Vol.2012-HPC-135 No.12
2012/8/1

bitonic sortの高速な並列化

中澤 隆久[1,a)]　田浦 健次朗[1,b)]

概要：
昨今、並列性能の重要性が高まっているが、代表的なソートアルゴリズムであるクイックソートは逐次実行部分のクリティカルパスの長さのため、並列性能が高いとは言い難い。本研究では並列性能の高いソートの一つである bitonic sort を基盤として、その利点である並列性能の高さを維持しながら、実用においての欠点であるほぼソートされた列に対しての無駄な処理の削減を達成した鋸ソートを提案する。実験の結果、鋸ソートはランダム列に対しては bitonic sort と同等のスケーラビリティを持ち、ほぼソートされた列に対してはごく短い時間でのソートを実現した。

キーワード：ソートアルゴリズム, 並列化

High-speed parallelization of bitonic sort

Takahisa Nakazawa[1,a)]　Kenjiro Taura[1,b)]

Abstract: Recently,the importance of parallel performance has increased. However, quick sort which is a typical sorting algorithms does not have parallel performance much because of the length of the critical path of sequential execution. In this study, we propose 'Saw sort' which is made based on the bitonic sort which is one of measure sorting algorithms with high performance in parallelization. Saw sort maintain parallel performance of bitonic sort and improve the disadvantage of it in sorting array which is almost sorted. As a result of experiments, it turned out that Saw sort has scalability equivalent to bitonic sort in sorting randoom array, and can sort almost sorted array within quite short time.

Keywords: sort algorithm, parallelization

1. はじめに

1.1 背景と目的

ソートアルゴリズムは基本的なアルゴリズムの一つであり、様々な計算問題で利用するため、その速度は重要である。また、近年は単体の CPU の周波数性能の向上の限界から、マルチコア化による性能向上が盛んになってきていることから、逐次実行の速度だけでなく、並列性能の高いアルゴリズムが望まれる。クイックソート [1] は $O(n \log n)$ で実行される高速なソートアルゴリズムであるが、クリティカルパスが $O(n)$ であるため、マルチコアでの並列化を行った場合の並列性能はあまり期待出来ない。

並列性能を出しやすいソートアルゴリズムの一つとして、bitonic sort[2] が挙げられる。bitonic sort は逐次計算量は $O(n \log^2 n)$ と、クイックソートよりも多いが、バイトニック列のマージのクリティカルパスは $O(1)$ であるため、全体のクリティカルパスは $O(\log^2 n)$ と考えることが出来るソートアルゴリズムで、その並列性能の高さから新たな並列ソートとして改良される事が多い [3][4]。

また、bitonic sort はソートにおける操作が、要素の比較とスワップのみで行われるためソートの際に余分なメモリ領域を確保する必要が無い。他に並列性能が高いソートしては、マージソートを改良した Cilk sort や、バケットソート系の、sample sort などが挙げられるが、いずれも中間配列としてのメモリ領域が配列の長さに応じて必要となる。

[1] 東京大学大学院情報理工学系研究科
The University of Tokyo
[a)] tnakazawa@eidos.ic.i.u-tokyo.ac.jp
[b)] tau@eidos.ic.i.u-tokyo.ac.jp

並列実行が要求される対象は比較的大きいため、メモリ領域を要求する他の並列性能が高いソートと比べ有利な点と考えられる。しかし、bitonic sort は、そのアルゴリズムの性質上、ほぼソートされた列に対するソーティングが他のソートアルゴリズムと比べ大幅に遅くなってしまう。ソートアルゴリズムを実際に用いる場合、対象が完全なランダム列ではなく、ほぼソートされているという局面は多く、大きな欠点と考えられる。

本研究では、bitonic sort の持ち味と言える高い並列性能を維持しながら、その欠点を改善した、鋸ソートを提案する。鋸ソートは実験の結果、ランダム列に対しては bitonic sort 程度の並列性能を維持することを達成し、bitonic sort が苦手とするほぼソートされた列に対しては、逐次実行でクイックソート以上の速度を出すことに成功した。

1.2 本稿の構成

2 章では、本研究の関連研究として、提案手法の元となったソートアルゴリズムの他、主要な並列ソートアルゴリズムについて述べる.3 章では、提案手法である鋸ソートの概要と実装を述べる。4 章では、実験としてランダム配列やほぼソートされた配列のソートをコア数を変更しながら並列実行することによって、提案手法の性能を評価したものを報告する。5 章で本研究をまとめ、今後考えられる課題について述べる.

2. 関連研究

2.1 並列ソートに重要な要素

本節では、本研究が参考にした主要なソートや、並列性能の高いソートについて述べる。

実用上における並列ソートに重要な要素として、

- 逐次計算量
- クリティカルパス
- 余分なメモリ領域
- ほぼソートされた列に対しての強さ

が挙げられる。逐次計算量は実行時間に直接影響し、クリティカルパスは並列性能の限界に影響するためその重要さは言うまでもない。

また、並列性能が求められる様な大きな問題サイズを扱う場において、ソートの過程において必要となるメモリ領域のオーダーは無視出来ず、実際にソートアルゴリズムを使用する場で頻出する、ほぼソートされた列へのソートに対しての強さは実用上重要と考えられる。

2.2 クイックソート

クイックソートは逐次実行では一般的に最速とされるソートである。適当な数を Pivot とし、Pivot 以上の数値を前方に、Pivot 以下の数値を後方に移動させ、二分割されたデータに対し同様の操作を繰り返すことでソートを完了する。

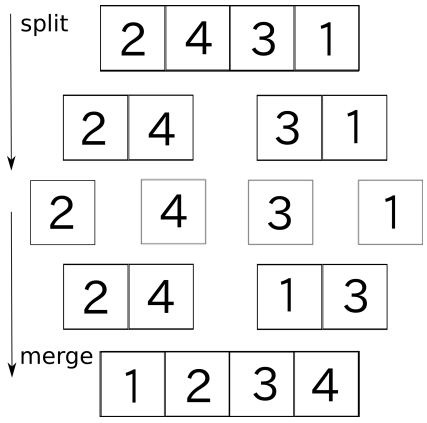

図 1 マージソート

平均計算量 $O(n \log n)$ と高速であるが、初回は並列実行が不可能なためクリティカルパスは $O(n)$ となるため、並列性能は高いとは言えない。

2.3 merge sort

bitonic sort と考え方の近い、マージソートについて記述する。マージソートはデータを再帰的に二分割していき、各分割ごとにソートを行い、その後ソート列をマージ (併合) することで、全体をソートするソートアルゴリズムである。マージの概要を図 1 に示す。

マージする二つの列は既にソート列であるため、それぞれの先頭を比較し、低い物から前に並べるという簡単なアルゴリズムでマージを達成することが出来る。

マージソートの計算量は $O(n \log n)$ であるが、n 個のデータをソートする場合を考えると、各分割はマージするまでそれぞれ独立して操作して良いため並列化出来るが、最後のマージは逐次で実行するため、クリティカルパスは $O(n)$ となるため、クイックソートと同様に並列性能はあまり良いとはいえない。

また、マージ部分の性質上、ソートするデータと同じだけのメモリ領域が中間配列として必要となる。

2.4 bitonic sort

bitonic sort もマージソートと同様に、データを再帰的に二分割していき、各分割ごとにソートを行うソートアルゴリズムである。bitonic sort では、マージ部分で bitonic 列を作成し、bitonic 列をマージしてソート列にするという手順をとる点が異なる。bitonic 列とは、図 2 に示すような「昇順列と降順列の接続」及び「要素をシフトすることでそのような列になる列」である。

bitonic sort では、ソートしたいデータの前半分を昇順にソートし、後半分を降順にソートすることで bitonic 列を生成する。

長さ 2^n のバイトニック列は、図 3 のように 2^{n-1} 間を

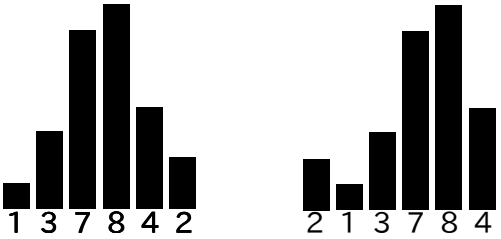

図 2 bitonic 列

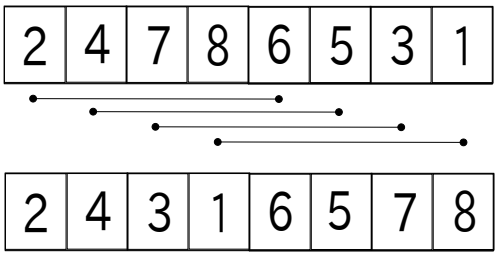

図 3 bitonic merge

比較し、昇順に並べ替えることで、前半部分と後半部分とで、長さ 2^{n-1} の二つのバイトニック列にすることが可能である。この交換を $2^{n-1} \sim 2^0$ までそれぞれのバイトニック列で繰り返し行うことで、ソートを完了することが可能である。

この作業における、それぞれのマージで行われる比較、交換はそれぞれ独立しているため、容易に並列化が可能である。

このため、逐次計算量は $O(n \log^2 n)$ とクイックソートやマージソートよりも大きくなるものの、クリティカルパスは $O(\log^2 n)$ となる。また、bitonic sort では全ての作業が要素の比較とスワップからなるため、マージソートのように temporary 配列としてのメモリ領域を確保する必要がない。一方で、bitonic 列を生成する関係上、ほぼソートされた列に対してはやや無駄が多く、通常は長さ 2^n のデータしかソート出来ないという欠点が存在する。

2.5 並列性能の高いソートアルゴリズム

前節で述べたように、bitonic sort は逐次計算量を犠牲に、マージソートの欠点であった「マージの並列性能」「メモリ領域の要求」を改善したソートアルゴリズムと言える。

並列性能を高めたソートアルゴリズムは他にもあり、以下にその簡単なアルゴリズムとその性能を記述していく。

2.5.1 Cilksort

Cilksort[5] はマージソート由来のソートアルゴリズムであり、単純にマージの並列性能を高める工夫が為されている。マージソートの並列性能の向上を阻害するクリティカルパス $O(n)$ となる、逐次実行で行っているマージを並列化する。

具体的には、マージする二つのソート列の片方の列の真ん中にあたる要素を Pivot として、もう片方の列を Pivot 以上、以下となる部分に分割しそれぞれの塊について要素数が一定以下になるまで、同様の操作を行う。分割されて生成された塊はそれぞれ独立しているため、容易に並列化してソートできる上に、マージ配列に入れる場所の一つ目のポインタが分かっているため、マージソートでは逐次で行っていた部分を並列実行することが出来る。

bitonic sort の様に、逐次計算量を犠牲にすることなく並列性能を高めることに成功しているが、中間配列としてのメモリ領域は必要となる。

2.5.2 バケットソート/基数ソート

バケットソートは、あらかじめ順番通り並べられた「バケツ」に対応するデータを入れていくことで、ソーティングを行うソートアルゴリズムである。

例として、1~1,000 の範囲の乱数が入った数列がある場合、バケツを 1,000 個用意しそれぞれを値 1~1,000 と対応づけ、数列の要素をバケツにより分ける。その後、1 のバケツから順番に入った値を並べるとソートが完了する。

バケットソートは逐次計算量が $O(n)$ であり、また並列化によって $O(\log n)$ にまで速度を高める事が出来る [6] 非常に高速なアルゴリズムであるが、使用にあたっては次のような制約がある。

- データの存在する範囲が有限個に限定される必要がある
- データの範囲 K 個だけバケツを用意する必要がある

前者は前提として当然の制約であるが、後者はバケットソートの明確な欠点となり得る。仮に数列長が 100 であっても、値の範囲 K が 1~1,000,000 である場合、バケットソートを行うには 1,000,000 のバケツを用意する必要がある。この様にメモリ領域の使用が大きくなる場合があり、K が大きすぎる場合確保しきれないといった問題が起きてしまう。

バケットソートにあった問題を解決したソートアルゴリズムが、基数ソートである。要素の範囲が k 桁である時、バケットソートを k 回することにより、全ての要素をソートする。m 進数であれば、バケツを m 個用意すればソートを行うことが出来るため、バケットソートが抱えていた問題は無く、計算量もバケットソートの高々 k 倍で済むためオーダーも変化せず、高速なアルゴリズムである。

ただし、バケットソートを k 回行う際の、二回目以降のバケットソートは値の小さなバケツの要素から順番に処理する必要があるため並列化が難しく、バケットソートと比較すると並列化は難しく、性能も低くなる [7][8]。また、性質上ある程度のメモリ領域が必要であることは避けられない。

2.5.3 Sample sort

サンプルソート [9] は、遊ぶコアを減らすことで並列性能を高めたソートである。要素数が n の配列をコア数 p でソートすることを考える場合、コア 1 つに対し n/p 個程度

の要素のソートがタスクとして与えられるのが望ましい。

サンプルソートでは p 個のバケット $B1 \cdots Bp$ に均等に要素を割り振るために、まず配列を $A1, A2 \cdots Ap$ に分割し、それぞれでソートを行う。その後、各 Ai からなるべく一つのコアに均等な数が割り当てられるような範囲を計算しそれぞれの Bi に設定、データを放り込む。

その後、各 Bi をソートすることで、ソーティングを完了する。それぞれのソートは逐次で行うため、クイックソートなどの逐次の早さに特化したものを利用可能なのが強みである。全体の逐次計算量は $O(p^2 + n \log n + p \log p)$ となり、クリティカルパスは $O((n/p) \log n + p \log p)$ となる。

サンプルソートは並列性能が高く、クリティカルパスも短く、またクイックソートを逐次ソートで使用する場合ほぼソートされた列にも比較的強くなる。しかし、p が非常に大きくなった場合のクリティカルパスが大きくなる他、バケットを用いるために、そのためのメモリ領域は必要となる。

3. 提案手法

3.1 鋸ソート

本節では本研究が提案する、「鋸ソート」アルゴリズムの説明を行う。鋸ソートは bitonic sort を基盤とし、短所であったほぼソートされた列に対しての弱さを大幅に改善したものである。

2.1 節で触れた、並列性能に重要となる要素をまとめた物が、表 1 となる。鋸ソートが、クリティカルパスの低さと余分なメモリ領域の少なさを兼ね備えることが分かるだろう。

以下に鋸ソートの擬似コードを示す。

鋸ソート

```
void saw sort(int start, int length){
  if(length > 1){
    int m = length/2;
    saw sort(0,m);
    saw sort(m,length);
    saw merge(0,length);  }
}
```

簡単のため、マージを行う関数 saw merge の中身は後述する。引数の start はソートの開始位置であり、length はソート列の長さを意味する。基盤となる bitonic sort と同じように、データを再帰的に二分割していき、各分割ごとにソートを行い、その後ソート列をマージ (併合) することで、全体をソートを行う。実際のプログラムではコア数の上限などから、n>1 まで再帰的に saw sort 関数を呼び出す必要性は薄いため、データが設定した閾値以下の長さには逐次でソートを行う様にした。

3.2 鋸マージ

次に、アルゴリズムのマージ部分である、鋸マージの説明をする。bitonic sort では bitonic 列を再帰的に作り出すという形でマージを行っていたが、bitonic 列の形状上、ほぼソートされた列に対しては bitonic 列の生成においてほぼ昇順のデータを降順にソートするという無駄な操作が顕著であった。

鋸マージでは、鋸列という並びを再帰的に作り出すという形でマージを行う。鋸列の定義を以下に示す。

鋸列

- 昇順ソート列の連続である。
- 昇順ソートの塊が高々 2 つである。
- シフト等を行わないで上記の条件を満たす。

bitonic 列は「昇順→降順」という並びであったが、これを「昇順→昇順」の鋸列を採用することで、ほぼソートされた列が対象であっても、無駄に崩すことなく、鋸列の構築を可能としている。

3.2.1 再帰的なマージに必要な要素

本節では、鋸マージの理解のために、鋸列が「高々 2 つのソート列の連続」である必要性を記述する。鋸マージは、bitonic merge と同様に、長さ 2^n の鋸列を長さ 2^{n-1} の二つの鋸列にする作業を再帰的に行ってソートを完了するアルゴリズムである。

このような再帰的なアルゴリズムに必要な要素として、作業の終了時に

- データの中間値より小さなデータが前半分に、中間値以上のデータは後ろ半分に移動している
- 前半分と後半分が最初の鋸列と同じ性質を持つ

事が挙げられ、bitonic merge はこれを満たしている。

この事を鋸列に当てはめると、作業終了時に前半分に存在して欲しい要素の候補は、二つ目のソート列の小さな部分であり、それらの要素と交換すべき要素の候補は一つ目のソート列の大きな部分である。ソート列が高々二つならば、これらを比較すればいいが、ソート列が三つ以上になると、これらの候補が単一では無くなるため、同様の操作で要件を満たせなくなってしまう。

そのため、鋸列はソート列二つ以下の連続で収める必要があり、要素の移動を行った上で、鋸マージにはそのための操作が要求されることになるが、この要件を満たすことで要素の交換だけでの再帰的なソートが可能となり、余分なメモリ領域の確保が不要となる。追加操作があるとはいえ、オーダーに影響を与える程度ではないため、無理なく要求されるメモリ領域を削減することが可能である。

3.2.2 実際に行った手法

以下に鋸マージの擬似コードを示す。簡単のため、二つの並んだソート列の一つ目の大きさが全体の半分よりも大

	クイックソート	マージソート	bitonic sort	cilk sort	bucket sort	sample sort	鋸ソート
逐次計算量	$O(n \log n)$	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log n)$	$O(K+n)$	$O(p^2 + n \log n + p \log p)$	$O(n \log^2 n)$
クリティカルパス	$O(n)$	$O(n)$	$O(\log^2 n)$	$O(\log^2 n)$	$O(\log(K+n))$	$O((n/p) \log n + p \log p)$	$O(\log^2 n)$
余分なメモリ領域	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(K)$	$O(n)$	$O(1)$
ソート列への強さ	○	△	×	△	×	○	◎

表 1 主要なソートの並列性能

きい場合についてのみのコードを記述した。

鋸マージ

```
void saw merge(int start, int length,int flag){
 if(length>1){
 int m = length/2;
 int swapt = 0; //swap すべき回数
//図 4 左上→右上//中間値以下の数値を前に、以上を
後ろに送りつつ、その個数を測定
   for(int i = 0; i < length-flag; ++i){
    if (items[start+m-1-i] > items[start+flag+i]){
      swap(items[start+m-1-i],items[start+flag+i]);
      swapt++;
    }
   }
//後半分を鋸列にするための準備
    for(int i = 0; i < (flag-m)/2; ++i){
    swap(items[start+m+i],items[start+flag-1-i]);
    }
//図 4 右上→右下:前半分を鋸列に
    for(int i = 0; i < swapt/2; ++i){
  swap(items[start+m-swapt+i],items[start+m-1-i]);
    }
//図 4 右下→左下:後半分を鋸列に
    for(int i = 0; i < (flag-m+swapt)/2; ++i){
     swap(items[start+m+i],items[start+flag+swapt-
1-i]);
    }
    int left = m-swapt;
    int right = flag+swapt-m;
   saw merge(0,n,left);
   saw merge(m,n,right);    }
}
```

更に説明の補助として、図 4 に擬似コードで行っていることを可視化した。図 4 左上が鋸マージを行う前、右下が鋸マージを行った後最終的に作られる配列の形状である。左半分に全体の中央値よりも小さい要素が行き、右半分に中央値よりも大きい要素が行った上で、両側に鋸列が生成されていることが確認出来る。

引数の flag によって、2 つのソート列の切れ目を与える

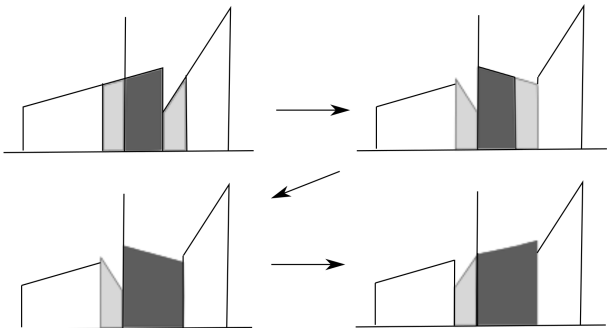

図 4 鋸マージ

ことで、比較すべき要素の位置を探索なしに決定することが可能である。更に、対象列のソートが既に終わっている場合、切れ目の位置が両端であるという形で表現されるため、余分な判定操作を入れる事無しにソート列に対する操作を省くことができる。また、簡単のために今回は逐次実行のコードを記述しているが、図の左上から右上に移行する際の二つの作業及び、右上から左下、左下から右下に移行する二つの作業はそれぞれ独立であるため、容易に並列化が可能である。更に、それぞれの for 文の中で行われる比較と交換は bitonic sort と同様に全て独立しており、これも並列実行が可能であるため、クリティカルパスは bitonic sort と同じになる。

4. 評価実験

実装は $c++$ を使って行った。また、並列化を行う際のライブラリには、*massivethreads*[10] を用いて実装を行った。実験環境は Intel(R) Xeon(R) CPU E7540 @ 2.00GHz を物理 24 コア持つマシンで行った。実験は提案手法である鋸ソートの他、比較対象として、並列化を行ったクイックソートと、提案手法の元となった bitonic sort の測定を行った。

実験では、int 型の長さ 2^{27} の配列を使用した。長さを 2 の階乗としているのは、bitonic sort の純粋な性能を測定するためである。bitonic sort は 2^n 以外の長さの配列をソートする場合、中身が 0 の空要素を加えて長さを 2^n として行うため、実際の性能を測るためにも配列の長さは 2 の階乗とした。実験として、ランダム列のソーティングと、既にソートされた列に外れ値を一つ混ぜた、ほぼソートされた列のソーティングを使用するコア数を変更しながらそれぞれのアルゴリズムで行い、3 回の実行の平均時間を測定した。

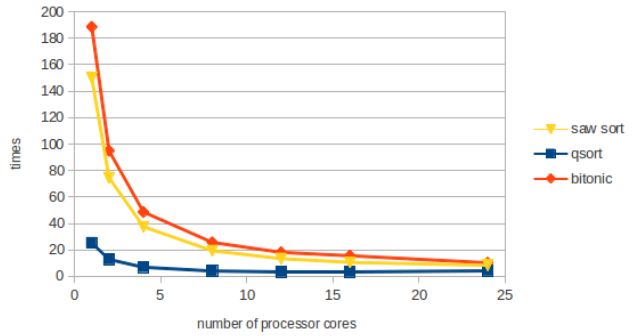

図 5 ランダム列に対する実行時間

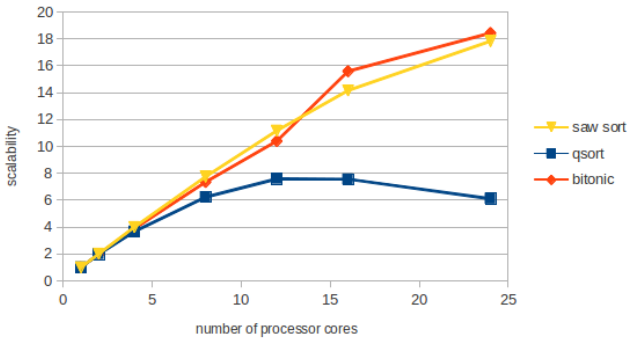

図 6 ランダム列に対する実行時間のスケーラビリティ

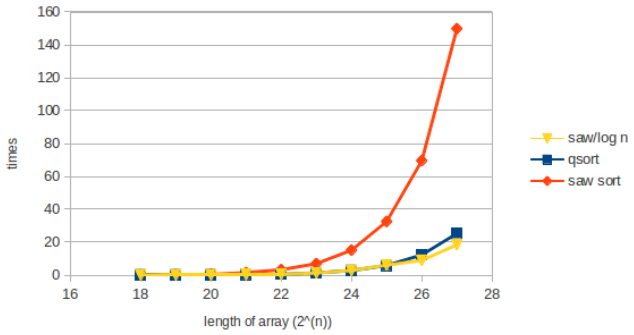

図 7 計算量の比較

ランダム列に対する実行時間を、図 5 に、実行時間のスケーラビリティを図 6 に示す。

図 6 から、bitonic sort はクイックソートよりも高い並列性能を持つことが確認出来、提案手法である鋸ソートは元となった bitonic sort とほぼ同等のスケーラビリティを持つことがわかる。

並列性能の高い bitonic sort は一見すると高いスケーラビリティを持つように見えるが、24 コアまでのスケーラビリティの伸び具合を見ると、bitonic sort と鋸ソートの両方で、コア数の増加に伴いスケーラビリティが低下しており理想的なスケールとは言い難い。これは、ボトルネックとなる、並列性能が出せずに少数及び単独のコアで作業を行っている部分の影響がだんだん大きくなったためという原因や、コア数の増加による実行時間の短縮から、タスク

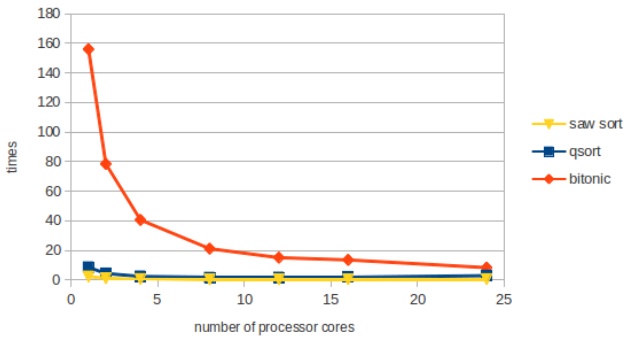

図 8 ほぼソートされた列に対する実行時間

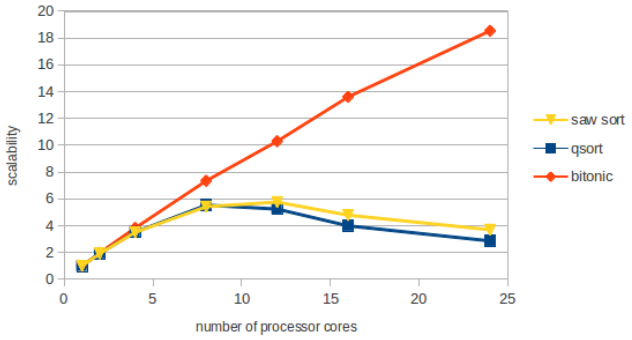

図 9 ほぼソートされた列に対する実行時間のスケーラビリティ

生成などの作業によるオーバーヘッドの影響が大きくなってきたためと考えることが出来る。前者の要因についてだが、本実験では元となった bitonic sort も同程度の性能であることから、鋸ソート独自の実装部分ではなく、両者共通の部分にそのような要因が存在し、改善の余地があると考えるのが妥当といえる。

また、図 5 から、実行時間では逐次でクイックソートに大きく劣り、並列化を行っても実験を行ったコア数内ではクイックソートよりも遅いという結果が得られている。この原因としては、逐次の全体計算量の影響があげられる。クイックソートは $O(n \log n)$ であり、bitonic sort 及び鋸ソートの計算量は $O(n \log^2 n)$ であるため、配列の長さを大きくした今回はその影響が顕著になったと言える。ソートを行うランダム配列の長さを変更しながら逐次での実行を行い、鋸ソートの実行時間を $\log n$ で割った実行時間を加えて比較したものが図 7 である。グラフから、鋸ソート/$\log n$ の実行時間はクイックソートの其れとほぼ一致しており、逐次コアでの時間差は、計算量オーダー由来の物と考えられる。

次に、ほぼソートされた列に対してのソーティングの実行時間を図 8 に、実行時間のスケーラビリティを図 9 に示す。

グラフから、bitonic sort の実行時間が明らかに長くなることが分かる。前節までに述べて来たように、bitonic sort の bitonic 列を生成するというアルゴリズムが、ソート列

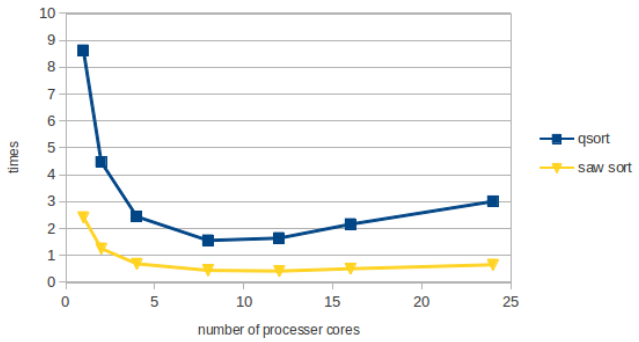

図 10 ほぼソートされた列に対する実行時間

に対しては無駄が大きい事が原因になったと言える。一方で鋸ソートとクイックソートはソート列に対して比較的強いため、実行時間はランダム列をソートした場合よりも大きく短くなる。

図 8 から bitonic sort を取り除いて拡大を行った物が、図 10 である。鋸ソートは逐次実行においても、クイックソートを凌駕する速度を持つことが分かる。

図 9 からは、bitonic sort に比べ、鋸ソートの並列性能は出ない事が読み取れるが、この原因は実行時間の短さのためと考えられる。4 並列の時点で既に実行時間 1 秒を下回る事から、並列化の為のオーバーヘッドの影響がスケーラビリティに強い影響を与える為になったと推測される。

5. おわりに

5.1 まとめ

本研究では、並列性能の高いソートの一つである bitonic sort を改良し、その並列性能を維持しながら、同ソートが苦手とする「ほぼソートされた列」に対するソーティングの速度の改善を実現する鋸ソートを提案した。鋸ソートは bitonic ソートが用いる bitonic 列がソートされた列との相性が劣悪な事から、相性が比較的良い二つのソート列の連続と定義した鋸列を提案し、これを再帰的に生成することでソートを実行するといった実装を行った。

実験では鋸ソートはランダム列に対しては bitonic sort と同等以上の性能を発揮し、ほぼソートされた列に対しては、逐次実行の実行時間をもソート列に強く逐次実行が早いクイックソートを上回る、という結果を残すことが出来た。

5.2 今後の課題

今回の実験結果から、以下のような課題が考えられる。

- 速度の改善
  実験で出た結果は、逐次実行、並列性能の両方で十分と言えるものでは無かった。そのため逐次実行時の計算量をできる限り減らして行くことが考えられ、そのために実装の最適化を行ってなるべく無駄をなくしていく事が肝要である。。並列性能は、特にほぼソートされた列に対してのスケーラビリティの改善が求められるところである。並列性能を低めていた原因をより詳しく追求し、あまり並列出来ていない部分であるボトルネックの解消を行うことで改善を測りたい所である。
- より正確な評価
  実験環境では、物理 24 コアのマシンを用いたが、実際には更に多くのコア数を用いたときに、コア数に応じたスケーラビリティを出すことが並列プログラミングとしては求められるところである。実装を改善すると共に、さらに多くのコア数を用いた評価を行いたい。また、今回比較対象として提案手法の元となった bitonic sort と、逐次実行の速度が高いクイックソートを用いたが、高速な並列プログラミングとを目指す目的がある以上、関連研究で挙げた様々な高速並列ソート手法との比較を行って行きたい。

参考文献

[1] Hoare, C.A.R.: Quicksort. Computer Journal 5(4), 10-15 1962.
[2] K. E. Batcher.: ”Sorting networks and their applications,” in Proc. AFIPS SJCC, vol. 32, Montvale, NJ: AFIPS Press, pp.307-314.1968.
[3] Inoue, H., Moriyama, T., Komatsu, H., Nakatani, T.: AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In: Proc. Int.l Conf. Parallel Architectures and Compilation Techniques (PACT 2007), pp. 189-198 2007.
[4] Bugra Gedik, Rajesh R. Bordawekar, Philip S. Yu.: CellSort: high performance sorting on the cell processor. VLDB Endowment: Proceedings of the 33rd international conference on Very large data bases.September 2007.
[5] C. Leiserson and A. Plaat.: Programming parallel applications in Cilk. SINEWS: SIAM News, 31, 1998.
[6] B.S. Chelebus.: A parallel bucket sort. Info. Process. Lett..27(2):57-61, Feburary 1988.
[7] S. J. Lee, M. Jeon, D. Kim, and A. Sohn.: Partitioned Parallel Radix Sort, J. Parallel Distr.Comput. (JPDC), 62:656 668 2002.
[8] A. Sohn and Y. Kodama.: Load balanced parallel radix sort, in ‘‘ Proc. 12th ACM International Conference on Supercomputing, Melbourne, Australia, July 14-17, 1998.
[9] Cheng, D.R., Edelman, A., Gilbert, J.R., Shah, V.: A novel parallel sorting algo-rithm for contemporary architectures. Submitted to ALENEX 2006.
[10] 中島潤，田浦健次朗．高効率な I/O と軽量性を両立させるマルチスレッド処理系．情報処理学会論文誌 プログラミング (PRO)．Vol.4，No.1，pp.13-26．2011.