1 に示す．VEC_SIZE は SIMD ベクトルに格納できる要素数のことを表しており，扱うデータの型は整数型であるため 16 としている．

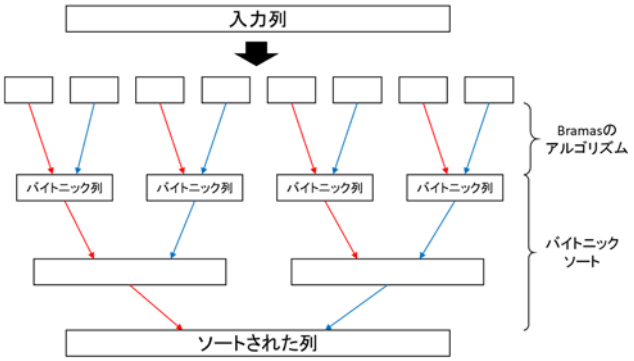


図 5　ハイブリッドソートの概要

**アルゴリズム 1**：ハイブリッドソート

```
入力　array：ソートしていない列
　　D：分割数
出力　array：ソート済みの列
1  Function hybrid_sort(array, D)
2    While i=0 to N-1
3      If i mod (2×N/D) = 0 Then
4        Bramas_sort_asc_order(array, i, i+N/D+1)
5      Else
6        Bramas_sort_desc_order(array, i, i+N/D+1)
7      End
8      i = i+N/D
9    End
10   While bt_size=2×N/D to N
11     While swap_block=bt_size to VEC_SIZE×2
12       orver32_btsort(array, bt_array, swap_block)
13       swap_block = swap_block/2
14     End
15     16elements _btsort(array, bt_array)
16     8elements _btsort(array, bt_array)
17     bt_size = bt_size×2
18   End
19 End
```

我々が提案する手法は並列化した際に高速でソートすることができるバイトニックソートを元にしているため，OpenMP を用いて並列化している．バイトニックソートは$2^m$の要素を持ち，一度に並列に処理できる数は$2^{m-1}$であるため，スレッド数$T$は$2^k$（$1 \leq k \leq m-1$）であることが望

ましい.

### 4.2 SIMD バイトニックソート

ベクトル 2 つ分以上, すなわち整数型で 32 要素以上のバイトニック列をさらに 2 倍の要素数のバイトニック列にするために大きく分けて 3 つのステップで処理を行う.

最初のステップとしてスワップブロックの要素数が 32 以上のときの処理を説明する. アルゴリズムをアルゴリズム 2 に示す. このとき, 連続した 16 要素と, それの比較対象となる連続した 16 要素をそれぞれレジスタにロードし, 単純に比較・交換をすればよい.

**アルゴリズム 2**: 要素数が 32 以上のスワップブロックのソート

```
   入力  array : 要素数が 32 以上のスワップブロック
         bt_array : バイトニック列のサイズ
         swap_block : スワップブロックのサイズ
   出力  array : 要素数が半分のスワップブロック
 1 Function over32_btsort(array, bt_array, swap_block)
 2   While i = 0 to N-1
 3     vec1 = load(array, i-(i mod swap_block)/2)
 4     vec2 = load(array, i-(i mod swap_block)/2+swap_block/2)
 5     If (i-(i mod swap_block)/2) mod (2×bt_array)
         < bt_array Then
 6       vec1 = min(vec1, vec2)
         vec2 = max(vec1, vec2)
 7     Else
 8       vec1 = max(vec1, vec2)
         vec2 = min(vec1, vec2)
 9     End
11     array[i-(i mod swap_block)/2
           : i-(i mod swap_block)/2+VEC_SIZE] ← vec1
       array[i-(i mod swap_block)/2+swap_block/2
12         : i-(i mod swap_block)/2+swap_block/2 + VEC_SIZE]
         ← vec2
13     i = i+VEC_SIZE×2
14   End
15 End
```

次のステップとしてスワップブロックの要素数が 16 のときの処理を説明する. アルゴリズムをアルゴリズム 3 に示す. このとき前半の 8 要素と後半の 8 要素を比較しなければならない. しかし 1 つのベクトルのみではベクトル演算ができないため, 効率よく AVX-512 命令を活用するために連続した 2 つのスワップブロックをまとめてソートする. まず連続した 2 つのスワップブロックをそれぞれレジスタにロードする. 各ベクトルをそれぞれベクトル 1, ベクトル 2 と呼ぶこととする. ロードした後, ベクトル 1 の後半がベクトル 2 の前半に, ベクトル 2 の前半がベクトル 1 の後半になるように並び替えを行う. その後 2 つのベクトルで比較・交換を行い, 元の位置に並び替える.

**アルゴリズム 3**: 要素数が 16 のスワップブロックのソート

```
   入力  array : 要素数が 16 のスワップブロック
         bt_array : バイトニック列のサイズ
   出力  array : 要素数が 8 のスワップブロック
 1 Function 16elements_btsort(array, bt_array, swap_block)
 2   While i = 0 to N-1
 3     vec1 = load(array, i)
 4     vec2 = load(array, i+VEC_SIZE)
 5     vec1 = vec1[0,1,2,3,4,5,6,7],vec2[8,9,10,11,12,13,14,15]
       vec2 = vec1[8,9,10,11,12,13,14,15],vec2[0,1,2,3,4,5,6,7]
 6     If (i mod (2×bt_array)< bt_array Then
 7       vec1 = min(vec1, vec2)
         vec2 = max(vec1, vec2)
 8     Else
 9       vec1 = max(vec1, vec2)
         vec2 = min(vec1, vec2)
10     End
11     vec1 = vec1[0,1,2,3,4,5,6,7],vec2[0,1,2,3,4,5,6,7]
       vec2 =
         vec1[8,9,10,11,12,13,14,15],vec2[8,9,10,11,12,13,14,15]
12     array[i : i+VEC_SIZE] ← vec1
13     array[i+VEC_SIZE : i+VEC_SIZE×2] ← vec2
14     i = i+VEC_SIZE×2
15   End
16 End
```

最後のステップとしてスワップブロックの要素数が 8 のときの処理を説明する. このときにレジスタ上で通常のバイトニックソートと同じ手順でソートしようとすると, スワップブロックがさらに小さくなった際に比較する値が同じベクトルに存在してしまうため, 多くの並べ替えが必要となる. したがってこのステップではより効率的に処理を行うために 1 つ先の処理を想定して次に比較する値が別のベクトルにあるように比較・交換を行う. このアルゴリズムをアルゴリズム 4 に示し, 手順を図 6, 動作の例を図 7 に示す. この 3 つのステップで処理を繰り返し行うことで入力列と同じ長さのバイトニック列を作り, 最後にもう一度この 3 つのステップで処理をすることでバイトニックソートが完了する.

**アルゴリズム 4**：要素数が 8 のスワップブロックのソート

```
入力　array：要素数が 8 のスワップブロック
      bt_array：バイトニック列のサイズ
出力　array：昇順もしくは降順にソート済みの列
1  Function 8elements _btsort(array, bt_array, swap_block)
2    While i = 0 to N-1
3      vec1 = load(array, i)
4      vec2 = load(array, i+VEC_SIZE)
5      vec1 = vec1[0,1,2,3,8,9,10,11],vec2[0,1,2,3,8,9,10,11]
       vec2 = vec2[4,5,6,7,12,13,14,15],vec1[4,5,6,7,12,13,14,15]
6      If (i mod (2×bt_array) ＜ bt_array Then
7        vec1 = min(vec1, vec2, mask=0x3333)
         vec1 = max(vec1, vec2, mask=0xCCCC)
         vec2 = max(vec1, vec2, mask=0x3333)
         vec2 = min(vec1, vec2, mask=0xCCCC)
8      vec2 = vec2[2,3,0,1,6,7,4,5,10,11,8,9,14,15,12,13]
9        vec1 = min(vec1, vec2, mask=0x5555)
         vec1 = max(vec1, vec2, mask=0xAAAA)
         vec2 = max(vec1, vec2, mask=0x5555)
         vec2 = min(vec1, vec2, mask=0xAAAA)
10     vec2 = vec2[1,0,3,2,5,4,7,6,9,8,11,10,13,12,15,14]
11       vec1 = min(vec1, vec2)
         vec2 = max(vec1, vec2)
12     Else
13       vec1 = min(vec1, vec2, mask=0xCCCC)
         vec1 = max(vec1, vec2, mask=0x3333)
         vec2 = max(vec1, vec2, mask=0xCCCC)
         vec2 = min(vec1, vec2, mask=0x3333)
14     vec2 = vec2[2,3,0,1,6,7,4,5,10,11,8,9,14,15,12,13]
15       vec1 = min(vec1, vec2, mask=0xAAAA)
         vec1 = max(vec1, vec2, mask=0x5555)
         vec2 = max(vec1, vec2, mask=0xAAAA)
         vec2 = min(vec1, vec2, mask=0x5555)
16     vec2 = vec2[1,0,3,2,5,4,7,6,9,8,11,10,13,12,15,14]
17       vec1 = min(vec1, vec2)
         vec2 = max(vec1, vec2)
18     End
19     vec1 = vec1[0],vec2[0],vec1[1],vec2[1],…,vec1[7],vec2[7]
       vec2 =
         vec1[8],vec2[8],vec1[9],vec2[9],…,vec1[15],vec2[15]
20     array[i : i+VEC_SIZE] ← vec1
21     array[i+VEC_SIZE : i+VEC_SIZE×2] ← vec2
22     i = i+VEC_SIZE×2
23   End
24 End
```

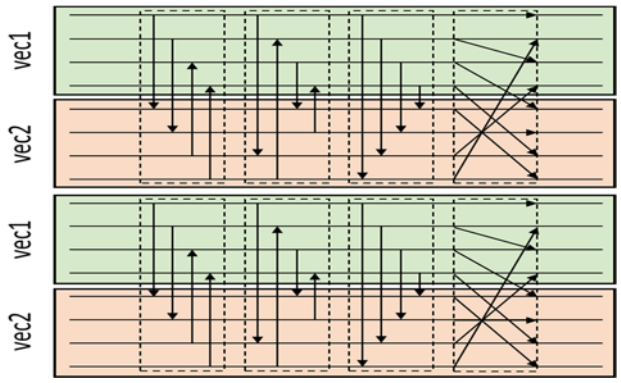


図 6　スワップブロックの要素数が 8 の場合の処理手順

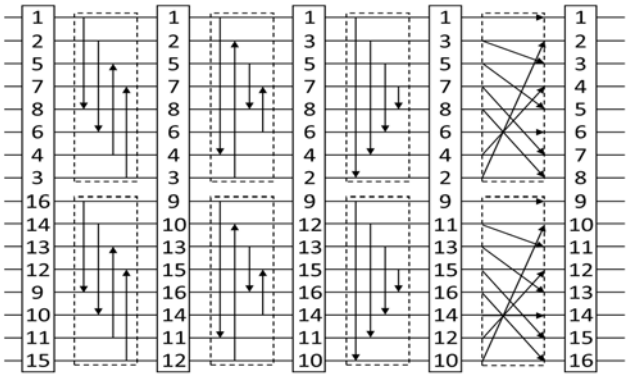

図 7　スワップブロックの要素数が 8 の場合の動作例

## 5. 評価

我々が提案するハイブリッドソートアルゴリズムを Intel Xeon Phi CPU 7210 上で実装し，文献[7]のアルゴリズムと比較する．ハイブリッドソートの入力列の分割数は 2，4，8，16，32 の 5 通りとし，処理に活用したスレッド数はいずれも 256 とした．また，用いたコンパイラは Intel C++ Compiler 17.0.1 で，最適化オプションは O3 とした．測定結果を図 8 に示す．横軸は要素数を低を 2 とする対数に変換した値であり，縦軸は要素数をソートにかかった時間で割った値である．入力列を 2 分割するだけでも高速に処理することができた．また分割数が 32 のとき要素数が$2^{28}$の場合に文献[7]のアルゴリズムとほぼ同じ結果になったが，入力列の大きい場合にいずれの分割数でも高速に処理することができた．分割数が 2 のとき大きく上下する部分が見られるが，これは大部分が文献[7]のアルゴリズムで構成されているためであると考えられる．

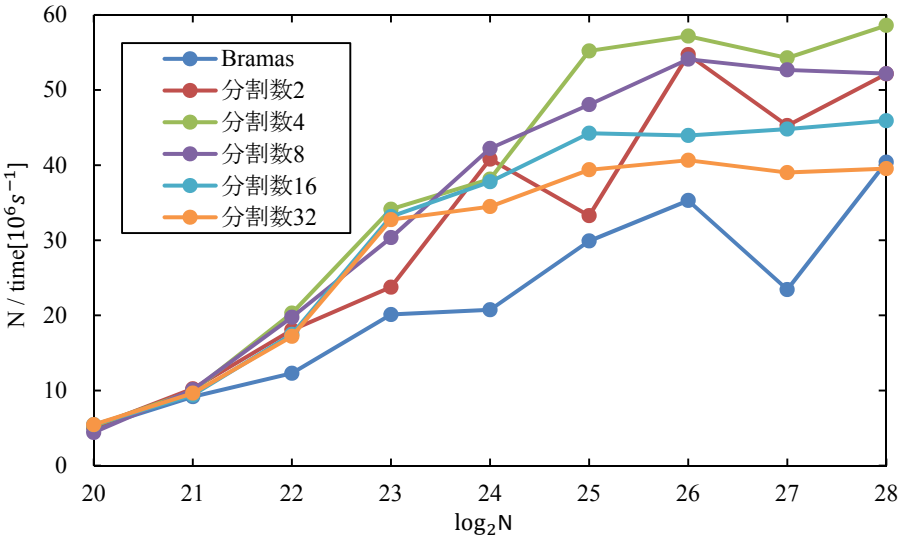


図 8　Bramas のアルゴリズムと提案手法の比較

## 6. 終わりに

我々は AVX-512 命令を活用した Xeon Phi 向けのソートアルゴリズムとして，文献[7]のアルゴリズムとバイトニックソートを組み合わせたハイブリッドソートを提案し，これを KNL 上で実装し評価をすることで整数型で従来よりも高速なソートが行えることを示した．今後は浮動小数型での実装と，要素数やスレッド数と処理速度の関連を調べ最適な分割数を見つけることが課題である．

## 参考文献

[1] Hoare, C. A. R. Quicksort. The Computer Journal. 1962, vol. 5, p. 10-15.
[2] Batcher, K. E. Sorting networks and their applications. AFIPS '68 (Spring) Proceedings. 1968, p. 307-314.
[3] “Intel Xeon Phi 製品ファミリー”. https://www.intel.co.jp/content/www/jp/ja/products/processors/xeon-phi.html, (参照 2018-04-04).
[4] “製品の開発コード名 Knights Corner”. https://ark.intel.com/ja/products/codename/57721/Knights-Corner, (参照 2018-04-04).
[5] “製品の開発コード名 Knights Landing”. https://ark.intel.com/ja/products/codename/48999/Knights-Landing, (参照 2018-04-04).
[6] “Intel Architecture Instruction Set Extensions Programming Reference”, https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf, (参照 2018-04-04).
[7] Bramas, B. A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake. International Journal of Advanced Computer Science and Applications. 2017, vol. 8, no. 10, p. 337-344.