Vol.2013-HPC-140 No.2
2013/7/31

省メモリでスケーラブルなマージソートアルゴリズム

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えない. 一方で並列性能の高いソートアルゴリズムである Cilk sort や sampling sort は一時メモリがソート対象の大きさ分必要となり, 大規模なデータのソートにおいてはその範囲に制約がかかってしまう. 本研究では並列性能が高く省メモリである bitonic sort を基盤として, その利点を維持しながら, 実用においての欠点である逐次計算量とほぼソートされた列に対しての無駄な処理の削減を目指す鋸ソートを提案する. 実験の結果, 鋸ソートは対象となるデータがキャッシュから大きくはみ出さない範囲においてはその全てを達成した.

キーワード：ソートアルゴリズム, 並列化

Memory efficient and scalable merge sort argolithm

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. On the other hand, in sorting of large-scale data, as for Cilk sort and Sampling which are sorting algorithms with high parallel performance, a lot of temporary memories are needed. 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 and memorry efficient. Saw sorting aims at reduction of the processings to the serial computational complexity and the almost sorted array which were the faults of bitonic sort, attains in the processing to the data of the size about cache.

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. 近年, データベースなどの巨大化から非常に大きな要素数のソートを行うことも増えてきている一方で, 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を期待することが出来る, 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] は平均 $O(n \log n)$ 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高々 $\log n$ 倍と, 台数効果があまり期待出来ないアルゴリズムと言える.

他に並列性能が高いソートアルゴリズムとしては, 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,radix sort などが挙げられるが, 要素の比較と交換のみで完結するクイックソートと異なり, 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 $O(n)$ 必要となる. 多くの一時領域を必要とするソートアルゴリズムは, 一度にソート可能な量が一時領域をあまり必要としないアル

[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

ゴリズムと比較して小さくなり, 並列実行が要求される対象は比較的大きいためその制約にかかる可能性が高くなる. また, 多くの一時領域を用いることは, キャッシュミスを誘発し易くするため, 速度にもいい影響は与えないと考えられる.

クイックソートのように要素の比較と交換によって完結し, 並列性能が高いソートアルゴリズムとして,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大きいが, バイトニック列同士のマージ操作のクリティカルパスは $O(1)$ であるため, ソート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].

並列性能の指標となるクリティカルパスの低さと省メモリの両立が行えるという点で bitonic sort は優秀であるが, 実際にはプロセス数 p がソート対象の大きさ n に対して遥かに少ない状況が多いため, 逐次計算量の多さが実行速度に大きく影響し, 実行速度があまり出ないという欠点がある. また, 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い. 例えば粒子の運動を観測するための粒子法などでは, 逐次時間ごとに微小に変化していく粒子の情報のソートを行うため, ほぼソートされた列を修正するといったソートを繰り返し行うことになる. このように実用上, ほぼソートされた列に対して弱いというのは大きな欠点と考えられる.

本研究では,bitonic sort の利点である高い並列性能と省メモリを維持しながら, 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 cilk sort やクイックソート並の逐次速度と bitonic sort 程度の並列性能を達成し,bitonic sort が苦手とするほぼソートされた列に対しての強さも, 大きく改善することに成功した.

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.

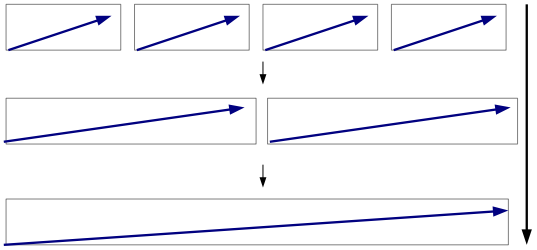

図 1 マージソート

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.

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 の源流である, 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, その後ソート列を繰り返しマージ (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, 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.

2.3.1 バケットソート/radix 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 [8] 非常に高速なアルゴリズムであるが, 使用にあたってはデータの範囲 K 個だけバケツを用意する必要があるため, 仮に数列長がごく短い物であっても, 値の範囲 K が莫大であれば, それだけのバケツを用意する必要がある. この様な場合メモリ領域が確保しきれないといった問題が往々に起きてしまう. この問題を解決したソートアルゴリズムが,radix 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, 計算量もバケットソートの高々 k 倍で済むため, 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 [9][10].

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ず, 扱う値にばらつきがあるとメモリ領域の使用量と実行時間が増加してしまう傾向がある.

2.4 並列性能の高いソートアルゴリズム

以下に並列度の高いソートアルゴリズムについてその簡単なアルゴリズムと性能について記述する. 以下に紹介する sample sort 系とマージソートの改良系に大別される. 最近の研究では, 前者は CloudRAMSort[7], 後者は odd-evenmerge[5] などが挙げられる.

2.5 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.

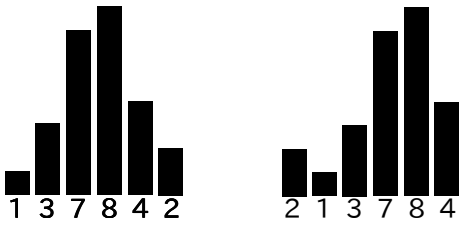

図 **2** bitonic 列

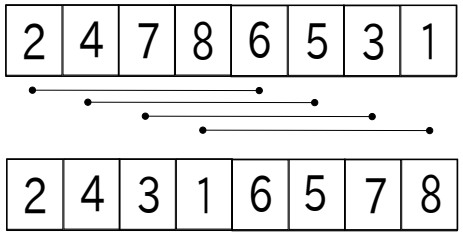

図 **3** bitonic merge

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.

2.5.1 Cilksort

Cilksort[6] 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 のと比較すると, 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 Sample sort

サンプルソート [11] 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. 分配が終わった後はバケット間で通信しないため分散環境にも適しており, これをベースとした CloudRAMsort[7] などは高い性能をあげている.

しかし,p がオーダーに入ってくる関係上, 並列度が高くなった場合にクリティカルパスが悪くなる. これは高並列度でのパフォーマンスが求められるアルゴリズムとしては無視出来ない欠点と言える. また, 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. 大規模な量のソートを行う際に余分なメモリ領域を必要としない事は代え難い利点であるが, 前節で説明した並列ソートの中で, それを満たす物は bitonic sort のみである.

鋸ソートは bitonic sort を基盤とし, 長所である省メモリ性と高い並列性能を損なうことなく, 短所であった逐次計算量の多さとほぼソートされた列に対しての弱さを改善したものである.

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, 通常のマージソートに近い動きとなり, 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. 一方で bitonic merge をベースとしたアルゴリズムの採用により, 余分なメモリ領域が不要となっている. また,, ソート列同士と対象とすることで, 既にソートが完了している場合, その列の以下のマージ操作の一切を省略することが出来るため, 計算量の削減も行えている.

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.

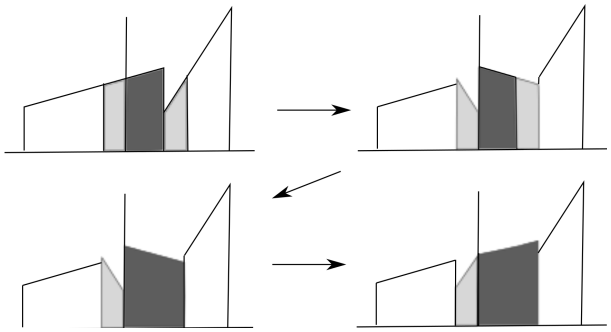

図 4　鋸マージ

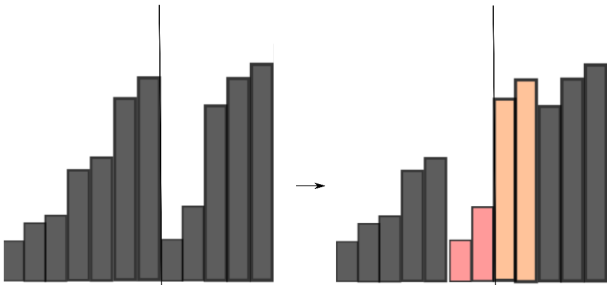

図 5　簡易鋸マージ

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る.

3.2.2　鋸マージ

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きい場合についてのみのコードを記述した. 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. 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.

実際に行った手法

```
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);     }
}
```

3.3　簡易鋸マージ

鋸マージは要素の反転を利用して, 鋸列を二つのの鋸列に分けた. この操作一連の計算量自体は $O(n)$ だが, 反転を行うために逐次計算量は 2 倍近くとなってしまう. クリティカルパスは変化しないものの, プロセス数 p が配列長

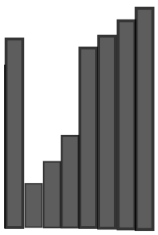

図 6　両側に偏りがあるケース

n と比較して大きく少ない現状, この計算量の増加は無視出来るものではない.

そこで, 基本的には図 5 で表現される簡易鋸マージを採用する. こちらは境目で二つに割り, 交換を行う. これにより, 一度の探索と一度の交換だけで再帰的に二つの鋸列を構築することが可能であり, 計算量を抑えることが可能である.

しかし, 問題点も存在する. 図 6 の様に二つのソート列の大きさが大きく偏った状態に陥ると, 最終的にソート完了するまでに N 回の再帰呼び出しが行われてしまうことになる. こうなると, 関数再帰呼び出しや二分探索のオーバーヘッドが大きく影響し始める. このため, 山の切れ目が全体長からみてある割合以上, 両端に寄った場合, 先述の鋸マージを行うという実装にした.

鋸マージならば, 図 6 のようなはずれ値も高々 $\log n$ 回の再帰呼び出しで正しい位置に移動することが出来る.

3.4　末端の処理

bitonic merge では基本的に末端まで先述のマージ処理を繰り返す. しかし, このマージ処理はクリティカルパスこそ $O(\log n)$ であるが, 逐次計算量は $O(n \log n)$ であり, 並列度が充足しきっている末端において, この操作を続けることの意味はあまり無いと言える. また, 関数の再帰呼び出しによるオーバーヘッドも無視出来ない.

そのため,「長さ $n/2$ の二つのソート列のマージ」は長さ n の一時領域さえあれば, 原始的なマージソートによって $O(n)$ の操作で終了することが出来るため, ある定数 m を下回った範囲のマージは, 予めプロセスに確保していた大きさ m の領域を使って原始的なマージを行う事とした.

m には全体の長さ n に対してごく小さい値を設定することで, メモリ使用量に影響を殆ど与えること無く, 高速化が可能である.

また,bitonic sort はマージソートベースであるため通常は最初に長さ 1 になるまで分割を行うが, タスク生成や再帰呼び出しのオーバーヘッドを考慮して十分なタスク数が得られている末端では, 一定の長さを下回ったらクイックソートを行った.

3.5　計算量

ここで, 実際の計算量を評価する. 元々の bitonic sort の逐次計算量は $O(n(\log n)^2)$ であるが, 鋸ソートでは長さ m 以下では並列性能を度外視した高速な逐次アルゴリズムを用いる事により, 元々 $O(n \log n)$ であったマージ部分の計算量を $O(n(\log n - \log m) + m)$ にまで削減することが可能である.

一方で, 空間計算量としてプロセス数 p に応じて $p * m$ だけの追加の領域が必要となる.

m を大きくすることで, 全体の逐次計算量が $O(n \log n)$ に近づいていく一方で, 必要な一時領域が $O(n)$ に近くなるトレードオフが存在することが分かる. m は任意に決定出来るため, 実際には調整次第で, 多少の一時領域を用いるだけでも大きく高速化が可能である.

3.6　各アルゴリズムとの性能比較

2.1 節で触れた, 並列性能に重要となる要素を主要なソートに対してまとめた物が, 表 1 となる. 鋸ソートはクリティカルパスの低さと空間計算量の少なさを兼ね備えた上で, ある程度の逐次計算量の削減とソート列への強さを達成していることが分かる.

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[12] 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, 代表的な高速並列ソートである Cilk sort, そして提案手法の元となった bitonic sort の測定を行った. まず,int 型の長さ 2^{24} の配列を使用し, ランダム列のソーティングと, 既にソートされた列のそれぞれの値に幅 100 の一様乱数を加えた, 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と使用メモリ量を測定した. また, 逐次マージへの切り替えは 2^{15} からとした. ランダム列に対する性能を, 図 7 に, メモリ使用量を図 8 に, ほぼソートされた列への性能を図 9 に示す. なおグラフでは, 提案手法である鋸ソートの逐次実行時の性能を 1 とし, それに対してのスケーラビリティを表示する.

図 7 から, 元となった bitonic sort はクイックソートよりも高い並列性能を持つが, 絶対性能では大きく劣ることが確認出来る. これは主に逐次計算量の差から来る物と考えられる.

一方で提案手法の鋸ソートは逐次実行時間は Cilk sort, クイックソートにやや劣るものの, ほぼ遜色のない速度が出ており, 並列性能も Cilk sort と同等以上の値が出ていることが分かる. また, 図 8 から, 空間計算量もスワップベー

	quick sort	merge sort	bitonic sort	cilk sort	sample sort	saw sort
逐次計算量	$O(n \log n)$	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log n)$	$O(p^2 + n \log n + p \log p)$	$O(\log n(n(\log n - \log m) + m)))$
クリティカルパス	$O(n)$	$O(n)$	$O(\log^2 n)$	$O(\log^2 n)$	$O((n/p) \log n + p \log p)$	$O(\log^2 n)$
余分なメモリ領域	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(\log n + m * p)$
ソート列への強さ	○	△	×	△	○	○

表 1　主要なソートの並列性能

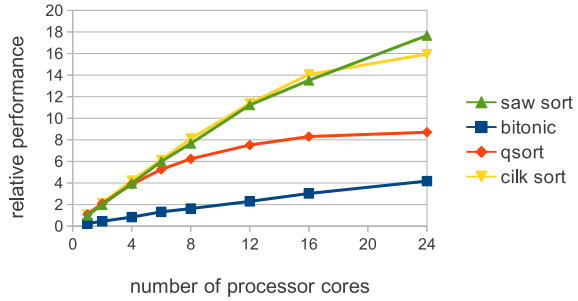

図 7　長さ 2^{24} のランダム列に対する性能

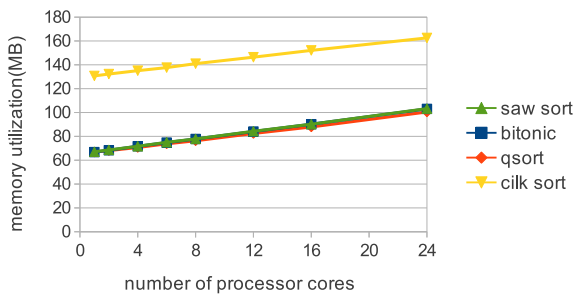

図 8　長さ 2^{24} の配列におけるメモリ使用量

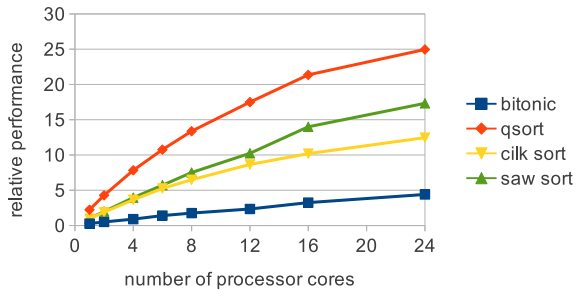

図 9　ほぼソートされた列に対しての性能

スである bitonic sort と quick sort の物からほぼ変化すること無く, 速度の向上が達成出来ていることが確認出来る. 一方で一般的なマージソートがベースとなっている Cilk sort は中間配列が必要なためメモリ消費量が倍近くになっている. なお, 全てのグラフにおいてコア数の上昇と共にメモリ使用量が増えているのは, 使用した並列ライブラリ MassiveThreads が 1 プロセスごとに内部で確保しているメモリの総計が増加するためである.

次に, ほぼソートされた列に対しては, 図 9 から, ソート

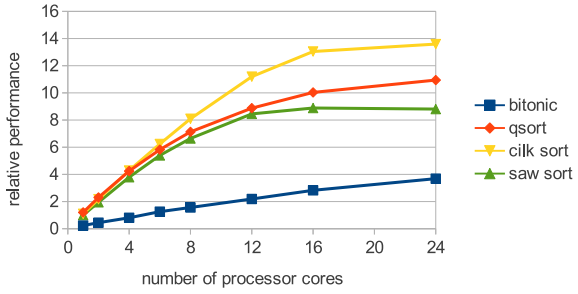

図 10　長さ 2^{28} のランダム列に対する性能

列に強いクイックソートが逐次実行で提案手法の 2 倍ほどの性能になることが分かる. しかし, スケーラビリティに関しては, 提案手法はランダム列に対する場合とほぼ同等の性能を持ち,Cilk sort やクイックソートを上回る結果となった. また, 今回扱った粒子法などを想定した「元々のソート列の値が一定の範囲で変化した列」に対しては, 提案手法はこの程度の性能であるが,「ソート列に対して多少のはずれ値が入った列」に対してはクイックソート以上に適しており, 逐次実行においてもクイックソートを上回る結果を残している.

次に配列長を 2^{28} にし, より大きなデータでの実験を行った. 結果を図 10 に示す. 元となった bitonic sort に比べると早いものの, 配列長 2^{24} の時と比較して, 提案手法の性能が落ちてしまったことが分かる.

この原因としては, キャッシュミスによる遅延が考えられる. 今回実験に使ったマシンの L3 キャッシュの大きさは 18MB*4 であり, 先の実験ではほとんどのデータがキャッシュに収まりあまりキャッシュミスを起こすこと無くソートが行えた. しかし, 今回の実験ではメモリ使用量は 1GB 以上と, マシンの L3 キャッシュの容量を大きく超えるため, キャッシュミスの発生は避けられない. これは提案手法である鋸ソートに限らず, クイックソートや cilksort もその影響を受けているが, 特に鋸ソートは 1 ステップごとに飛び飛びのデータ交換を行うため, 現状の実装では, キャッシュミスを他のソートと比較して大きく発生してしまうことが, 今回の遅延の一つの原因として考えられる.

そこで検証のため, キャッシュミスの発生を抑えるよう, 序盤は交換部分のみ並列化を行い, 分割される列をタスクとして生成せずある程度大きさがキャッシュ付近になったところでタスク生成を開始し処理を並列化する処理にした

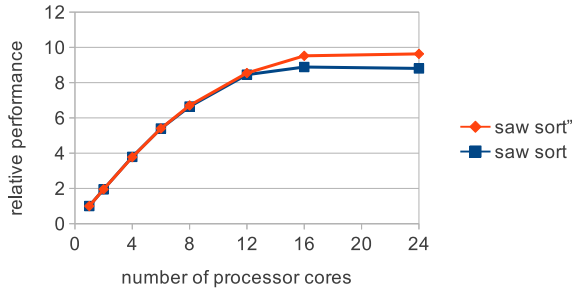

図 11　キャッシュミスの影響の比較評価

鋸ソート”での比較実験を行った.

結果を図 11 に示す. 鋸ソート”はキャッシュミスの削減を目指すという点ではかなりナイーブな改良であるが, 元の鋸ソートと比較すると, コア数を増やした場合において, スケーラビリティの改善が見られることから, 多くのコアを用いる場合, キャッシュミスを減らすようなタスクの分配を行うことがより重要であると考えられる.

5. おわりに

5.1 まとめ

本研究では, 並列性能が高く余分のメモリ領域の少ない実用的なソートアルゴリズムを実現するため, 並列性能の高く省メモリなソートの一つである bitonic sort を改良し, その長所を維持しながら, 同ソートの大きな欠点である「逐次計算量の多さ」「ほぼソートされた列に対する弱さ」の改善を行った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形がかけ離れていることに着目し, 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. 更に, 逐次計算量を減らすために, 普段は簡易的な処理でマージを行い, また, 末端での処理を改善した.

実験では鋸ソートはメモリ消費量を大きく増やす事なく, ランダム列に対してクイックソートや高速並列ソートとして知られる Cilk sort と同等以上の性能を実現し, ほぼソートされた列への弱さも解消される, という結果を残した.

一方で, キャッシュを大きくはみ出す大きさの対象に対しては, キャッシュミスの影響などから, スケーラビリティが低下した. 省メモリであることはより大きなデータを扱う場合に効力を発揮するということもあり, この点は大きな改善点と言えるだろう.

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.

- より大きなデータ対象に対する速度の改善
  今回の実験では, キャッシュを大きくオーバーするデータに対してはあまりいい結果を残せなかった. この問題を解消するためには, 不用意にキャッシュミスを発生させない必要がある. そのためより大きなデータを対象にした場合でもスムーズに動くためのアルゴリズムの手順が求められると言える. また, 連続したデータを扱うシーンが多く, その部分に対する SIMD の適用で操作自体の速度を上げていくことも重要であると言える.
- より広い範囲での評価
  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ところであり. 大規模なデータの並列ソートということで, 分散環境への適用もいずれは考えて行きたい.

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] Jatin Chhugani, Anthony D. Nguyen. : Efficient implementation of sorting on multi-core SIMD CPU architecture Proceedings of the VLDB Endowment VLDB Volume 1 Issue 2, August 2008 Pages 1313-1324. 2008.
[6] C. Leiserson and A. Plaat.: Programming parallel applications in Cilk. SINEWS: SIAM News, 31, 1998.
[7] Changkyu Kim, Jongsoo Park. : CloudRAMSort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster. SIGMOD ’12 Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. Pages 841-850. 2012.
[8] B.S. Chelebus.: A parallel bucket sort. Info. Process. Lett..27(2):57-61, Feburary 1988.
[9] S. J. Lee, M. Jeon, D. Kim, and A. Sohn.: Partitioned Parallel Radix Sort, J. Parallel Distr.Comput. (JPDC), 62:656 668 2002.
[10] A. Sohn and Y. Kodama.: Load balanced parallel radix sort, in ‘‘Proc. 12th ACM International Conference on Supercomputing, Melbourne, Australia, July 14-17, 1998.
[11] Cheng, D.R., Edelman, A., Gilbert, J.R., Shah, V.: A novel parallel sorting algo-rithm for contemporary architectures. Submitted to ALENEX 2006.
[12] 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.