VMX 命令セットを用いる高速なソートアルゴリズム

井上 拓[†] 森山孝男[†]
小松秀昭[†] 中谷登志男[†]

データを値の順番に並べ直すソート処理は多くのソフトウェアで使用される最も基本的な操作の 1 つであり，ソート処理の高速化は多くのワークロードの性能向上に寄与する．ソート処理は基本的な操作であるため，古くから多くのアルゴリズムが提案されているが，近年の高性能な汎用プロセッサの SIMD 命令を用いて高速にソートを行うことのできるアルゴリズムはこれまで提案されていない．そこで本研究では PowerPC アーキテクチャが持つ SIMD 命令セットである VMX を使用して，並列に処理を行うとともに分岐予測ミスの影響をなくすことで高速にソート処理を行うことのできるアルゴリズムを提案する．このアルゴリズムを実装し，PowerPC 970FX プロセッサ上で評価を行い，クイックソートと比較して最大で 5.6 倍の性能向上が得られることを示した．

A Fast Sorting Algorithm for VMX Instruction Set

Hiroshi Inoue,[†] Takao Moriyama,[†] Hideaki Komatsu[†] and Toshio Nakatani[†]

Sorting is one of the most common operations done by computers and used in variety of software. Sorting is a well-studied problem and hence there are many sorting algorithms and implementations available. However there is no sorting algorithm that effectively exploits SIMD execution units of recent general purpose processors. In this paper, we proposed a novel sorting algorithm for VMX instruction set of the PowerPC architecture. This algorithm divides input data in sorting phase and merges them after sort in order to exploit parallelism of the VMX instruction set. Also it removes stall cycles due to conditional branches by replacing them with vector minimum and vector maximum instructions. We implemented the algorithm and evaluated the performance on the PowerPC 970FX processor. Our results showed that our new algorithm achieved up to 5.6 times higher performance than quicksort.

1. はじめに

データを値の順番に並べ直すソート処理は，多くのソフトウェアで使用される最も基本的な操作の 1 つである．そのため，ソート処理の高速化により用途を問わず多くのワークロードで性能の向上が期待できる．ソートは基本的な操作であるため，古くから多くのアルゴリズムが提案されている[1)] が，近年の高性能な汎用プロセッサの SIMD 命令セット（たとえば PowerPC の VMX 命令セット[2)] や IA-32 の SSE 命令セット[3)]）を用いて高速にソートを行うアルゴリズムはこれまで提案されていない．これらの SIMD 命令セットではメモリ内で連続した複数のデータをベクタレジスタにロードし，一度に処理を行うことができる．たとえば VMX では 128 ビット長のベクタレジスタを用いることで 32 ビット整数データであれば連続した 4 つのデータを一度にロード，ストアすることができる．しかし，このメモリアクセスは 128 ビット境界にアラインした連続した 128 ビットアクセスに限られるため，実用的に多く用いられているクイックソートやヒープソートといった，メモリへのアクセスパターンがデータに依存して不連続になるソートアルゴリズムは VMX などの SIMD 命令セットを用いて高速に実装することが困難である．SIMD 命令セットを用いて効率的にソートを行うためのアルゴリズムには，通常のソートアルゴリズムへの要求に加えて，(1) 連続的なメモリアクセスパターン，(2) SIMD 命令の並列度を活かすための細粒度並列性の 2 つの特長を持つことが求められる．

そこで，本研究では PowerPC アーキテクチャが持つ SIMD 命令セットである VMX を主な対象として，SIMD 命令セットに適した新たなソートアルゴリズムを提案する．

† 日本アイ・ビー・エム株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan

このアルゴリズムではメモリへのアクセスパターンがデータに依存しないソートアルゴリズムであるコームソートをベースとする．さらに入力データを SIMD 命令の並列度に応じた数の独立した系列のデータとして扱い，それぞれを並列にソートすることで SIMD 命令の並列性を活かすことができる．加えて本アルゴリズムは大小比較の結果に応じてデータを交換する操作を vector minimum 命令と vector maximum 命令を用いて条件分岐なしに実装することで，条件分岐にともなう分岐予測ミスの影響を取り除き SIMD 命令の並列度以上の性能向上を得ることができる．

本アルゴリズムではデータを複数系列の独立データとして取り扱うことで VMX 命令の並列度を活かせるが，最終的にソートされた各系列のマージ処理が必要となる．マージ処理は多数の条件分岐を含むため要素数が少ない場合にはソートと比較して無視できない処理時間を必要とする．そこで本アルゴリズムではさらに系列を越えるデータの比較・交換処理を追加することでこのマージ処理を条件分岐なしで行えるようにし高速化する．

本アルゴリズムを VMX 命令を用いて実装し，PowerPC 970FX プロセッサ上で評価を行った結果，クイックソートと比較して最大で 5.6 倍の性能向上が確認された．

2. 従来技術

2.1 コームソート

ソートのアルゴリズムとして，最も基本的なものにバブルソートがある．これは **図 1** に示すように，ある要素を次の要素と比較し，次の要素よりも大きな値だった場合にデータを交換することを繰り返すものである．バブルソートは直感的である反面，計算量が $O(N^2)$ のオーダと，クイックソートなど他のソートアルゴリズムと比較して遅いため実用的にはほとんど用いられない．バブルソートをベースにして計算量のオーダを改善したアルゴリズムとしてコームソート（combsort）がある[4]．バブルソートではつねに 1 つ隣の要素としか比較・交換を行わないが，図 1 に示すようにコームソートではより離れた要素との比較・交換を行い，要素の間隔（コード中 gap と表記）を徐々に狭めていくことで，まず大まかにソートし最終的には完全にソートされた状態にする．コームソートの計算量はクイックソートなどと同様にデータに依存するが平均的にはほぼ $N\log(N)$ に比例した計算量[4]となる．比較する要素間の距離は例に示したようにイタレーションごとに 1/1.3 倍にしていくのが一般的である．コームソートは計算量のオーダ的にはクイックソートなどと同程度であるが，多くの場合，性能的にクイックソートに及ばず，また広い範囲のメモリにアクセスを繰り返すためキャッシュヒット率の面で不利であることなどから，実用的にはほとんど用いられていない．

```
[ bubble sort ]
  for (i = 0; i < N-1; i++) {
    for (j = 0; j < N-1-i; j++) {
      if (a[j] > a[j+1]) swap(a[j], a[j+1]);
    }
  }


[ combsort ]
  gap = (N * 10) / 13;
  while (gap > 1) {
    for (j = 0; j < N - gap; j++)
      if (a[j] > a[j + gap]) swap(a[j], a[j+gap]);
    gap = (gap * 10) / 13;
  }
  do {
    cont = 0;
    for (j = 0; j < N - 1; j++) {
      if (a[j] > a[j + 1]) {
        swap(a[j], a[j+1]);
        cont = 1;
      }
    }
  } while(cont);
```

図 1 バブルソートとコームソートのアルゴリズム
Fig. 1 Pseudo-code of bubble sort and combsort.

2.2 VMX（AltiVec）命令セット

Vector Multimedia eXtension（VMX，AltiVec とも呼ばれる）[2] は，PowerPC の SIMD 命令セットであり，PowerPC 970 シリーズ[5] や，Cell Broadband Engine[6] の PowerPC コアなどで実装されている．VMX では 128 ビット長のベクタレジスタ（以下 VMX レジスタと表記）を用いて，8 ビット × 16 スロット，16 ビット × 8 スロット，32 ビット × 4 スロットのいずれかの形式での並列処理が可能である．VMX 命令セットのうち本研究で使用する，min 命令，max 命令，compare 命令，select 命令，permute 命令のそれぞれについて挙動を説明する．

min，max，compare の各命令は 2 つの VMX レジスタを入力，1 つの VMX レジスタを出力とし，16 バイトの VMX レジスタのデータをスロット 1 からスロット 4 の 4 つの 32 ビット整数の列として取り扱う．min 命令（vec_min）は 2 つの入力レジスタの各スロットの値を比較し小さい方を出力レジスタの対応するスロットに出力する．max 命令（vec_max）は同様に各スロットの大きい方の値を出力する．compare 命令のうち，たとえば vec_cmpgt（compare greater than）命令は各スロットごとの値を比較し，第 1 引数のレジスタの値の方が第 2 引数のレジスタの値より大きかった場合には出力レジスタの対応するスロットのすべて

のビットを 1 にし，第 2 引数の値の方が大きいか等しかった場合にはすべてのビットを 0 にする．

select 命令（vec_sel）は 3 つの VMX レジスタを入力，1 つの VMX レジスタを出力とし，出力レジスタの各ビットについて第 3 引数の対応するビットが 0 だった場合には第 1 引数の対応するビットを，1 だった場合には第 2 引数の対応するビットを出力する．

permute 命令（vec_perm）も 3 つの VMX レジスタを入力，1 つの VMX レジスタを出力とし，VMX レジスタのデータを 16 個の 1 バイト値の列として扱う．この命令ではまず第 1 引数と第 2 引数をこの順で結合し 32 個の 1 バイト値の配列とする．そのうえで第 3 引数の各スロットの下位 5 ビットの値をインデックスとして配列からデータを取り出し，出力レジスタの対応するスロットに出力する．これによりバイト単位で重複も含む任意の順にデータの並べ替えを行うことができる．

これらの命令は VMX 命令セットのものであるが，他の多くの SIMD 命令セットでも類似の命令が存在し，存在しない場合でも他の命令で代替可能な場合がある．たとえば，Cell Broadband Engine の SPU と呼ばれるコアの SIMD 命令セット[7] では，vec_min および vec_max 相当の命令が存在しないが，これは compare 命令と select 命令で代替が可能である．IA-32 の SSE 命令セットでは，vec_sel 相当の命令が存在しないがこれは vector and 命令と vector or 命令などを用いて代替が可能である．

3. 関 連 研 究

SIMD アーキテクチャのプロセッサを用いて高速にソートを行うアルゴリズムとしては，Graphic Processing Unit（GPU）を用いてソートを行うアルゴリズムが提案されている．GPU ハードウェアの特長としては，並列度の高い画像処理に特化しているため汎用プロセッサと比較して非常に多数の SIMD 型演算器を備えることや，GPU と広帯域のバスで接続されたグラフィックメモリを持つことがあげられる．Purcell ら[8] は GPU 上での画像生成の一部として，バイトニックマージソートを GPU 上で実装する手法を示した．Govindaraju ら[9),10] は，GPU を画像生成に限らないアクセラレータとして使い，汎用プロセッサ上のプログラムのソート処理をオフロードできることを示した．この手法では GPU でのメモリアクセスの制限（たとえば任意のアドレスへのデータ書き込みができない点）を回避し，GPU が持つ多数の演算器を効率良く使用するために，Periodic Balanced Sorting Network[9] やバイトニックマージソート[10] を改良したアルゴリズムを提案している．このアルゴリズムでは SIMD 命令の並列度を活かすためにデータを 4 系列に分け独立にソートした後，それらのマージ処理を行う．これらのアルゴリズムは GPU が備える多数の SIMD 演算器を効率良く利用できるため，結果として巨大なデータのソートについて CPU 上でのクイックソートを超える性能を示した．

Govindaraju ら[10] のアルゴリズムは本研究で提案するアルゴリズムと同様に，(1) 連続的なメモリアクセスパターン，(2) SIMD 命令の並列度を活かすための細粒度並列性という SIMD 命令セットに適したアルゴリズムに求められる 2 つの特長を備えている．さらにデータの比較・交換を基本操作とするので vector minimum 命令と vector maximum 命令を用いることで分岐遅延なしに実装できる点でも本アルゴリズムと同様である．本研究で提案するアルゴリズムの利点としては，本アルゴリズムの計算量はベースとなったコームソートと同様にほぼ $N \log N$ に比例し，Govindaraju らのアルゴリズムの $N \log(N)^2$ オーダの計算量と比較して小さい点があげられる．さらに本研究のアルゴリズムでは，ソート後のマージ処理を条件分岐なしに行うことができる点も大きな利点である．一方，Govindaraju らのアルゴリズムの利点としては，全データを多数のブロックに分けそれらのブロックを再帰的にマージしていくアルゴリズムを用いることで，各ブロックのマージ処理を独立に処理でき，本研究のアルゴリズムよりも高いスレッドレベルの並列性が得られる点があげられる．本研究のアルゴリズムではスレッドレベルの並列化は容易ではないが，スーパスカラの SIMD 型演算器を効率良く動作させるために必要な細粒度の並列性は十分にあるうえ，計算量のオーダが Govindaraju らのアルゴリズムよりも小さく，さらにソート後の系列間のマージ処理を条件分岐なしに行うことができる特長がある．そのため，並列度の限られる汎用プロセッサでの実施には本アルゴリズムの方が適していると考えられる．また，スレッドレベルの並列性が必要な場合には並列マージソートなどと組み合わせ，各ブロックを本アルゴリズムで並列に処理後，マージソートで各ブロックのマージを行うことなどで対応が可能である．

4. アルゴリズムの詳細

4.1 32 ビット整数のソート

本研究で提案するソートアルゴリズムを VMX 命令を用いて 32 ビット整数をソートする例を用いて説明す

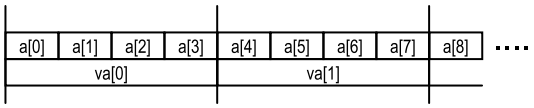


図 2 ソート対象データのメモリレイアウト
Fig. 2 Data structure of the array to be sorted.

る．ソートする N 個の int 型（32 ビット整数）データ a[i] はメモリ内で**図 2** に示すように配置され，vector int 型データ va[i] は 4 つの int 型のデータ a[i*4]〜a[i*4+3] からなるものとする．以下では説明のために要素数は 4 の倍数であり a[0] は 128 ビット境界にアラインされているものとする．この条件を満たさない場合でも若干の処理の追加のみで対応でき，この処理は先頭要素と最終要素のみに限定されるため性能への影響は小さい．また本手法はこのデータ型や並列度に限定されるものではなく，たとえば VMX を用いて単精度浮動小数点型のデータのソートを行うことも可能である．

バブルソートやコームソートの基本操作である大小比較の結果によるデータの交換は VMX 命令を用いて**図 3** 上側のコードのように記述できる．図 3 下側の VMX を用いない命令列では各要素を別々に処理するため，命令数が多いだけではなく要素の処理ごとに条件分岐が含まれ，これらの条件分岐は入力データに依存し，分岐予測がヒットしにくいため，予測ミスにともなうパイプラインストールで命令数以上に実行時間が必要になる．結果として比較・交換の処理を VMX 命令を用いて実装することでこの予測ミスにともなうペナルティを受けず，命令の並列度以上の性能向上が得られる．

図 3 のように vec_min/vec_max を用いて N/4 個の vector int 型データに対してコームソートと同様の操作を行うことで，N 個の int 型データ列は a[4n], a[4n+1], a[4n+2], a[4n+3]（ただし n は 0, 1, 2, ..., N/4−1）の 4 つの系列内でそれぞれソートされた状態になる．この VMX でのソート後にマージソートと同様に 4 つの系列の大小比較を行いながらマージ操作を行うことで，全体のデータに対してのソートを行うことができる．このとき，マージ処理のために追加の作業用メモリ領域が必要になる．最後のマージ処理は $O(N)$ の計算量となるため，各系列内でのソートの計算量と比較して大きなデータに対しては全体の計算時間に占める割合は少なくなる．

本アルゴリズムの利点をまとめると，以下の 2 点があげられ，結果として大きな高速化が期待できる．

(1) ベクタ処理で 4 データの処理が並列に行える．

```
[ with VMX ]
  tmp = va[0];
  va[0] = vec_min(tmp, va[1]);
  va[1] = vec_max(tmp, va[1]);

[ without VMX ]
  if (a[0] > a[4]) swap(a[0], a[4]);
  if (a[1] > a[5]) swap(a[1], a[5]);
  if (a[2] > a[6]) swap(a[2], a[6]);
  if (a[3] > a[7]) swap(a[3], a[7]);
```

図 3 データの大小比較・交換手法
Fig. 3 Implementations of comparing and swapping values with and without VMX.

(2) コントロールフローがデータに依存して変わらないため，分岐予測ミスによるペナルティが生じなくなる．

ただし，本アルゴリズムでも $gap = 1$ のループでのソート完了の判定のために条件分岐が必要となるが，このループは $gap > 1$ の場合のループと比較して実行回数が少ないので性能へ及ぼす影響は小さい．

4.2 マージ処理での条件分岐の除去

本アルゴリズムでは，SIMD 命令の並列度を活かすためにデータを 4 系列の独立したデータとして扱うために，すでに述べたように最後に系列ごとのマージ処理が必要になる．マージ処理は計算量のオーダがソート自体よりも小さいため要素数が大きい場合にはその影響を無視できるが，分岐予測ミスによるペナルティが大きく，要素数が小さい場合にはソートと比較してその処理時間を無視できなくなる．

そこで本アルゴリズムでは以下のように系列を越えたデータの比較・交換の操作を追加することでマージ処理を条件分岐なしに行えるようにする．まず，ソートの各段階において**図 4** のような処理を追加する．コード中で vector_cmpswap は上で述べた vec_min/vec_max を用いて各スロットの中での比較・交換を行う操作を示し，vector_cmpswap_skew は比較・交換を 1 スロットずらして隣接する系列との間で行う操作である（**図 5**）．vector_cmpswap_skew も VMX を使用して条件分岐なしに効率良く実装することができる．この処理を追加することにより最終的に各系列内でデータがソートされていることに加えて，2 番目の系列の先頭データは 1 番目の系列の最終データよりも大きいことが保証され，同様に第 3 系列，第 4 系列の順で値が大きくなることが保証される．その結果 int 型配列 a は，a[0], a[4], a[8], ... a[N−4], a[1], a[5], ... a[N−3], a[2], a[6], ..., a[N−2], a[3], a[7], ... a[N−1] の順でソートされた状態になる．これを正しく整列するのは条件分岐を含まないデータの移動のみで可能であり分岐予測ミスのペナルティがないため，条件分岐のあ

```
gap = (N/4 * 10) / 13;
while (gap > 1) {
  /* comparison within each sequences */
  for (j = 0; j < N/4 - gap; j++)
    vector_cmpswap(va[j], va[j+gap]);
  /* comparison between neighbor sequences */
  k = N/4 - gap + 1;
  for (j = 0; j< N/4 - k; j++)
    vector_cmpswap_skew(va[j], va[j+k]);
  /* dividing gap by the shrink factor */
  gap = (gap * 10) / 13;
}
do {
  cont = 0;
  /* comparison within each sequences */
  for (j = 0; j < N/4 - 1; j++) {
    vaj0 = va[j];
    vector_cmpswap(va[j], va[j+1]);
    cont |= (vaj0 != va[j]);
  }
  /* comparison between neighbor sequences */
  vaj0 = va[0];
  vector_cmpswap_skew(va[0], va[N/4-1]);
  cont |= (vaj0 != va[0]);
} while(cont);
```

図 4 マージ処理での条件分岐の除去
Fig. 4 Removing conditionla branches from the merging phase.

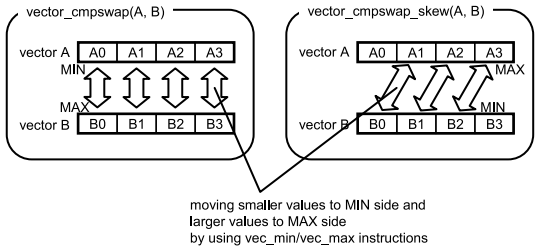


図 5 vector_cmpswap 処理および vector_cmpswap_skew 処理
Fig. 5 The vector_cmpswap and vector_cmpswap_skew operations.

るマージ処理と比較して高速に行うことができる．

この vector_cmpswap_skew 操作を追加した場合，系列を越えるデータ交換が起こってしまうと各系列内で小さなデータが末尾近くに，大きなデータが系列の先頭近くに来てしまい，それらを正しい位置に移動するために大きな処理時間がかかる場合があるので，このような交換が起こる確率を小さくなるようにしなくてはならない．そこでさらにソートを開始する前に，各 vector int 型データ内でスロット 1 からスロット 4 に向かってデータが大きくなるように並べなおす前処理を追加する（**図 6**）．コード中で，vector_swap4 および vector_swap2 は図 6 の下の図に示すように，2 つの vector int 型データ内で要素を交換する操作である．vector_cmpswap と異なり比較などは行わずつねに入れ替えを行う．図 6 のコード中で，最初の vector_cmpswap および vector_swap4 で小さいデータはスロット 1，2 へ，大きいデータはスロット 3，4

```
for (i = 0; i < N/4 - 1; i++) {
  vector_cmpswap(va[i], va[i+1]);
  vector_swap4(va[i], va[i+1]);
  vector_cmpswap(va[i], va[i+1]);
  vector_swap2(va[i], va[i+1]);
}
```

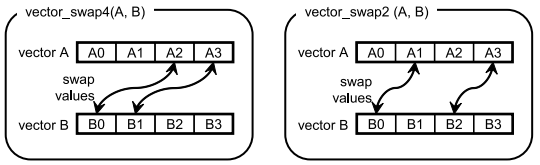


図 6 マージ処理での条件分岐除去のための前処理
Fig. 6 Pseudo-code of the preconditioning.

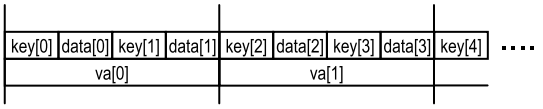
へまとめられる．さらに次の vector_cmpswap および vector_swap2 でスロット 1 と 2 およびスロット 3 と 4 で，それぞれ小さいデータは左のスロット，大きいデータは右のスロットに移動する．その結果，ほぼ小さいデータほど左のスロットに移動したデータ配置となる．これによりソート中に系列を越えるデータ移動が生じる回数を大きく減らすことができる．

以上のような前処理および系列を超える比較をソートに追加することで，マージ処理での条件分岐を除去することができる反面，この処理は大きな要素数のデータをソートする場合には逆に処理量が増加し性能の低下を招く．そのため，ソートするデータ数に応じてマージ処理での条件分岐を除去するかどうかを決める必要がある．

4.3 32 ビットキーと 32 ビットデータのペアのソート

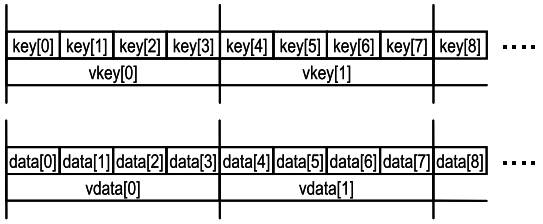
ソートの実用的な用途としては構造体などのデータ構造をそこに含まれるキーによってソートする場合がほとんどである．4.1 節で述べたアルゴリズムは単純に整数型データ配列をソートするための手法であったが，本節ではこれを拡張し，キー以外の付加データを持つ，キーとデータのペアをキーの値でソートするアルゴリズムを示す．以下ではキーおよびデータをそれぞれ 32 ビット長とする．

図 7 に示すようにメモリ内で 1 つの配列にそれぞれが 32 ビット長の key と data が交互に配置されたペアを key の値によってソートする．図 7 下図のコードを用いることで key の値に応じて {key, data} のペアを条件分岐なしで移動することができる．すなわち 4 行目の vec_perm 命令で，スロット 1 とスロット 3 に含まれるキーに対する比較結果をそれぞれスロット 2 とスロット 4 へコピーすることでつねにデータがキーの比較結果によって移動するようにしている．他の部分に関しては前述のアルゴリズムから特に変更の必要は




```
pat={0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,
0x08,0x09,0x0A,0x0B,0x08,0x09,0x0A,0x0B};
tmp=va[0];
mask=vec_cmpgt(va[0], va[1]); // create mask
mask=vec_perm(mask, mask, pattern);
va[0]=vec_sel(tmp,va[1],mask); // select smaller
va[1]=vec_sel(va[1],tmp,mask); // select larger
```

図 7 キーとデータを 1 つの配列に持つペアのソート
Fig. 7 Implementation of comparing and swapping for array of {key, data} pairs.




```
mask = vec_cmpgt(vkey[0], vkey[1]);
tmp = vkey[0];
vkey[0] = vec_sel(tmp, vkey[1], mask);
vkey[1] = vec_sel(vkey[1], tmp, mask);
tmp = vdata[0];
vdata[0] = vec_sel(tmp, vdata[1], mask);
vdata[1] = vec_sel(vdata[1], tmp, mask);
```

図 8 キーとデータを別の配列に持つペアのソート
Fig. 8 Implementation of comparing and swapping for arrays of keys and data.

ない．

図 8 上図に示すようにキーと付加データが独立したメモリ領域に配置されている場合には，図 8 下のコードのように key 配列の比較結果の mask を用いて，key，data の両方を移動させることで同様に条件分岐なしにソートを行うことができる．

4.4 キャッシュを超えるサイズのデータのソート

本研究で提案するアルゴリズムは，データがキャッシュに収まる場合は非常に高速であるが，クイックソートと比較してメモリアクセスの局所性が低いため，データがキャッシュのサイズを超えてしまった場合にはキャッシュヒット率が低くなり性能が低下してしまう．その場合にはデータアクセスの局所性の高いクイックソートやマージソートと組み合わせることで大きなデータ全体を本アルゴリズムでソートするよりも高い性能を得られる．一般にクイックソートを用いる場合にはデータサイズが小さくなった後は挿入ソートなどのアルゴリズムに切り替えることが多いが，挿入ソートなどの代わりに本アルゴリズムを用いる場合にははるかに大きい要素数で切替えを行うことができる．

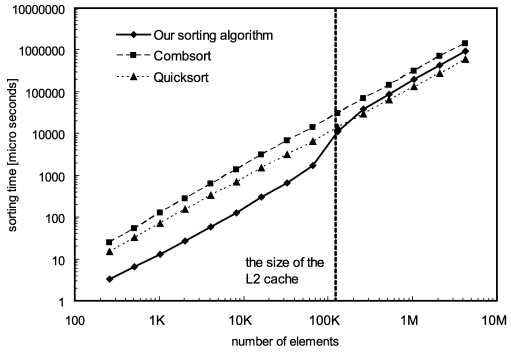


図 9 本アルゴリズムとコームソート，クイックソートの性能
Fig. 9 Performance comparison of our algorithm, combsort and quicksort (Smaller values are faster).

5. 性能評価

5.1 評価環境

本アルゴリズムについて 2.2 GHz の PowerPC 970FX プロセッサ（L2 キャッシュ512 KB）を搭載したマシン上で性能の評価を行った．OS には Linux-2.6.5，コンパイラには gcc-4.0.1 を用いた．プログラムは VMX のイントリンシック[11] を用いて VMX 命令を明示的に示した C 言語で記述しており，32 ビットプロセスとしてコンパイルした．また，比較対象として関連研究であげた Govindaraju ら[10] のバイトニックマージソートを改良したアルゴリズム，コームソートおよびクイックソートを実装した．Govindaraju らのアルゴリズムは本来は GPU 向けの手法であるが，これを本アルゴリズムと同様に VMX 命令をイントリンシックで明示的に記述して実装した．さらに Govindaraju らのアルゴリズムを 4.3 節で述べた手法と同様に拡張し，キーとデータのペアをソートする場合についても評価を行った．コームソートおよびクイックソートは VMX 命令を用いずに実装した．クイックソートでは要素数 8 以下で挿入ソートに切り替えている．ピボットの選択は以下のように行った．まず先頭および最後尾のデータを含む 9 つのデータを均等間隔に取り出し，3 つのデータからなる 3 グループを作る．各グループの中央値を選び，さらにそれら 3 データの中央値をピボットとして選択した．

5.2 評価結果

5.2.1 32 ビット整数ソートの性能

図 9 に本アルゴリズム，コームソートおよびクイックソートでランダムデータで初期化した 32 ビット整数配列をソートする場合の性能を示す．横軸は要素数

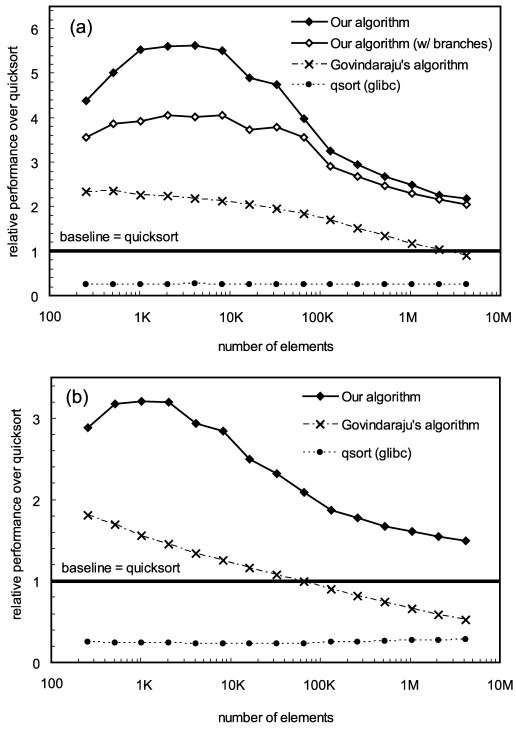


図 10 クイックソートからの性能向上率；(a) ランダムな 32 ビット整数 (b) ソート済み 32 ビット整数

Fig. 10 Relative performances of our algorithm and Govindaraju's algorithm over quicksort. (a) for random integers. (b) for presorted integers (Larger values are faster).

で，縦軸はソートにかかった時間を示す．両方の軸について対数表示を用いている．図 9 より明らかに本アルゴリズムではキャッシュサイズの影響を大きく受け，要素数 10 万程度から急激に性能が悪化していることが分かる．コームソートも本アルゴリズムと同様のメモリアクセスパターンを持つが，計算速度が遅いためキャッシュサイズを超えた場合の性能悪化が顕在化していない．

この評価結果より本アルゴリズムをクイックソートと組み合わせる場合，今回の環境ではクイックソートから本アルゴリズムに要素数 70,000 で切替えを行った．要素数 70,000 はデータサイズとしては 280 KB に相当する．この切替えを行う要素数は，データをクイックソートの 1 ステップでピボット以上と以下に分けた後，それぞれを本アルゴリズムでソートするより，データ全体を本アルゴリズムでソートする方が高い性能が得られる最大の要素数として求めた．

図 10 に本アルゴリズムをクイックソートと組み合わせた際の性能を示す．横軸は要素数，縦軸はクイックソートとの相対性能であり数字が大きいほど速いこ

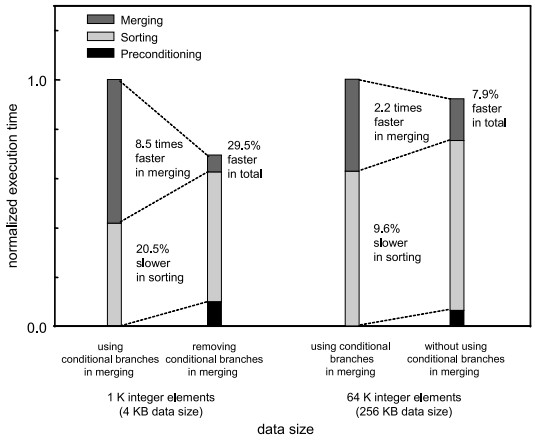


図 11 本アルゴリズムの実行時間の内訳

Fig. 11 Execution time breakdown with and without removing conditional branches in the merging phase (Shorter is faster).

とを示す．また 4.2 節に記述した最後のマージ処理で条件分岐を取り除く手法を用いた場合と用いない場合の結果を示した．比較として Govindaraju らのアルゴリズムおよび glibc-2.3.3 に含まれる qsort 関数の性能も示している．図 10 の (a) はソートする配列を乱数で初期化した場合，(b) はすでにソート済みであるデータで初期化した場合である．(a) よりランダムデータのソートでは本アルゴリズムはすべてのデータサイズについてクイックソートおよび Govindaraju らのアルゴリズムを上回っており，特に L2 キャッシュに収まるデータサイズではクイックソートの 4 倍以上，最大で 5.6 倍の性能向上が得られた．Govindaraju らのアルゴリズムは，本研究の手法と同様に条件分岐を用いないため分岐予測ミスのペナルティがなく，データサイズが小さい場合にはクイックソートよりも高い性能が得られたが，計算量のオーダが本アルゴリズムやクイックソートよりも大きいため，データサイズが大きくなるとクイックソートの性能を下回った．

また，(a) よりマージ処理で条件分岐を取り除く処理を用いることで，最大で 30%程度性能向上が得られることが分かる．この処理の効果を詳しく見るために，マージ処理で条件分岐を取り除いた場合と取り除かない場合について，2 つのデータサイズについて実行時間の詳細を**図 11** に示す．図より条件分岐なしに処理を行えることでマージ処理が大きく高速化し，ソートの処理量の増加と前処理の追加を加えても全体として性能向上が得られていることが分かる．前処理は最大で全体の 15%程度の処理時間を使用しているが，前処理を行わない場合には，4.2 節で述べたように系列を越えるデータ移動の回数が 1.5 倍程度に増加するとと

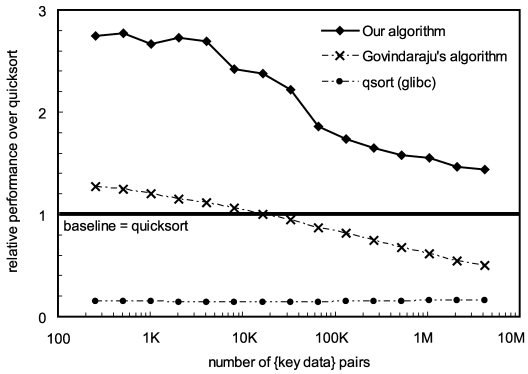


図 12 クイックソートからの性能向上率（32 ビットキーとデータのペア）

Fig. 12 Relative performances of our algorithm and Govindaraju's algorithm over quicksort for {key, data} pairs (Larger values are faster).

もに，最終的にソートを行う gap = 1 のループを回る回数が劇的に増加してしまう．たとえば 64 K 個のデータをソートする場合には，前処理を行う場合 gap = 1 のループを実行する回数は，スカラでのコームソートと同程度でありデータによって 4〜6 回であったのに対して，前処理を行わない場合には 3000〜8000 回程度であり，実用的な性能は得られなかった．このループはバブルソートと同じ動作であるため，ループに入る時点で少数でも最終的な位置から離れたデータが存在すると非常に多くの回数ループを回ってしまうことになり，前処理の効果が大きい．

(b) のソート済みデータを再度ソートする場合はピボットの選択が最適になるうえ，分岐予測がヒットするのでクイックソートにとって非常に有利な条件であるが，この条件でも本アルゴリズムはすべてのデータサイズでクイックソートおよび Govindaraju らのアルゴリズムを上回る性能を示した．この条件ではクイックソートも分岐ミスの影響をほぼ受けないことで，分岐ミスを取り除いたことによる相対的な性能向上は得られないが，SIMD 命令により 4 データを並列に処理できるため性能向上を得ることができる．

5.2.2 32 ビットキーと 32 ビットデータのペアのソートの性能

図 12 に 2 つの 32 ビット整数のペアをソートする場合の性能を示す．キーは乱数で初期化しており，図 10 に示したようにキーとデータを 1 つの配列内に交互に配置するメモリ配置を用いている．この場合は 32 ビット整数配列のソートよりも処理が複雑になるが，本アルゴリズムはすべてのデータサイズでクイックソートおよび Govindaraju らのアルゴリズムの性能を上回っている．

6. おわりに

本研究では PowerPC アーキテクチャが持つ SIMD 命令セットである VMX を主な対象として，SIMD 命令セットに適した高速なソートアルゴリズムを提案した．

このアルゴリズムは，(1) 連続データのロード・ストアのみのアクセスパターンである，(2) 入力データを SIMD 命令の並列度に応じた数の独立した系列データとして扱い，それぞれの系列を並列にソートすることで SIMD 命令の並列性を活かせる，(3) 大小比較の結果に応じてデータを交換する操作を vector minimum 命令と vector maximum 命令を用いて行うことができ，条件分岐にともなう分岐予測ミスの影響を受けないという特長がある．このアルゴリズムではデータを複数系列の独立データとして取り扱うことで，VMX 命令の並列性を活かせるが最終的に各系列のマージ処理が必要となる．マージ処理は多数の条件分岐を含むため多くの処理時間を必要とする．そこで本アルゴリズムではさらに系列を越えるデータの比較・交換処理を追加することでこのマージ処理を条件分岐なしに行えるようにする．

本アルゴリズムを実装し，PowerPC 970FX プロセッサ上で評価を行った結果，クイックソートと比較して 32 ビット整数のソートで最大で 5.6 倍の性能向上が確認された．

ソートは多くのプログラムで必要とされる最も基本的な処理の 1 つであり，本研究で提案した汎用プロセッサの SIMD 命令に適した高速なソートアルゴリズムは HPC 用途をはじめ，用途を問わず多くのプログラムの性能向上に寄与するものと期待される．

参考文献

1) Knuth, D.E.: *The Art of Computer Programming, Vol 3: Sorting and Searching* (1973).
2) IBM Corp.: *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*.
3) Intel Corp.: *IA-32 Intel Architecture Software Developer's Manual*.
4) Lacey, S. and Box, R.: A Fast, Easy Sort, *Byte Magazine*, Vol.April, pp.315–320 (1991).
5) IBM Corp.: *IBM PowerPC 970FX RISC Microprocessor User's Manual*.
6) Pham, D., Asano, S., Bolliger, M., Day, M.N., Hofstee, H.P., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang,

M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T. and Yazawa, K.: The Design and Implementation of a First-Generation CELL Processor, *International Solid-State Circuits Conference*, IEEE, pp.184–185 (2005).
7) IBM Corp., Sony Computer Entertainment Inc. and Toshiba Corp.: *SPU Instruction Set Architecture*.
8) Purcell, T., Donner, C., Cammarano, M., Jensen, H. and Hanrahan, P.: Photon mapping on programmable graphics hardware, *Proc. SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware*, ACM, pp.41–50 (2003).
9) Govindaraju, N.K., Raghuvanshi, N. and Manocha, D.: Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors, *Proc. ACM SIGMOD international conference on Management of data*, ACM, pp.611–622 (2005).
10) Govindaraju, N.K., Raghuvanshi, N., Henson, M. and Manocha, D.: A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors, UNC Tech. Report (2005).
11) Freescale Semiconductor Inc.: *AltiVec Technology Programming Interface Manual*.

(平成 17 年 10 月 4 日受付)
(平成 18 年 1 月 20 日採録)

井上　拓（正会員）

1977 年生．2000 年慶應義塾大学理工学部システムデザイン工学科卒業．2002 年同大学院理工学研究科総合デザイン工学専攻修士課程修了．同年日本アイ・ビー・エム（株）に入社．現在同社東京基礎研究所に勤務．ハイパフォーマンス・コンピューティングシステム，プログラム最適化技術の研究に従事．

森山　孝男（正会員）

1962 年生．1985 年東京工業大学工学部情報工学科卒業．1987 年同大学院修士課程修了．同年日本アイ・ビー・エム（株）に入社．現在同社東京基礎研究所に勤務．並列マシンのオペレーティングシステム，グラフィックスの並列処理の研究に従事．

小松　秀昭（正会員）

1960 年生．1985 年早稲田大学大学院理工学研究科電気工学専攻修了．同年日本 IBM 東京基礎研究所入社．コンパイラ，アーキテクチャ，並列処理の研究に従事．博士（情報科学）．

中谷登志男（正会員）

1975 年早稲田大学理工学部数学科卒業．同年日本アイ・ビー・エム（株）入社．1985 年米国プリンストン大学大学院コンピュータ・サイエンス学科より M.S.E. および M.A., 1987 年 Ph.D. を各取得．同年より同社東京基礎研究所においてプログラム最適化技術の設計・実装・評価の研究に従事．2000 年よりディスティングイッシュト・エンジニア．現在同研究所システムズ担当．基礎研究部門におけるコンパイラ・ファームウエア技術のストラテジストを兼任．MICRO-22（1989）および MICRO-23（1990）にて，Best Paper Award を連続受賞．ACM，IEEE 各会員．