修士論文

# タスク並列処理に適した省メモリなマージソートアルゴリズム

Memory efficient merge sort algorithm for task parallel processing

平成 26 年 2 月 6 日提出

指導教員　田浦 健次朗 准教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-126429 中澤 隆久

## 概 要

配列を昇順及び降順に並び替える整列アルゴリズムは, 様々な計算問題で利用するため, その実行速度は重要である. 近年, データの巨大化から, 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, 並列ソートアルゴリズムの需要が高まっている. 代表的な並列ソートアルゴリズムは空間計算量 $O(n)$ である Cilk sort, Sample sort 及びそれらを基盤とし改良を施した物が多いが, 一時メモリがソート対象の大きさ分必要となってしまうため, 大規模なデータのソートにおいては一度にソート可能な範囲に制約がかかってしまう. また, 並列ソートアルゴリズムの研究においては速度を追求するため,Pthreads や MPI といった非常に primitive なプログラミングモデルを用いるものが殆どである. このような実装は利用可能なコア数の変化や, 入力配列のばらつきの偏りに対応し辛く, 実際のプログラムでパフォーマンスが常に出せるとは言いがたい. 更に, コードの記述が逐次実行と比較して複雑な上実験環境が変化した場合にコードの書き換えが必要であるなど, プログラマに高度な知識が求められるといった問題がある. そのため, 高生産に並列計算を行うことが可能な並列プログラミングモデルの需要が高まっており, その一つであるタスク並列モデルは, 『タスク』という独立実行可能な新たな処理単位を導入する事で多くのソートアルゴリズムが行う再帰を用いる分割統治法も直感的かつ効率的に並列化することができる. そしてワークスチーリングを用いた動的負荷分散が可能であることから, 利用可能なコア数の変化にも大きな影響を受けずに実行が可能である. ただし, ソートアルゴリズムを代表する広範囲のデータを読み込む問題に対して単純なワークスチーリングを用いた場合, 各 CPU が扱う箇所が分散するため共有キャッシュミスなどの影響で実行速度が低下するといった問題もある.

本研究ではこのような背景からタスク並列処理に適し, 省メモリかつほぼソートされた列に適性を持つ実用的なソートアルゴリズム, 鋸ソートと, タスク並列モデルを用いて分割統治法の問題を解く際に有効な共有キャッシュを考慮したワークスチーリング方針を提案する. 鋸ソートは並列性能が高く省メモリな bitonic sort を基盤とし, その利点を維持しながら欠点である逐次計算量の多さとソート列に対しての無駄な処理を削減を達成した物である. 実験の結果, 鋸ソートは省メモリ性を持ちながら高速並列ソート Cilk sort に劣らない並列性能を持ち, ソート列に強い quick sort 並のソート列への適性を持つことを確認した. また, キャッシュを大きく超えるサイズの配列に対しては, 通常のスチーリング方針では速度が遅くなってしまうものの, 提案スチーリング方針の導入によってキャッシュミスを 20%削減し, 改善することに成功した.

# 目 次

# 図 目 次

# 表 目 次

# 第1章　序論

## 1.1　ソートアルゴリズム

ソートアルゴリズムは様々な値を持ったデータ配列を昇順及び降順に並び替える基本的なアルゴリズムである. 単純な並び替え目的以外にも二分探索を行う際の前処理など, 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・メニーコアによる性能向上を期待することが出来る並列アルゴリズムの需要が高まっている. そのため, 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. quick sort[7] 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$\log n$ 倍で頭打ちとなる. そのため, 台数効果があまり期待出来ないアルゴリズムと言える.

並列性能が高いソートアルゴリズムとしては, merge sort を改良し, クリティカルパスを $O(n)$ から $O(\log^2 n)$ に短縮した Cilk sort や, bucket sort を改良した, sample sort, radix sort などが挙げられるが, 要素の比較と交換のみでソートを行う quick sort と異なり, いずれのソートも中間配列としてのメモリ領域が配列の長さ $n$ に応じて $O(n)$ 必要となる. このような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ない.

quick sort のように要素の比較と交換によってソートを行い, さらに並列性能も高いソートアルゴリズムとして, bitonic sort[8] が挙げられる.bitonic sort は逐次計算量は $O(n \log^2 n)$ と, quick sort よりも大きくなるが, bitonic 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 [9][13].

並列性能の指標となるクリティカルパスの低さと, 中間配列を必要としない省メモリの両立が行えるという長所を bitonic sort は持つが, 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. そのような場合, ランダム列に対するソートよりも実行時間が短縮されることが望まれるが, bitonic sort の場合はアルゴリズムの性質上ランダム列に対する実行時間とほぼ同じだけの実行時間となってしまう. 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になると考えられる.

ここまで挙げた要素である,

- 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

はどれも重要であり, これらの性質を併せ持った並列ソートアルゴリズムが望まれる.

## 1.2　タスク並列処理系

近年, 並列計算機はマルチコア化や NUMA アーキテクチャによるメモリの不均一化や階層化, ハードウェアアクセラレーションの導入などにより, 計算環境の複雑化が顕著である. 現在, 標準的に使われている並列プログラミングモデルとして, 共有メモリ環境においては OpenMP[5], 分散環境においては MPI[11] が挙げられる.

これらのプログラミングモデルは, 実際に行われる計算や通信を把握・考慮したプログラミングを要求するため, 繊細なチューニングを行う事が出来る利点がある一方で, プログラマ側がロック等の対処も全て行わなければならず, 逐次プログラミングからの飛躍が大きいという欠点がある. また, これらのモデルではデータを静的に分割する並列化モデルである SPMD モデルを用いた並列化を行うことが多いが, SPMD モデルはデータの量と計算量が比例するという前提に基づいて行う物であるため, 入力によってデータ全体の負荷が一様とはならない場合並列度が低下してしまうといった問題があり, ソートアルゴリズムにおいてもこの問題は存在する. このようなプログラミングモデルによる記述の難解さは, 近年の計算環境の複雑化・多様化からこれからさらに大きくなると考えられる.

このような背景から, より高水準・高生産に並列プログラムを記述するためハードウェアをより高度に抽象化し, 平易な記述で高い実行性能を達成することを可能としたプログラミングモデルの研究が近年盛んである. 代表的な処理系として, IntelTBB[4], Chapel[2], Cilk[1], X10[3] などが挙げられ, これらの言語では並列実行可能な処理を細粒度で分割した『タスク』という処理単位を導入し, それらをシステムがハードウェアの CPU コアに割り振って並列処理を実現するタスク並列処理が導入されている.

タスク並列処理によって, 再帰や多重ループといった SPMD モデルでは効率的な並列化が難しいアプリケーションも平易に記述することが可能となる一方で, スケジューラーは処理中に生成される大量のタスクを各 CPU に効率的に割り振ることが重要となる.

現在広く行われているスケジューリング手法として, Lazy Task Creation[6] に基づいたタスク生成方針を採用しランダムワークスチーリングによる負荷分散を行うものが挙げられる. この手法は, タスクの負荷分散を問題なく行えるが, 今後複雑化する大規模並列計算機上でより高い実行性能を出すためには, より工夫したタスクスケジューリングを行う必要性があるといえる.

## 1.3　本研究の目的と貢献

以上の背景の下, 本研究は省メモリで実行可能かつ, クリティカルパスが $O((\log n)^2)$ となりソート列に強い, タスク並列モデルを用いた並列ソートアルゴリズムである鋸ソートを提案する.

提案手法では, 予備配列としての余分なメモリを $O(n)$ で必要とせず, ユーザーが定義する定数 $p$ を用いて $O(\log n + p)$ で表されるメモリ消費でのソートを実現し, Cilk sort 並の逐次実行速度と並列性能を得ることに成功した. また, ほぼソートされた列に対しては他の並列ソートと比較して逐次実行時に比べての実行時間の短縮幅が大きくなることを確認した. これにより, 提案手法は粒子法などのソート列への微小変化データに対するソートを行う問題において, 有用なソートであると考えられる.

また, このような扱うデータサイズが広範囲かつ大きな問題をタスク並列による分割統治法で行う場合に生じる, コア間の共有キャッシュミスの大量発生を, 共有キャッシュのサイズを考慮するヒントを与えることで削減するワークスチーリング方針を提案し, 鋸ソートで実装・評価を行いその効果と実用性を示した.

## 1.4　本稿の構成

本稿の構成は以下のようになっている. 2 章では代表的な並列ソートアルゴリズムと重要な性質について触れ, 3 章ではタスク並列処理について述べる. 4 章では関連研究を紹介する. 提案手法として, 5 章では鋸ソートアルゴリズムについての説明を行い, 6 章では共有キャッシュを考慮したワークスチーリング方針についてそれぞれ述べる. 7 章で提案手法の評価を行い, 8 章で結論を述べる.

# 第2章　並列ソートアルゴリズム

本節では, 並列ソートアルゴリズムにおいて重要視される要素と, 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アルゴリズムについて述べる.

## 2.1　並列ソートに重要な要素

現在, 様々なソートアルゴリズムが存在しているが, 1.1 節で述べたように, 並列ソートアルゴリズムとして実用的な運用を行うために, 満たさなければならない要素が存在する.

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. 以下でそれぞれについて簡単に述べる.

**逐次計算量**　ソート対象の配列長 $n$ に対する計算量の総計のオーダーであり, 単一コアでの実行時間に直接影響する要素である. 総計算量は並列実行においても, 実行時間に比例する要素であるため, その重要さは言うまでもない.

**クリティカルパス**　並列化が可能な部分を完全に並列化した, すなわち台数が無限にある場合に最適な並列実行を行なった場合のアルゴリズムの計算量であり, 並列性能の限界を定義する要素である. ソートに限らず, 並列アルゴリズムにおいて重要な要素であり逐次計算量との差が $n$ 倍未満である場合, 台数効果が $\frac{\text{逐次計算量}}{\text{クリティカルパス}}$ で頭打ちになる.

**余分なメモリ領域**　merge sort 等, 一部のソートアルゴリズムではソートを行う際に, 中間配列としてのメモリ領域が対象配列とは別に $O(n)$ 分必要となる. この中間配列のための一時領域も含めた配列全体がマシンのメモリに乗らなければ, ディスクアクセスが頻繁に発生するため, 実行時間は大幅に増加してしまう. すなわち, 多くの一時領域を必要とする in-place でない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ゴリズムと比較して小さくなり, 並列実行が要求されるようなソート対象は配列全体のサイズも大きくなることが多いため, その制約にかかる可能性が高くなる.

また, メインメモリに乗り切るサイズの問題であっても, 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, 実行速度にも悪影響を及ぼす可能性があると考えられる.

**ほぼソートされた列に対しての強さ**　実アプリケーションとしてのソートを考える場合, ソート対象の配列は必ずしもランダム列ではなく, ほぼソートされた列であ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, タイムステップごとに微小に変化していく粒子の情報に対してソートを行うため, ほぼソートされた列をソート列に修正し直すといったソートを繰り返し行うことになる. このようなほぼソートされた列に対しては, ランダム列に対するソートよりも実行時間が短縮されることが望まれるが, アルゴリズムの中にはランダム列に対する実行時間とほぼ変わらない時間がかかる物も存在する.

粒子法などの大規模並列計算においてソートアルゴリズムを運用する場合, このようなソート列への強さは実行時間に大きく影響するため, 重要な要素と考えられる.

他に, 並列ソートに限らずソートアルゴリズムとして考慮すべき要素として安定性がある. ソート対象の中の同等な値を持つ要素の順序が, ソート前後において常に保存されるものを安定ソート (stable sort) といい, データベースなどで複数の値を持つデータ配列に対して, その内の一つに注目してソートを行う場合などに安定性が必要となる場面が存在する.

安定でないソートでこのような安定性を保つためには整列したいデータに元のデータ列の順序を追加しておき, ソートする際にその情報を参照するようにする必要があり, これによって安定ソートに変更することは可能である. しかし, $O(n)$ の外部領域が必要となる他, 参照による遅延も無視出来ないため安定ソートであることは, ソートアルゴリズムとしての汎用性を大きく高めるといえる.

## 2.2　逐次計算量が少ないソート

前節で述べたように, 実行時間に直接影響する逐次計算量は少ないに越したことはない. 本節では逐次計算量が少ないソートを挙げる.

### 2.2.1　quick sort

quick sort は分割統治法の一種であり, 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る. これにより配列の前方に中央値よりも小さな要素が全て移動する. 以降は, 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.

データ配列の並びに計算量が影響されるアルゴリズムであり, 最悪の場合計算 $O(n^2)$ となることが知られているが, Pivot をランダムで選出した 3 つの値の中央値をとるなどの工夫を行うことでこれを回避すること可能である. 一方でほぼソートされた列には強く大幅に計算量を削減出来る他, 平均計算量 $O(n \log n)$ とランダム列に対しても高速なアルゴリズムである.

分割統治法であるため, 並列化も容易であるが, 初回は並列実行が不可能であり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.

また, 要素の比較と交換で完結する swap ベースのソートアルゴリズムであるため必要な余分なメモリ領域が $O(\log n)$ となる in-place アルゴリズムであるが, 安定ソートではないため, 安定性を保証する場合 out-of-place となる.

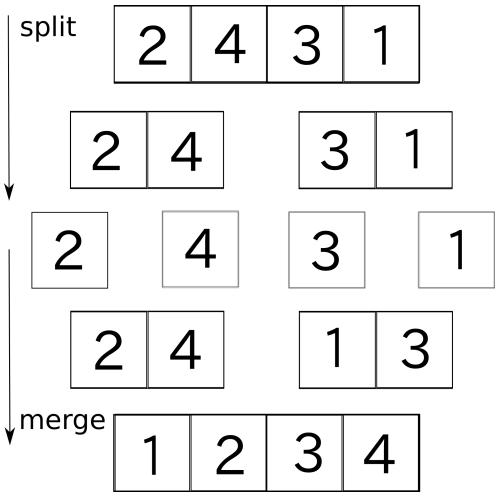


図 2.1. merge sort

### 2.2.2　merge sort

merge sort 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, 最終的にデータ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 2.1 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事を繰り返す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.

最悪計算量がナイーブな quick sort よりも少なく, $O(n \log n)$ となることが知られており, 安定した速度を持つアルゴリズムであり, さらに安定ソートである. 一方で, ほぼソートされた列に対しては大幅に計算量を削減することは出来ない.

$n$ 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簡単に並列化出来るが, 原始的な merge sort では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 $O(n)$ となるので, quick sort 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, 基本的に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ため通常 $O(n)$ の外部領域が必要となる.

### 2.2.3　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.

多くのソートが $O(n \log n)$ かかる中, 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 [19] 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

この二つの制約から, 実際にバケットソートを使用出来るケースは非常に限定的な物となる. とくに後者の制約が厳しく, 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, $K$ が大きすぎる場合メモリが確保しきれないといった問題が起きてしまうためソートが行えなくなる.

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, radix ソートである. 要素の範囲が $k$ 桁である時, 桁ごとのソートを桁数分の $k$ 回することにより, 全ての要素をソートする. この際使用するソートアルゴリズムによって計算量が変化するため基本的にはバケットソートが採用される事が多い. $m$ 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, 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安定性が逐次実行時でしか保持されないバケットソートを採用する場合並列化が難しく, 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 [20][21].

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, バケットソート系の性質上, 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.

## 2.3　並列性能の高いソートアルゴリズム

前節で軽く触れたが, 逐次性能の高いソートアルゴリズムは並列性能についてあまり考慮されているとは言えず, そのまま並列ソートとして用いることは難しい. 本節では並列性能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. 近年研究されている高速並列ソートは, 以下に紹介する sample sort 系と merge sort の改良系に大別される.

### 2.3.1　bitonic sort

bitonic sort は merge sort 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 列をマージして最終的に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 2.2 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 bitonic 列を生成する.

長さ $2^n$ のバイトニック列は, 図 2.3 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

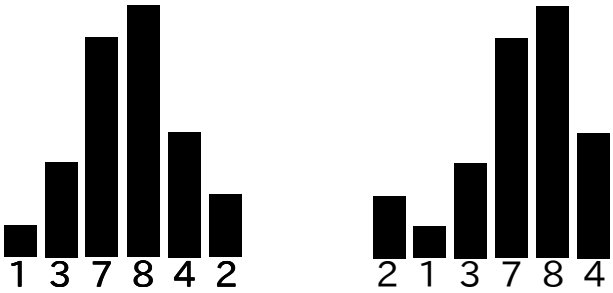


図 2.2. bitonic array

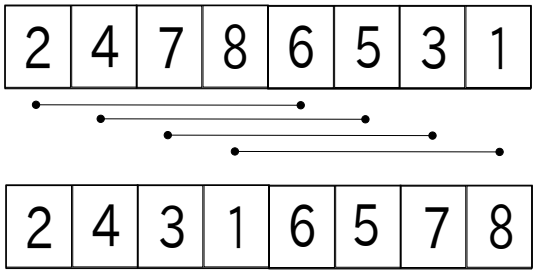


図 2.3. bitonic merge

$2^{n-1} \sim 2^0$ 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.

bitonic sort はこのマージにおける比較の順序が, それまでの比較の結果によらずあらかじめ決まっておりそれぞれ独立であるソーティングネットワークの部類であり, 容易に並列化が可能である. このため, 逐次計算量は $O(n \log^2 n)$ と元の merge sort 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交換からなるため, merge sort のように中間配列としてのメモリ領域を確保する必要がなく, 安定ソートでもある. 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も一旦崩してしまうため無駄が多いという欠点が存在する.

### 2.3.2　Cilksort

Cilksort[15] は merge sort 由来のソートアルゴリズムであり, 個々のマージの並列性能を高める工夫が為されている. merge sort の並列性能の向上を阻害しクリティカルパス $O(n)$ となる原因を作っている, 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, merge sort 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.

### 2.3.3 Sample sort

サンプルソート [22] は, なるべく計算をコア毎に均一に分割することで, 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 $Bi$ に設定し, データを割り振る.

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, quick sort 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.

データの分割が上手く行われた場合, サンプルソートは高い並列性能を出す. また, 一般的には quick sort を分割後の各コアで行うため, ほぼソートされた列に対しても強くなる. 分配が終わった後はバケット間で通信が行われないため分散環境にも適しているといえる.

しかし, 均一にデータを分割するための下準備としての sampling 作業のため, p がオーダーに入ってくる関係上, 使用コア数が多くなった場合に全体の逐次計算量とクリティカルパスに影響を及ぼしてしまうこれは, 大規模並列計算機を用いた高並列度でのパフォーマンスが求められる並列ソート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, そのための中間配列としてのメモリ領域は $O(n)$ 必要となる.

### 2.3.4 parallel radixsort

逐次ソートとして非常に高速な radix sort の並列化を行なったものが parallel radix sort である.

radix sort のアルゴリズム上, そのままでは並列化が行えないため, まずデータをコア数分に分割し, それらをまずコア毎に独立に radix sort を行う. その後, 各コア毎に生成されたヒストグラムを元にして, コア全体で要素の再分割を行い, ソートを完了する.

元の逐次計算量が少ない radix sort をベースとするため非常に高速な並列ソートであるが, 並列度はそこまで高いとは言えず, 入力データのばらつきによって実行速度が左右されやすいという欠点がある. またビット列を参照するため long 型や double 型のデータをソートする場合 int 型のデータに対するソートよりも時間がかかるという欠点がある. 入力により実行時間が左右され易いため, 他の並列ソートと比較して radix sort 向きの問題を解くことに特化している並列ソートであり, 汎用性はやや低い. 尚, radix sort ベースであるため中間配列としてのメモリ領域 $O(n)$ が必要となる.

## 2.4 ソートアルゴリズムの性能比較

近年の並列ソートアルゴリズムは, 細部の違いはあるものの, 基本的にはここまでに挙げたソートのいずれかを基盤としている.

本節では 2.1 節で挙げた並列ソートに重要な要素について, これらのソートアルゴリズムの性能比較を行ったものを表 2.1 に示す.

表より, 全ての要素において優秀な値を持つソートは存在しないことが分かる. 逐次計算量の少ないソートはいずれもメモリ領域かクリティカルパスで問題があり, この二つの要素に関して優秀な値を持つ bitonic sort は他の要素に問題を抱えている事が分かる.

並列ソートアルゴリズムの研究はこれらのソートアルゴリズムのどれかをベースにすることが多いが, $O(n)$ 使用している一時領域のオーダーを $O(\log n)$ に落としこむ事や, 逐次でしか行えないアルゴリズムの並列化を行うためにはアルゴリズムの抜本的な変換が求められるものが多い. そこで, 本研究では bitonic sort をベースアルゴリズムとして扱い, メモリ領域とクリティカルパスの利点を保ちつつ, 逐次計算量とソート列への強さに対しての欠点を改善することで, 新たな並列ソートアルゴリズムを提案する.

表 2.1. 主要なソートの並列性能

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

# 第3章　タスク並列処理系

## 3.1　タスク並列モデルとソートアルゴリズム

1.2 節で述べたように, タスク並列処理系では, 並列実行可能な処理を細粒度で分割した「タスク」という処理単位を導入し, それらをシステムがハードウェアの CPU コアに割り振って並列処理を実現する. タスクは OS レベルのスレッドと比較して生成及び管理のコストが非常に軽量であるため, 大量にタスクを生成することで高い並列度を出すことが可能である. 生成されたタスクの負荷分散などは処理系が自動的に行うため, プログラマはタスクの生成と終了同期のみを考慮すればよく, データを静的に分割し, コアに割り当てる SPMD モデルの並列化と比較して直感的にプログラミングを行うことが出来, 再帰や多重ループ等を含む場合もプログラム構造を変更しなくて良いため, 逐次プログラムからの飛躍が少ないという長所がある. また MPI などの primitive な並列化を行う並列プログラミングモデルなどと比較すると, チューニングの繊細さという点ではタスク並列モデルは劣ってしまうが, 一方で動的負荷分散が出来ることから, 利用可能な CPU コア数が変化しうるような状況に強いことから, 実用的な実装であると言える. タスク並列で記述された高速な並列ソートアルゴリズムは, 大きな変更なしに様々なアプリケーションに適用できる汎用性の高いプログラムといえるため, これからの今後複雑化する大規模並列計算機上でより高い実行性能を出すために, タスク並列に適した並列ソートアルゴリズムが求められると言える.

## 3.2　分割統治法

分割統治法は, 大きな問題を再帰的に小さな問題に分割していき, 最終的に問題を解決する手法であり, merge sort や quick sort などに代表される様に, ソートアルゴリズムでは比較的よく用いられる手法である. 分割統治法は再帰呼び出しを用いて記述する事が多いため, MPI などを用いて SPMD モデルの並列プログラミングを行うことがやや難しい. タスク並列モデルを用いることで, 各関数で行われる再帰呼び出しをそれぞれタスクとして考えることが出来る. ナイーブな quick sort の実装を図 3.1 に示す. アルゴリズム中の spawn と sync はそれぞれタスク並列モデルにおける, タスクの生成と生成されたタスク群の終了同期のための命令であり, 逐次実行のプログラムとの差分は再帰呼び出し部分でこれらの命令を用いる部分のみである. このように非常に平易に並列化を行うことが出来ることから, タスク並列モデルは分割統治法を記述する際に相性が良いと言える.

```
1  void q_sort(int *items, int n)// 引数はそれぞれ配列と配列長
2  {
3    int pivot, left, right;
4    if(n > 1)
5    {
6      // 配列の中央の要素を分割値
7      pivot = items[n/2];
8      left = 0;
9      right = n-1;
10     // 分割値以上と未満の二つの配列に分割する
11     while(1)
12     {
13       // 左から分割値以上の要素を探す
14       while(items[left] < pivot)left++;
15       // 右から分割値未満の要素を探す
16       while(items[right] > pivot)right--;
17       // 交差していたら分割を終える
18       if(left >= right)break;
19       // 見つかった二つの要素を交換する
20       swap(items[left], items[right]);
21       // 比較する場所を一つ進める
22       left++;
23       right--;
24     }
25     // 分割した二つの配列をそれぞれソートする
26     spawn q_sort(items, left);
27     spawn q_sort(items+left, n-left);
28     sync;
29   }
30   return;
31 }
```

図 3.1. タスク並列モデルを用いた quick sort の記述

## 3.3　一般的なタスクスケジューリング手法

タスク並列処理では基本的に実行 CPU 数分のスレッドを生成し, タスクの実行を行う. タスクはそれぞれのスレッドが持つローカルなタスク・キューに格納され各スレッドはそれを順番に処理していく. 効率の良い実行性能を得るためにはこれらタスクの効率の良いスケジューリングが不可欠である. 本節ではタスク並列処理における一般的なスケジューリング手法の説明を行う.

### 3.3.1　Work-first スケジューリングとワークスチーリング

タスクはスレッドと比較して遥かに生成管理のコストが小さいとはいえ, 不要に大量のタスクを生成することは望ましくない. そこで一般的には, 親タスクが子タスクを生成した場合, 親タスクを中断してキューに格納し, 子タスクの方をスレッドが実行する Work-first スケジューリングが行われている. 実行開始時にタスクを持っていないスレッドや, ローカルキューに処理すべき

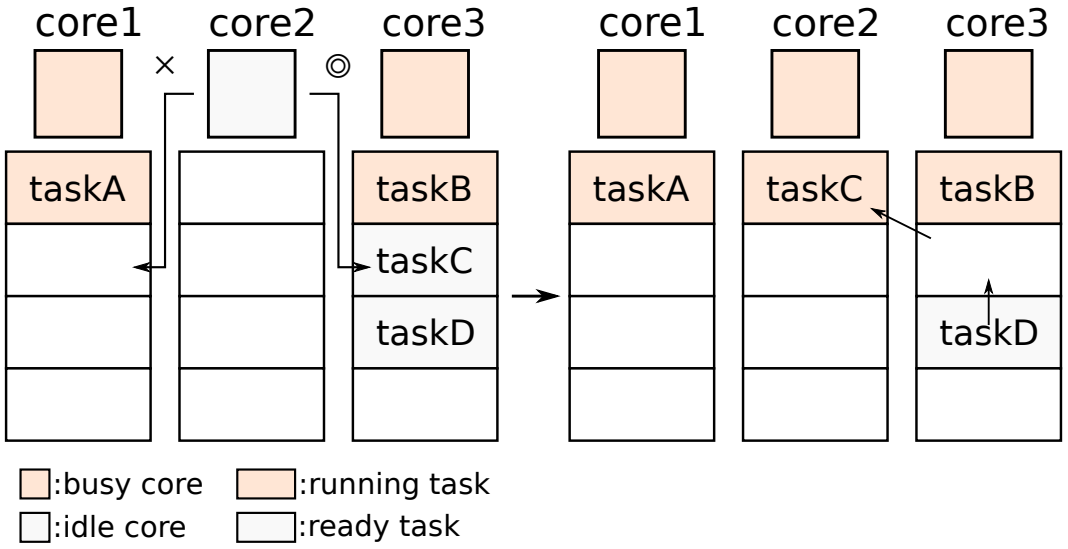


図 3.2. work stealing

タスクが無くなったスレッドはアイドル状態になってしまう. このままでは負荷分散が取れないため, アイドル状態となったスレッドは他のスレッドのキューから実行可能なタスクを奪い, それを実行するワークスチーリングを行う [10]. ワークスチーリングの例を図 3.2 に示す. 図 3.2 ではアイドル状態となったコア 2 が他のコアのキューを見に行き, スチール可能なタスクを持つコア 3 のキューからタスクをスチールすることで, 負荷分散を行っている.

また, スチーリングの際に大抵の場合盗まれる側のワーカーのキューには実行可能なタスクが複数存在するが, 分割統治法のプログラムのように, タスクは再帰的に生成される場合が多いため, 盗まれる側のワーカーが最も古く作ったタスクを盗むのが, 効率が良いとされている. 古くに作られたタスクはそれだけタスクグラフの根に近い部分のためより多くのタスク生成が期待出来るほか, 全体の処理量も多く次のワークスチーリングの発生までの時間が比較的長くなり負荷分散が上手く行くためである.

### 3.3.2　タスクスケジューリングにおける課題

タスク並列モデルは平易に記述が出来, 動的負荷分散も行えるが, 多くのタスクを生成し, それを各スレッドが処理していく過程に置いて個々のタスクがメモリにあるデータを読み込む場合, キャッシュミスの問題が無視出来なくなることが指摘されている [23].

キャッシュは, メモリよりも小容量である代わりに高速にアクセスが可能であり, 時間のかかるメモリへのアクセス回数を減らすことでデータ転送の冗長化を低減させる目的で置かれる記憶階層である. 現在のマシンにおいては共有キャッシュとプライベートキャッシュを組み合わせた, 多段構造のキャッシュが用いられることが多い. CPU は最も近くのキャッシュにまずアクセスを行うが, この際キャッシュに必要なデータが存在しない場合にはキャッシュミスが発生し, より深い階層のキャッシュにデータを探しに行く.

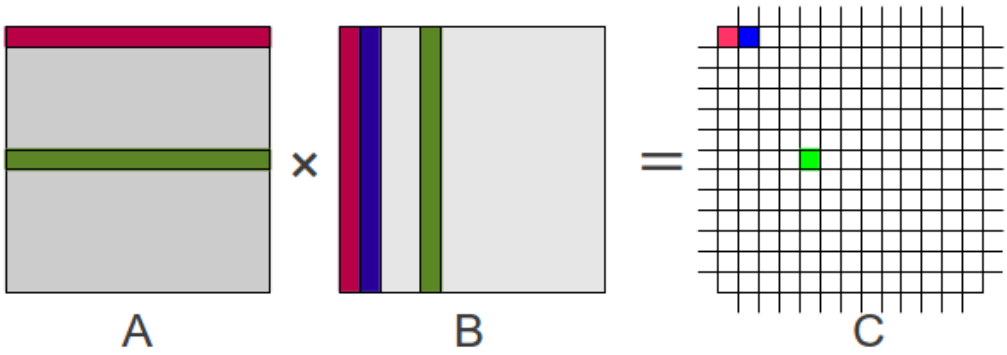


図 3.3. 行列積計算において読み込むデータ範囲

キャッシュミスの際のアクセスにかかるコストは主記憶に近いほど高くなり, その遅延は無視出来るものではない. そのため, スレッドがデータに触る際のキャッシュミス率を削減することがプログラムの実行速度に大きく関わってくると言える.

しかし, タスク並列処理を普通に実装した場合, このような問題を解決出来るとは言いがたい.

図 3.3 は密行列 $A \cdot B = C$ の演算をタスク並列処理で行う例である. 簡単のため, 計算すべき領域 C を分割し, その領域の計算をそれぞれタスクとする場合の例について述べる. この場合赤の部分を計算するのに必要な A の領域と青の部分の計算に必要な A の領域は同じであるが, 緑の部分を計算する場合は全く違ったデータが必要になる.

本来キャッシュを有効に利用するため, 赤の部分を計算するタスクと青の部分を計算するタスクは同じコア及び同じ共有キャッシュを持つコア群で行われるべきである. しかし何も考えずにタスクの生成とワークスチーリングが行われれば, 当然処理していくタスクが計算する箇所はスレッドごとに飛び飛びになるため, キャッシュミスが大量に発生してしまい, 大幅な遅延が生まれてしまう.

このような問題はプログラマがタスク生成のタイミングをプログラム内で調整する等の処置である程度は改善が可能とされているものの, スケジューラ側からキャッシュミスを積極的に増やさないためのスケジューリングも必要である.

特に負荷分散のためのワークスチーリングは, 通常はワークスチールを行う対象をランダムで決定し, 盗めるタスクがあればかならず盗むという greedy な方針が採択されることが多いが, 結果的にスチール回数やキャッシュミスを不要に増大させ実行時間が逆に伸びてしまう事があるため, 改善の余地があると言える.

# 第4章　関連研究

本章では関連研究について述べる．4.1 章では, 近年研究されている高速並列ソートについて挙げ, 4.2 章ではタスク並列処理系における, 高速化のための改良スチーリング方針に関連した研究を挙げる．

## 4.1　高速並列ソート

ソートアルゴリズムには様々な物があるが, ソート自体が比較的単純な問題であるため前に述べた様に近年のソートアルゴリズムの研究では基本的なアルゴリズムは 2 章で挙げたもののいずれかであり, 抜本的に異なるアルゴリズムの考案はあまりされていない. 既存の高速ソートアルゴリズムを組み合わせることでより高速化を図った物や, SIMD や GPU といった近年の計算機の構成に即したソートアルゴリズムが提案されているほか, 分散メモリ環境においての高速並列ソートなどの研究が盛んである [12][14]．これを踏まえて, 以下に近年の代表的な並列ソートについて挙げていく．

### 4.1.1　SIMD を用いた高速化

現在の計算機に用いられる CPU のほとんどは SIMD 演算が行える様になっている．SIMD 演算の概要を図 4.1 に示す. 通常のスカラ演算では図の左側のように 1 つの命令で 1 つのデータに処理を行うが SIMD 演算は図の右側のように, 一つの命令で複数のデータに対する処理を行う演算手法である. これにより実行を効率化出来, ソートアルゴリズムにおいても SIMD を利用することによって, 実行性能を上昇させる研究が盛んである．

ただし SIMD 演算には一定の制約があり, SIMD 演算が適用出来るような演算というものは限られる．ソートアルゴリズムにおいては, そのまま SIMD を適用出来るアルゴリズムはほぼ存在しない．そのため Chhugani ら [14] は merge sort をベースにしたソートで, 末端の部分を SIMD 演算の可能なソーティングネットワークを用いる事によって高速化を行った. その結果逐次版と比較して最大 3 倍以上の性能を得ている．また SIMD 演算を行わない部分についてもメモリバンド幅を考慮した Multi-way merge 等種々の最適化を行っており, 単純な並列ソートとしても高い性能を得ている．また, 井上ら [9] は SIMD を有効利用しつつ, 計算機のキャッシュサイズに収まる場合と収まらない場合についてそれぞれバブルソートの改良である Comb sort と, merge sort をベースとした SIMD 利用可能な異なるアルゴリズムを用いることでキャッシュミス率等を削減し, 高速化を行なった．Satish[33] らは様々な SIMD を用いた並列ソート同士の比較実験の結果, SIMD レジスタの幅が広がるなどしていくと考えられる将来的な計算機においては, 並列 radix sort よりも並列 merge sort の方が適しているとした．

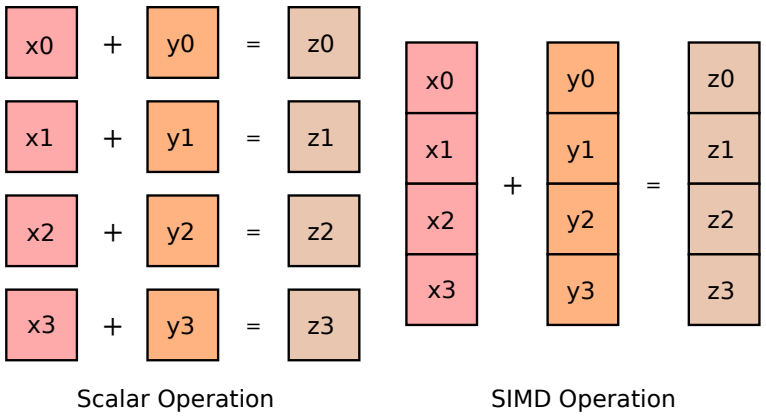


図 4.1. SIMD 演算

本節で挙げた研究の他, 近年の高速並列ソートアルゴリズムはいずれも merge sort 及び radix ソート, sample sort がベースとなっているため, 速度は高速であるが, メモリ使用量は $O(n)$ となっている.

### 4.1.2　分散メモリにおける並列ソート

4.1.1 節に紹介した研究は全て共有メモリ環境で動作することを想定されたソートアルゴリズムである. 一方で, 近年の大規模並列計算機の中にはクラスタシステムを代表とした分散メモリ環境が増加してきており, 分散メモリ環境での高速並列ソートの研究も盛んである.

ソートアルゴリズムはその性質上, 配列中の要素を移動させるためノード間の通信は避けられないが, 分散環境においてはこの際の通信コストが実行時間に大きく影響を与えるため, 逐次計算量やクリティカルパスだけでなく, 効率の良い通信を行えるかが大きなファクターとなる. このため共有メモリで並列性能が高いソートアルゴリズムでもノード間の通信が頻発するアルゴリズムでは実行時間は長くなってしまう. 2 章で軽く述べたように, Sample sort はノード間通信がノード毎に要素を振り分けた後はほぼ発生しないため比較的分散環境に対して相性が良く, Kim[17] らは 256 ノードの Xeon X5680 クラスタで Sample sort をベースとした CloudRAMSort を実装し, 1TB のデータを 4.6 秒でソートした.

分散メモリ環境においては, 並列ライブラリ側の実装も共有メモリ環境と比較して進んでおらず, 特に効率的に動くタスク並列モデルに関しては研究が盛んではあるものの実用レベルに達している物はこれまでの所存在しないが, 今後の処理系側の進展を見込んで, 計算機の構成に実装を大きく変更しないで済むタスク並列処理系で高速な並列ソートを考案することには意義がある. また, 通信コストが重い分散環境において, 通信数を減らす事になるため省メモリであることはより利点があると考えられる.

## 4.2　ワークスチーリング方針の改良

3 章で述べた様に, 多くのタスク並列処理系ではプロセスがアイドル状態になった時に発生するワークスチールはランダムに対象を選択しており, 負荷分散の面では問題ないことが示されている [18]. しかし, メモリから広範囲のデータを読み込む問題ではランダムなワークスチールが必要以上のキャッシュミスを生じさせ, 実行時間に悪影響を整えるなど扱う問題によっては必ずしもランダムなワークスチーリングが適しているとは言いがたい.

このような問題を解決するため, タスクの生成時のタスクのキューへの入れ方や, スチールを行う際のタスクの個数などの調整によって最適化を行う研究は以前からされているものの [30][31], より高度なチューニングを行う場合にはスケジューリング手法に何らかの変更を施すことは必要不可欠であると言える.

本節ではタスク並列処理系側のプログラムの実行速度を向上させる研究の中から特にワークスチーリング方針の変換に焦点を当てた研究を挙げる.

### 4.2.1　階層型キャッシュを考慮したタスクスケジューリング

ランダムスチーリングによるキャッシュミスの問題について軽く触れたが, プライベートキャッシュ環境に置いてはランダムスチーリングによって, キャッシュミス由来の大幅な悪影響を与えることは無いことが示されている他, 効率のよいスケジューリング手法も提案されている [18][26].

しかし現在の大規模な計算機ではほぼ共有キャッシュが採用されており, このような場合ランダムスチーリングを行うと, 実行時間が大幅に増加してしまう恐れがある.

Chowdhury らはキャッシュミスを防ぐために, タスクをその大きさに応じて区分を行う Space-bounded scheduler を提案した [27]. この手法では, あるタスクの親タスクがレベル i キャッシュに収まらず, かつそのタスクがレベル i キャッシュに収まる時に, そのタスクを *level-i* タスクと定義する. あるプロセッサが一旦 *level-i* タスク $A$ を実行し始めた場合, そのプロセッサが属するレベル i キャッシュのメンバーのみが, そのタスク $A$ 及び $A$ の子タスクのスチール・実行を可能にし, その他のプロセッサにはスチールさせないという制限を加える物である.

具体例を図 4.2 に示す.*level*-3 タスクであるタスク $t$ がプロセス P1 に実行された場合, 同じ L3 キャッシュを用いているプロセス P4 までが $t$ 以下のタスクをスチール及び実行が可能となり, プロセス P5 以降はたとえアイドル状態でスチーリング可能な状態であっても, このタスク $t$ 由来のタスクをスチールすることは不可能となる.

このスケジュール方針によってキャッシュミスを確実に削減可能であるが, *level-i* タスク $t$ のみを実行している場合, アイドル状態であるレベル i キャッシュのメンバが制約によって $t$ 由来のタスクのスチーリングが不可能であるため, アイドル状態で待機してしまうといったスケジューリング由来の待機遅延が発生するため, 実行時間の限界が保証されないという欠点がある.

問題や計算環境によっては例えキャッシュミスを生むとわかっていても並列実行を行なった方が結果的に実行時間が減るということも多い. そのような問題を抱えているため, Space-bounded scheduler は非常に primitive なキャッシュミス削減のためのワークスチーリング手法であると言える.

Blelloch らはこの手法を改良し, キャッシュミスとプロセス稼働率のバランスを上手く取れるスケジューラを提案した [29]. 提案されたスケジューラでは *level-i* のタスクの定義をレベル i キャッ

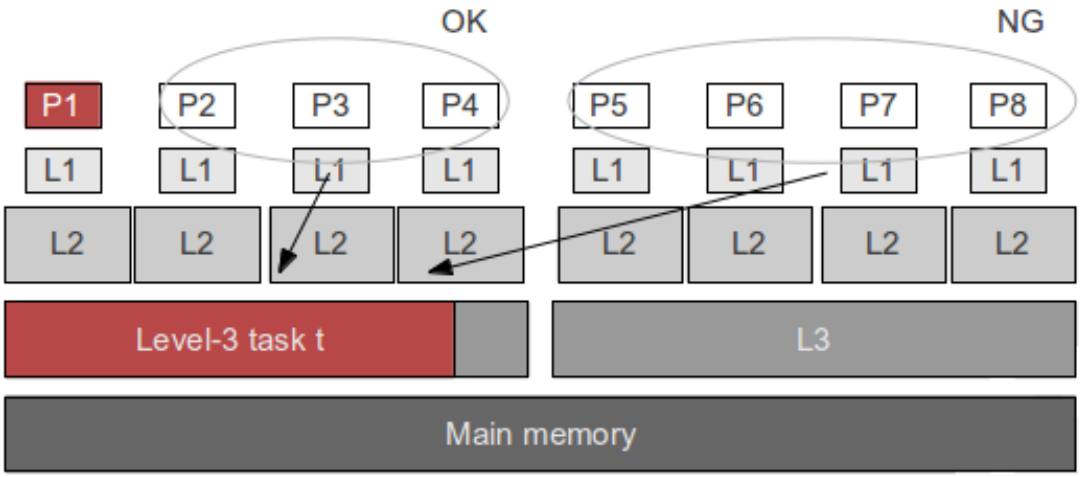


図 4.2. Space-bounded scheduler

シュのサイズの 1/3 に収まる物とし, 既にキャッシュで *level-i* タスクが実行されている場合には許容されていなかったキャッシュをオーバーしてしまう新たな *level-i* タスクの実行を一定の範囲で許容する他, 子タスクのスチールに関して別のレベル i キャッシュに属するコアからのスチールを制限するなどの改良が行われている. これにより, 実行時間を最小化するためのキャッシュミスとプロセス稼働率のバランスを上手く取ることを可能とした.

また, Blelloch らはキャッシュを一切無駄なく利用出来る, 理想的なキャッシュを想定して, 大まかなキャッシュの使用量とキャッシュミス数からアルゴリズムを評価する CO (cache-oblivious) モデル [28] を拡張し, 並列計算にも対応した PCO(parallel-cache-oblivious) モデルを提案し, これを用いて提案スケジューラを用いる場合実行時間が上限を持つことを証明した.

### 4.2.2　負荷分散方式を変更可能なスケジューラ

スチーリング方針の変化によって問題に対して高度なチューニングを行う場合, 最適なスチーリング方針は当然扱う問題によって異なると言える. そこでスチーリング方針をプログラマが変更する必要があるが, タスク並列処理系の内部は非常に煩雑であり, 変更を行うためにはプログラマ側に高度な知識が求められてしまう. この問題は逐次プログラムから大きな変更なく並列プログラムを記述可能というタスク並列処理モデルの大きな利点を損なう物であり, 望ましくない.

当研究室の中島らは, ワークスチーリングの戦略をカスタマイズ可能なタスク並列処理系である, MassiveThreads を提案した [24]. MassiveThreads では, プログラマがプログラムに問題に最適化したスチーリング関数を記述することで, 並列実行時に発生するワークスチーリングの方針を変更することが可能である. これによりプログラマは難解なタスク並列処理系の内部を直接変更することなく問題に最適化したワークスチール方針を実装することが出来るため, プログラマに求められる知識の敷居が大きく下がったと言える.

中島らは実際に, 粒子法などで用いられる適合細分化格子法 [25] の一種である Tree-based AMR

に対して, タスク木の深さを考慮し, スチール時に参照したコアに盗めるタスクが存在する場合でもすぐにスチールを行わず, 複数のコアを参照し最も根に近いタスクをスチールする depth-aware なスチーリングを実装した. この変更により, 動的負荷分散の効率が上昇し, スチール回数が減ったことにより実行速度が改善された.

# 第5章　鋸ソート

本章では, 本研究で提案する省メモリでスケーラブルな並列ソートアルゴリズム「鋸ソート」について述べる. 2.1 節で並列ソートに重要な要素について触れたが, 各種ソートアルゴリズムの基盤となるような物の中で低クリティカルパスと省メモリを兼ね備えている物は bitonic sort のみである.

2.4 節で触れたように, この二つの要素に関してオーダーレベルの改良を図るのはアルゴリズムの抜本的な変換が必要なため難しいと言える. そのため bitonic sort を基盤として, この二つに対する利点を保持しつつ, 逐次計算量の多さとソート列への弱さという二つの大きな欠点を克服したものが提案手法である鋸ソートである.

本章の構成は以下である. まず, 5.1 節でアルゴリズム全体の概要を説明し, 5.2 節と 5.3 節で提案手法のために定義する鋸列と, 鋸列を用いた再帰的マージの成立要件に関して述べる. これを受けて 5.4 節でアルゴリズムの主軸である鋸マージと, 状況に応じて用いるその簡易版についての説明を行う.5.6 節で計算量の評価を行い, 5.7 節でその他のソートアルゴリズムとの性能比較と考察を行う. なお, アルゴリズム中の spawn と sync はそれぞれタスク並列モデルにおける, タスクの生成と生成されたタスク群の終了同期のための命令とする.

## 5.1　アルゴリズムの概要

提案アルゴリズムである鋸ソートは分割統治法を用いてソーティングを行う. 図 5.1 に鋸ソートの概要を示す. 前に述べた通り bitonic sort を基盤とするため, 全体の大きな流れとしては bitonic sort とほぼ同じとなる. 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. 全体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れらをマージすることで全体のソートを完了する. 分割された領域はそれぞれ, 最終的にマージされるまでは一切お互いのデータに触れないため並列実行が可能である.

この分割作業は配列長 length が 1 になるまで行うことも可能だが, コア数に対し十分な数のタスクが生成出来ればその時点で並列度は問題なく出せる. そのような状態で繰り返す再帰呼び出しを行うことは不要であるばかりか関数呼び出しのオーバーヘッドなどから逆に遅くなってしまう事が多いため, 実際のプログラムでは設定した閾値以下の長さでは逐次でソートを行った. この際のソートアルゴリズムとしては逐次性能が高い物の中から用途に合ったものを選択するのが望ましい.

```
1  void saw_sort(int start, int length){
2    // しきい値T以下になるまでは前半後半に分割して再帰
3    if(length > T){
4      int m = length/2;
5      spawn saw_sort(0, m);
6      spawn saw_sort(m, length);
7      sync;
8      // それぞれの結果をマージ
9      saw_merge(0, length);
10   }else{
11     // 十分に短くなったら逐次ソート
12     sequential_sort(0, length);
13   }
14   return;
15 }
```

図 5.1. 鋸ソート

## 5.2　鋸列

アルゴリズムのマージ部分において, 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のソートを行う場合は bitonic 列の生成の過程で入力のほぼ昇順のデータを, 昇順→降順といった形にしてしまうため, 元々のソートされている部分を全く活かすことが出来ず, ランダム列とほぼ変わらない実行時間になってしまうといった欠点が顕著であった.

提案アルゴリズムでは, 入力列が持つソート部分を活かすため, 鋸列という並び方を定義し, 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れによりソート全体の動きは通常の merge sort に近い動きとなる一方で, 通常の merge sort と異なり, bitonic merge をベースとしたアルゴリズムを採用しているため, マージの際に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, ほぼソートされた列が入力であった場合に, それを活かした計算量の削減を行うことが可能となる. しかし, 「昇順→昇順」という接続にしたため, bitonic 列では保証されている, 「長さ $n$ の bitonic 列において, $\frac{n}{2}$ 離れた要素同士は必ず前半と後半に分かれる」という特性が失われてしまう. そのため, シフトを認めてしまった場合, 再帰的なマージが著しく困難になってしまうため, シフトを認めないという新たな条件が追加される.

## 5.3　再帰的なマージを成立させるために必要な要素

本節では, 鋸マージの理解のために, 再帰的なマージを成立させるために必要な条件を記述する. 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.

このような再帰的に実行するマージ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, 大きな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全体の配列と同じ性質を持つ

事が挙げられる. 前者の条件は, 関数の構造上再帰呼び出しされる前半と後半がそれぞれデータ的に独立であるため最終的に前半にあるべき要素が後半に残ってしまう場合, 最終的にソートが失敗してしまうため設定され, 後者の条件は, 再帰的に同様の処理を行うため $n$ 回目の入力が最初の入力と同じ特性を持っていなければならないために設定される.

bitonic sort における bitonic merge はこの条件を満たしているため, 再帰的なマージが可能となっている. 同様の条件を鋸列にも当てはめると, 鋸列は bitonic 列と比較して, 要素のシフトが出来ないという点で自由度が低く, それを考慮した要素の扱いが求められる.

具体的には, 前者の条件を満たすために鋸列において bitonic merge と同様に配列全体の前半分にある最大候補と後ろ半分にある最小候補の比較交換をナイーブに行なった場合, その回は問題なく完了するが次の入力が昇順列 3 つ以上の連続になる場合があり, 後者の条件を満たさなくなる. その場合次回のマージ関数では最大候補及び最小候補が同定出来ずにマージが行えなくなる. そのため, これら二つの条件を満たすアルゴリズムを用意することが鋸列に対して再帰的なマージを行う上で必須となる.

## 5.4　鋸マージ

図 5.2 に鋸マージの擬似コードを示す. 実際には二つの並んだソート列のどちらが大きいかで場合分けが生じるが, 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 5.3 に擬似コードで行われる操作の各ステップを可視化した. 右下の最終状態において, 前半分と後半分がそれぞれ鋸列となることが確認出来る.

マージの際には, まず前半分の最大候補と後ろ半分の最小候補を同定する必要があるが, 呼び出し元の関数から引数の flag によって 2 つのソート列の切れ目を与えることで, これらの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. 前節で触れたように, 中央に二つのソート列の境界がない場合は, ただ移動すべき要素の交換を行うだけでは再帰的なマージを行うことは出来ない. そこで step2∼step4 で境界が中央でない場合は要素の反転を上手く利用することで前半と後半をそれぞれ鋸列に補正し直すといった操作を追加で行っている. 尚, 二つ目のソート列の方が大きい場合は step2 が前半分を鋸列にするための準備操作に置き換わる.

図 5.2 ではマージ操作に関しては逐次実行のコードを記したが, それぞれのステップで行われる 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いるため, これも並列実行が可能である. このような部分に関しては, アルゴリズム中では forall で括る事とする. 実際には並列実行されるため, クリティカルパスのオーダーは変化しない.

```
1  void saw_merge(int start, int length, int flag){
2    if(length > M){
3      int m = length/2;
4      int swapt; // swap すべき回数
5      // step:1
6      // 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
7      forall{
8        for(int i = 0; i < length−flag; ++i){
9          if (items[start+m−1−i] > items[start+flag+i]){
10           swap(items[start+m−1−i], items[start+flag+i]);
11           swapt++;
12         }
13       }
14     }
15     //step:2
16     // 後半分を鋸列にするための準備
17     forall{
18       for(int i = 0; i < (flag−m)/2; ++i){
19         swap(items[start+m+i], items[start+flag−1−i]);
20       }
21     }
22     //step:3
23     // 前半分を鋸列に
24     forall{
25       for(int i = 0; i < swapt/2; ++i){
26         swap(items[start+m−swapt+i], items[start+m−1−i]);
27       }
28     }
29     //step:4
30     // 後半分を鋸列に
31     forall{
32       for(int i = 0; i < (flag−m+swapt)/2; ++i){
33         swap(items[start+m+i], items[start+flag+swapt−1−i]);
34       }
35     }
36     int left = m−swapt;
37     int right = flag+swapt−m;
38     // 前半と後半を再帰的にマージ
39     spawn saw_merge(0, n, left);
40     spawn saw_merge(m, n, right);
41     sync;
42   }else{
43     // 長さが M 以下になった所で一事領域を使った高速マージに移行
44     high_memorymerge(items, start, length);
45     }
46   return;
47 }
```

図 5.2. 鋸マージ

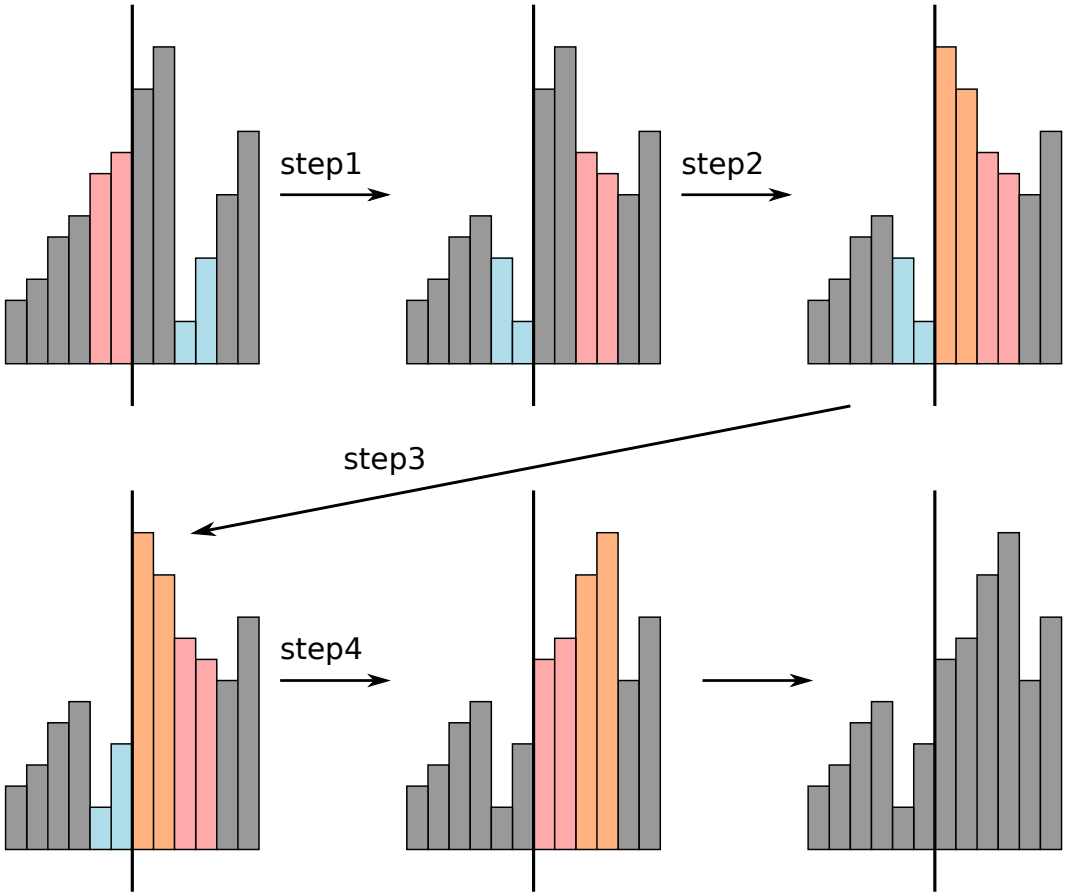


図 5.3. 鋸マージ

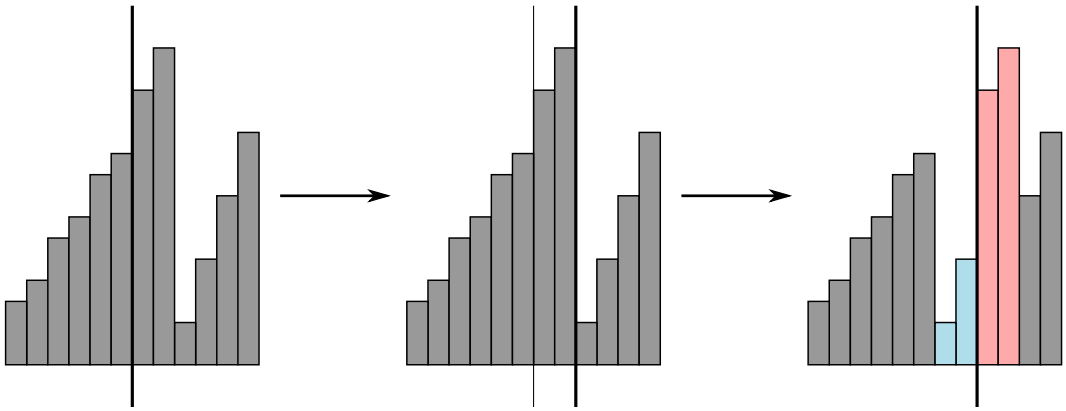

図 5.4. 簡易鋸マージ

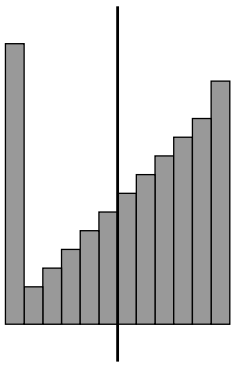

図 5.5. 両側に偏りがあるケース

### 5.4.1　簡易鋸マージ

鋸マージは要素の反転を利用して, 鋸列を二つの鋸列に分割した. この操作一連の計算量のオーダは $O(n)$ であり, bitonic sort から変化しないが, 反転を行うために実際の逐次計算量は 2 倍弱近くに増加してしまう.

理論上のクリティカルパスは変化しないものの, 現実的にはプロセス数 $p$ が配列長 $n$ と比較して遥かに少ない. そのような状況下で, 計算量における定数倍の増加は実行時間に大きく影響するため, 無視出来るものではない.

そこで, 基本的には図 5.4 で表現される簡易鋸マージを採用する. こちらは鋸列を構成する二つのソート列の境目でマージ対象を二つに割り, 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, 計算量を大きく抑えることが可能である.

しかし, 問題点も存在する. 図 5.5 の様に二つのソート列の大きさが大きく偏った状態が入力として与えられた場合, 一回の簡易鋸マージに対して 1 個ずつしか左の大きな要素が移動しないため, 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.

結果として, 関数の再帰呼び出しや二分探索のオーバーヘッドが大きな悪影響を与える他, スタックオーバーフローを引き起こしてしまう.

このような入力に対しては通常の鋸マージを行うべきであるため, 二つのソート列の切れ目が全体長からみて一定の割合以上両端に寄った場合, 通常の鋸マージを行うという実装にした.

通常の鋸マージならば, 入力配列が図 5.5 のようなケースにおいても高々$\log n$ 回の再帰呼び出しで正しい位置に移動することが可能である.

## 5.5　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省メモリであることのみとなる. しかし, 一つの関数が扱うデータ範囲は末端に行くほど狭くなり, 省メモリ効果は薄くなっていく.

省メモリにこだわる必要が無くなっている末端においては, ソート全体の最初の処理で末端の操作を変えた時と同様に, 条件に合ったアルゴリズムの中から用途に合ったものを選択するのが望ましい. 「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, merge sort を用いることで $O(n)$ の操作で終了することが可能である. さらに, merge sort は Clik sort のように並列化を行う事が出来るため, この変更によりクリティカルパスが増加することはない.

そこで, 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 $m$ の領域を使った逐次計算量 $O(n)$ で済む merge sort を行う事とした.

この定数 $m$ 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, メモリ使用量に対する悪影響を殆ど与えること無く, 高速化が可能である.

## 5.6　計算量の評価

ここで, 提案アルゴリズムの計算量を評価する. 基盤とした 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 $m$ 以下ではメモリ領域を使用した高速なマージ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 $O(n(\log n - \log m + \frac{m}{n}))$ 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. この $m$ の調整によって計算量とメモリ使用量の調節が可能である. m の値によるオーダーの変化を表 5.1 に示す.

$m$=0 の時は bitonic sort と, $m$=$n$ の時は Cilk sort と同じオーダーを持つことが分かる. $m$ 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. $m$ は任意に調整が行えるため, 調整次第で多少の一時領域を用いるだけで大きく高速化を図ることが可能である. 例えば, $m = \sqrt{n}$ の場合, 逐次計算量のオーダーは変化しないものの, 係数として $\frac{1}{2}$ がかかるため, 実際の実行時間は大きく短縮出来る上メモリ使用量は Cilk sort と比較した場合大きく抑える事が出来る.

## 5.7　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について, 提案手法を加えてまとめた物が, 表 5.2 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弱さの改善を達成していることが分かる.

表 5.1. 鋸ソートの計算量と余分なメモリ使用量

| m の値 | 逐次計算量 | メモリ使用量 | クリティカルパス |
|---|---|---|---|
| n | $O(n \log n)$ | $O(n)$ | $O(\log^2 n)$ |
| $\frac{n}{\log n}$ | $O(n \log n \log\log n)$ | $O(\frac{n*p}{logn})$ | $O(\log^2 n)$ |
| $\sqrt{n}$ | $O(n \log^2 n)$ | $O(\sqrt{n} * p + \log n)$ | $O(\log^2 n)$ |
| 0 | $O(n \log^2 n)$ | $O(\log n)$ | $O(\log^2 n)$ |

表 5.2. 並列ソートアルゴリズムと提案手法の性能比較

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

# 第6章　共有キャッシュを考慮したワークスチール

本章では, 本研究で提案するタスク並列モデルにおける共有キャッシュを考慮したワークスチール方針について述べる.

3.3 節で述べたように, タスク並列処理において, アイドル状態となったスレッドが行うワークスチールの対象は通常ランダムで決定され, 負荷分散に関しては問題がないとされている. しかし, 同一ソケット内で走っているタスクのワーキングセットが共有キャッシュサイズを超える場合, タスクの実行中に, ソケット内のコア同士でキャッシュの追い出し合いが起こる. 結果としてこのような状態の間, 共有キャッシュミスが発生し続けてしまう. ランダムスチーリングにより様々な箇所のタスクが同ソケット内で走る場合, 特にこのような状態に陥りやすい.

共有キャッシュは基本的にメインメモリに近い階層であるため一回のミスのコストが比較的大きく, 現在の大規模計算機の殆どは共有キャッシュを持つため, タスク並列処理系をこれらのアルゴリズムで用いる場合に考慮せねばならない問題の一つであると言える.

本研究が提案するワークスチール方針では, 計算機の各ソケットに１つ権限の強いコアを設定し, そのコアのみが他のソケットに存在するタスクを盗みに行く権利を持つ Socket-aware なスチール方針とプログラムがタスクを作る際にタスクにヒントを付加し, 各ソケット内での共有キャッシュミスを削減するための Cache-aware なスチール方針を組み合わせたものを用いることで, キャッシュミスの削減を達成する.

本章の構成は以下である. まず 6.1 節で扱う対象となる問題についての説明を行い, 6.2 節と 6.3 節で提案アルゴリズムのスチール方針について述べる.

## 6.1　対象とする問題

本研究ではソートなどに代表される, 広範囲のデータを扱う分割統治法で記述されるアルゴリズムについて特に扱う. このような問題は処理全体のタスク木は n 分木の形状をとる. 扱うデータ範囲を再帰的に分割するため, 問題によって異なるものの, 基本的に子ノードの扱うデータ範囲はその親ノードが扱うデータ範囲の一部であるため, 木構造の葉に近いタスクほど扱うデータ範囲は狭くなる.

先述したように, 同一ソケット内で走っているタスクのワーキングセットが共有キャッシュサイズを超える場合, 共有キャッシュミスが発生し続けてしまう. そのため, 各コアが単独で実行することになるデータ範囲, 即ちその部分木の根となるノードの深さがソケット内の総計が共有キャッシュサイズを超えないような値になっている事が望ましい.

しかし, 一般的なタスク並列処理モデルに置いては, ランダムワークスチーリングが採用されているため, 上記のような都合の良い状態になることはあまりない. ワークスチールが発生する場合スチール対象の中で最も根に近いタスクを奪うため各コアはタスク木で離れた位置にある比較的

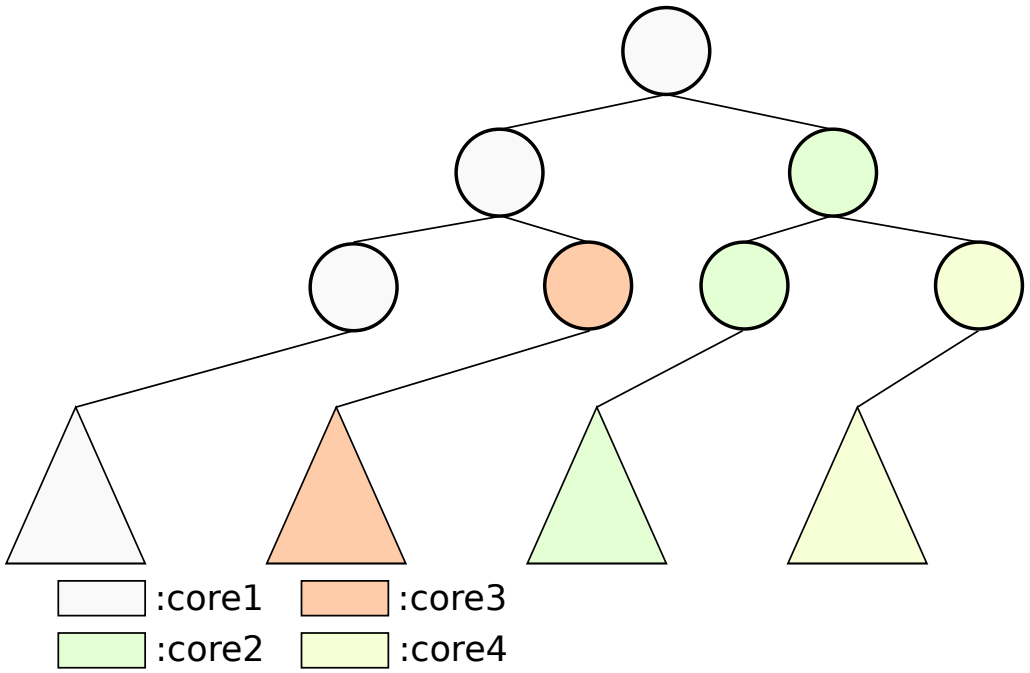


図 6.1. 分割統治法のタスク木

大きな部分木を扱う傾向がある. 理解のため, 4 コアでこのような問題をタスク並列モデルで実行した場合のタスク木の様子を図 6.1 に示す. アイドルコアが無くなるまではほぼ均等に分割され, その後の計算によって, それぞれのコアの扱うデータ範囲は配列の $\frac{i}{4} \sim \frac{i+1}{4}$ にあたる部分となる. 配列サイズがこれらのコアの共有キャッシュを大幅に上回る場合は多く, このようなタスク配置が行われると, 実行時に共有キャッシュミスが大量に発生してしまう. Work-first 実行であるためプライベートキャッシュミスに関しては比較的 friendly であるが, 扱うデータ範囲がある程度大きくなるとその利点も薄れてしまうため, このような問題においては単純なランダムワークスチーリングは適さないと言える.

## 6.2　Socket-aware Steal

そこで本研究では共有キャッシュミスを減らすためのスチール方針として, ソケットを考慮したワークスチール方針を提案する.

図 6.2 に概要を示す. ランダムワークスチーリングでは各コアが全てのコアの中からランダムでスチール対象を選択するが, 提案アルゴリズムでは全てのコアを対象と出来る物は各ソケットに 1 つのみとする. 以下, このコアを便宜上マスターコアと呼び, その他のコアをメンバーコアと呼ぶことにする. マスターコアはプログラム開始時に設定され, マスターコア以外のメンバーコアは同ソケット内のみを対象としてスチールを行い, 基本的に他のソケットにワークスチールを行うことを制限する. また, マスターコアも同ソケット内から優先的にスチールを行い, 同ソケット内にスチール可能なタスクがない場合のみ他ソケットに対してのスチールを試みる方が望ましい. こ

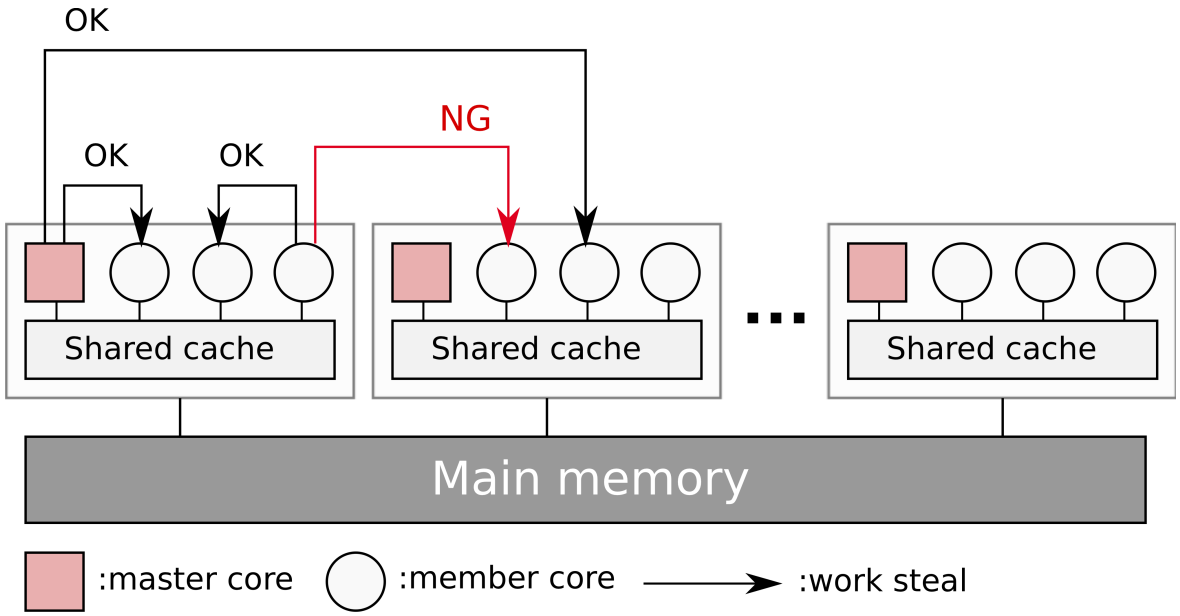


図 6.2. Socket-aware Steal

れにより, 同じソケット内で一つの部分木を扱えるため, 共有キャッシュの利用効率を向上させる事が可能である.

### 6.2.1　周回ワークスチーリング

提案アルゴリズムによって, ソケット内で最初にマスターコアがスチールしてきたタスクから生成される一つの部分木に対してのみの並列処理を行うことが可能となった.

一方で一つの部分木に対して実行コアが集中することで 1 回のスチールによって得られるタスクの深さがランダムスチーリングの場合と比較して深くなるため, スチール回数が増加しがちになる. 末端の軽いタスクを何度もスチールし合うといった状況はあまり望ましくないため, スチールの質を改善するためソケット内のスチールには周回スチーリングを用いた手法を採用した.

周回スチーリングは図 6.3 のように自分の隣のコアから各コアを順番に見ていく手法であり, スチール対象の決定に比較的重い random 関数を呼ばなくてすむため比較的早く, コア数が $p$ の時 $p-1$ 回の参照で全てのコアを見回る事が可能であるという利点がある [32]. 一方で, あるコアでスチール可能なタスクがあった場合にすぐ盗む greedy な手法を取る場合, コア間のタスクの深さに偏りが出来, 負荷分散がうまくいかず性能が悪化する可能性があるといった欠点がある. そのため周回スチールを採用する場合, 全コアを取りこぼしなく見まわる必要がある問題や複数コアを参照する問題が望ましい.

提案アルゴリズムでは各タスクには再帰呼び出しの深さをヒントとして予め付加しておき, スチール対象として同ソケット内の全てのコアを周回し最も根に近いタスクにスチールを行う. また, マスターコアはソケット外に探しに行くかどうかの判定のために同ソケット内のコアに盗めるタスクが無いことを確認する必要がある他, ソケット外へのスチーリングはなるべく減らすこ

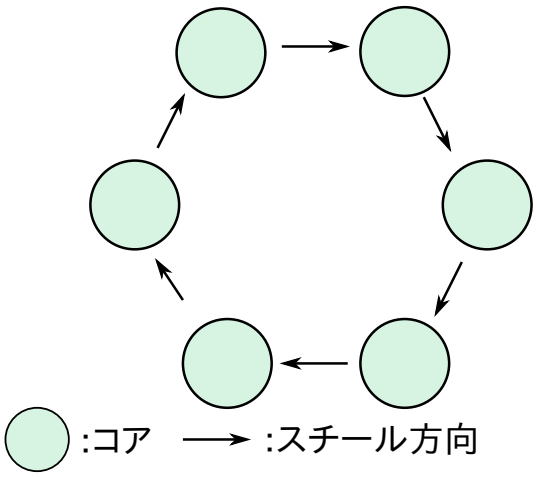


図 6.3. 周回スチーリング

とが望ましいため, その時スチール可能な最も深いタスクを盗むべきである. このためそれぞれの場合で周回スチーリングは相性が良いといえる.

### 6.2.2　考慮すべきケース

本研究で扱うような問題に関しては, 上記の方針で問題なく Socket-aware なタスク実行が行えるが, タスク並列処理系で扱う問題の中にはあるタスクが次のタスクを生むまでに大量の処理を行う物も多い.

このようなタスクを, マスターコアがスチールした場合, 本来全体を俯瞰すればスチール可能なタスクがあるにも関わらず, その権利を与えられないそのソケットのメンバーコアはアイドル状態となってしまう. キャッシュミスを考慮する余り並列度そのものが落ちてしまい実行時間が遅くなっては本末転倒であるため, このような状況に陥らないために, メンバーコアに対してもある程度スチールが失敗している場合に限りソケット外のコアにスチールを行う権利を与える, などといったソリューションを与える必要があると言える.

## 6.3　Cache-aware Steal

前節で述べた Socket-aware Steal によって各ソケットが同じ部分木を扱う事が可能になり, それぞれの部分木が取り扱うデータ範囲が共有キャッシュに収まる場合, 共有キャッシュミスの削減を図ることが可能になる. しかし, Socket-aware Steal ではソケット内では負荷分散を重視したスチールを行うため部分木全体が取り扱うデータ範囲が共有キャッシュの容量を上回る場合, 結局ソケット内で走るタスクの扱うデータの和集合は共有キャッシュを超えてしまい, ランダムスチーリングを行う場合と大差ない共有キャッシュミスが発生してしまう.

これを解決するため, 各タスクが触れるデータ範囲に応じてスチールしていいかどうか判定を考慮する必要があり, 図 6.4 に理想的なタスクの並列実行の様子を示す. タスク木の先端において,

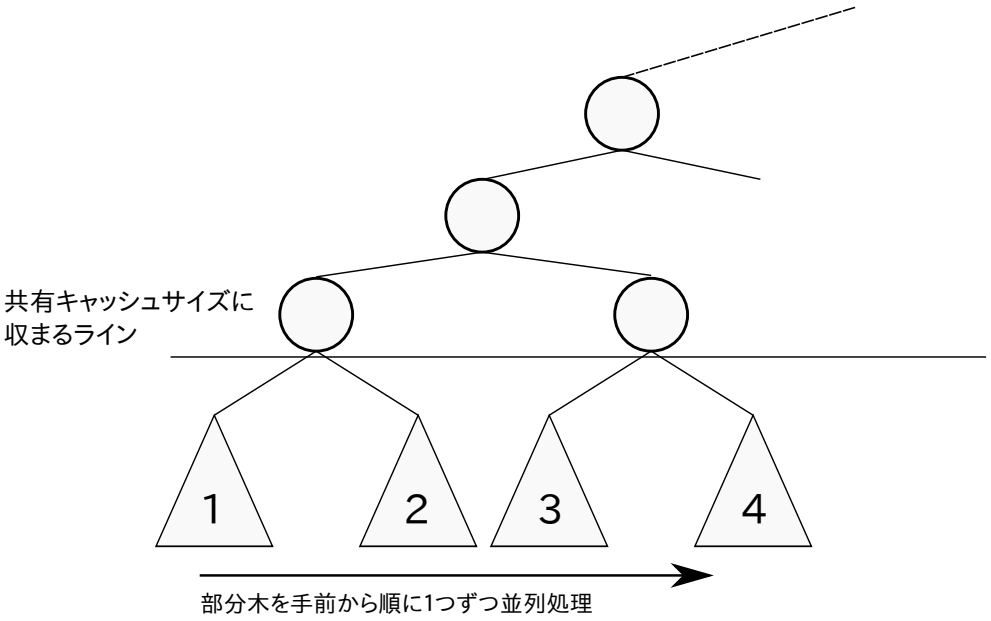


図 6.4. Cache-aware Stealing

あるタスクが共有キャッシュに収まる範囲になった場合にそのタスク以下のみをソケット内で実行することによって, そのタスクから生じる共有キャッシュに収まる部分木ごとにソケット内のコアで並列実行を行う事が可能となる. 結果, 共有キャッシュミスを削減出来る.

### 6.3.1　実行順序の制御

Cache-aware Steal は理論上は前述したような実行を行えば可能であるが, 実際に図 6.4 のようなタスク実行を行うためにはタスクの実行順序を制御しなければならず, ソケット内で更にスチールに関しての何らかの制約を加える必要がある.

あるソケットのマスターコアで１つのタスクが実行され始めたとき, まずタスクキューに入って行くのは根に近いタスクであるが, これをメンバーコアにスチールさせてしまうと Cache-aware Steal が行えなくなってしまう. マスターコアの実行が共有キャッシュに収まる範囲になったときに, メンバーコアは別のタスクを実行している事になるが, これを割り込みで中断させることは不可能であるためである.

これを回避するため, 共有キャッシュに収まるタスクが生成されるまではメンバーコアにはスチールさせないような制約をタスク側にヒントを与えるなどして行う必要がある. しかしこのような制約を行なった場合, タスクが共有キャッシュに収まった後も, 通常の実装ではスチールが行われないという問題が生じる. ワークスチーリングでは基本的に対象のタスクキューの先頭のタスクのみを参照するため制約がかかった根に近いタスクが先頭にある状態ではキューの奥にあるスチール可能なタスクを取る事は不可能である.

このような問題を解決するためにはいくつかの方法が考えられる.

- ワークスチールの際, スチール対象のタスクキューの先頭だけでなく全てを見るようにする.

- 通常のタスクキューの他にスチール対象となっても参照しないタスクキューを作り, スチールされたくないタスクはそちらのキューに入れる.
- 実行初期にソケット数分のタスクを生成し, 各ソケットのマスターコアに分散させる. その後は, 共有キャッシュに収まらない範囲ではタスクを生成しない様にアプリ側を変更する.

一つ目の手法は最も素直な方針であり, 柔軟性も高い. しかし, タスクキューの先頭以外を参照することによるレイテンシは無視出来ないため, タスクキューの先頭に実行中で無いが, 制約によりスチール不可能なタスクがあった場合のみ奥を参照するなどの工夫でなるべくレイテンシを削減する必要がある. 二つ目の手法は, 新たに作ったメンバーコアが参照しないキューもマスターコアのみが参照出来る様にする事によって, Socket-aware-steal との親和性も高い.

これらの二つの手法のいずれかを取るのが理想的ではあるが, どちらも並列タスク処理系側の処理であるため, そのような機能が実装されていなければ実装を行う必要がある. 一方で三つ目の手法は, アプリ側の再帰呼び出し時のタスク生成の有無を調節することでタスクの動きを調節するため, 多少融通は聞かず, 実行する問題によっては並列度を多少下げてしまう恐れはあるもののアプリ側の書き換えで済むため, 処理系内部の変更を行う事無く実装が可能であるという利点がある.

## 6.4　提案アルゴリズム

本研究では, 本研究室が開発を行っているタスク並列ライブラリ MassiveThreads を用いて行っており, 4.2.2 節で触れたように, MassiveThreads ではスチール関数を実行プログラム側で設定することにより, プログラマが難解なタスク並列ライブラリの中身を弄ることなくスチール方針などを変更する事が可能である.

現状の MassiveThreads ではスチールを行うコアを決定する方針と, タスクにヒントを与えスチールの際にそれを読み取ってスチールを行うかどうかの判定をすることが可能であるが, タスクキューの奥を参照したり, 新たなタスクキューを作成したりすることは現状の MassiveThreads では未実装である. そのため, 本研究で扱う鋸ソートではあまり大きな悪影響を与えないと考えられる事から, Cache-aware Steal の部分に関しては実装が比較的平易である三つ目の手法を採用した.

実際に用いたワークスチールアルゴリズムの擬似コードを図 6.5 に示す. スチール対象を決定する関数を, MassiveThreads が通常用いるものからプログラム側で既述したこの関数に差し替えるという形で実際には運用する. この返り値である target に等しい番号のコアに対し, この関数を抜けた後実際のスチーリングが行われる.

コード中の SOCKET はマシンのソケット数を, CORE は各ソケットのコア数を示す実行計算機依存の定数である. 鋸ソートアルゴリズム側では, 再帰の深さをヒントとしてタスク生成の時にタスクに与えている. ソートであるため, 再帰の度に触れるデータ範囲は狭まっていくので, 深さが低いほど根に近くなる. また, 簡単のためコア番号の取得などに関する関数など MassiveThreads に実装されている物についての関数内部の詳細や, 対象とする鋸ソートの実行の際には考慮する必要性が無い 6.2.2 節で触れたメンバーコアの例外処理に関しての関数の記述及び実行開始時に各ソケットにつき 1 つタスクを割り当てるための処理は省略した.

```
1  int cache_aware_steal(int rank)
2  {
3    int n_can;// スチール対象として見に行くコアの個数
4    int thnum = myth_workre_num();// 関数で実行中のコア番号取得
5
6    bool master = master_checker(thnum);// コア番号によって判定
7
8    if{master == true){
9      n_can = SOCKET*CORE-1;
10   }else{
11     n_can = CORE-1;
12   }
13   int candidates[n_can];
14   int socket_group = myth_socket_num(thnum);// 自分が所属するソケットをコア番号から取得
15   // スチール対象の候補を決める.
16   if(master == ture){
17     // マスターコアは自分以外全てのコアを周回
18     for(int i = 0; i < n_can; i++){
19       if(i == thnum)i++;
20       candidates[i]=i;
21     }
22   }else{
23     // メンバーコアはソケット内のみで周回
24     int thnow = thnum;
25     int i;
26     while(i < CORE){
27       thnow = thnow+SOCKET;
28       if(thnow >= SOCKET*CORE){
29         thnow = thnow - SOCKET*CORE;
30       }
31       candidates[i]=thnow;
32       i++;
33     }
34   }
35   // 実際にそれぞれの候補にヒントを読みに行く
36   int depth;
37   int target=-1;
38   int mindepth=99999;
39   while(target < 0){
40     for (int i=0;i<n_can;i++){
41       depth = get_hint(i);// 関数を用いて
                 i番のコアからヒントを取得　スチール可能なタスクがない場合-1
42       if(depth > 0){
43         // 比較してより浅い方を選ぶ
44         if (depth<mindepth){
45           mindepth=depth;
46           target=candidates[i];
47         }
48       }
49     }
50   }
51   return target;
52 }
```

図 6.5. 提案ワークスチールアルゴリズム

# 第7章　評価

本章では, 本研究の提案手法である, 鋸ソートと共有キャッシュを考慮したワークスチール方針の評価を行う.

## 7.1　評価環境

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[24] を用いて実装を行った.4.2.2 節で述べたように, MassiveThreads は, プログラマが実行プログラムにスチーリング方針を決定する関数を記述することで, 処理系の中身を変更することなくワークスチール時の動きを変更することが可能であり, この機能を用いて提案手法を実装した.　実験は 24 コアを持つマシンで行なった. 詳細を表 7.1 に示す.

## 7.2　問題設定

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 quick sort と, 提案手法の元となった bitonic sort, 代表的な高速並列ソートである Cilk sort について実装を行い, 測定を行なった. 配列には int 型の配列を用い, それぞれのソートにおいて配列についての 3 回の実行の平均から実行時間と使用メモリ量の測定を行なった. また, Clik sort は 2-way merge を採用し, 鋸ソートにおける逐次マージへの切り替えは $2^{15}$ からとした. また, ワークスチール方針は全て MassiveThreads に初期実装されているランダムスチーリングを採用した.

## 7.3　コア数に対する性能変化

並列性能を測定するため, コア数による実行時間とメモリ使用量の変化を調べた. メモリの使用量は getrusage システムコールを用いて測定した. 実験では長さ $2^{24}$ と $2^{28}$ の配列について行なった. それぞれ要素数 16M と 256M にあたる. 二種類の長さを用意したのは, 16M 要素のサイズが 64MB であり, 実験マシンの L3 キャッシュ18M*4B にほぼ乗り切るため実行時間においてキャッ

表 7.1. 評価実験に用いたマシンの性能

| CPU | 周波数 | ソケット数 | コア数/ソケット | 総コア数 | L1 | L2 | L3 |
|---|---|---|---|---|---|---|---|
| Intel(R)Xeon(R)CPUE7540 | 2.00GHZ | 4 | 6 | 24 | 32K | 256K | 18M |

シュミスの影響が支配的にならないケースであるためである．なお, 256M 要素のサイズは 1GB であり, こちらは完全にキャッシュをオーバーするケースを想定している.

また, 配列の種類として, 完全なランダム列のほか, 実アプリ上で頻出するほぼソートされた列として, ソート列に外れ値一つを加えたもの, ソート列のそれぞれの値に対し幅 100 の一様乱数を加えた物を用意し, それぞれ測定を行なった.

ランダム列に対する性能を, 表 7.2 に, はずれ値一つのソート列に対する各ソートの性能を 7.3 に, 幅 100 の一様乱数を加えたソート列に対する各ソートの性能を 7.4 にそれぞれ示す.

まず, ランダム列に対しての性能から考察を行う. 逐次実行時間に着目すると, 16M 要素の場合, 逐次計算量の差から quick sort と cilk sort は bitonic sort と比べ実行時間が速い中, 提案手法の saw sort も前者二つにやや劣る物のほぼ同時間でのソートを達成しており, bitonic sort からの逐次計算量の削減に成功していることが分かる．一方で 256M 要素の場合, どのソートもキャッシュミスが大幅に増加していくため, 実行時間は 16 倍以上になることが確認出来る. この時 quick sort が他のソートと比較してやや速くなっていることがわかるが, これは quick sort が最もデータの読み書きを行う回数が少ないためキャッシュミスの影響も少ないためと考えられる.

逐次性能を踏まえた上で, 並列性能に注目する.saw sort の逐次実行での性能を 1 とし, それに対する各ソートのスケーラビリティをグラフにしたものを要素数 16M と 256M についてそれぞれ図 7.1 と図 7.2 に示す. 図 7.1 から, quick sort は 8 コアを超えた辺りからスケーラビリティが劣悪になることが分かる. これは初回の逐次で行う実行分の時間がコア数が増えるに従いボトルネックとして顕著になるためである．その他のソートは高いスケーラビリティを得ており, Cilk sort と saw sort は絶対性能の点からもいい結果を残している. なお, コア数が多くなると実行時間が 0.2sec 以下と非常に短くなることからタスク並列処理系の負荷分散などのオーバーヘッドの影響が大きくなるため, 並列度の高いアルゴリズムでもリニアに性能があがるわけではないが, 十分高いスケーラビリティと言える．一方で, 要素数 256M の場合は全体的にスケーラビリティが落ち込み, 特に提案手法の高並列での減衰が大きい. 本来要素数が大きい方が負荷分散は取りやすいため, タスク並列処理系のオーバーヘッドの影響は相対的に減る. しかし実行時間が長くなってしまった原因としては, 共有キャッシュミスの影響や, 並列度が高くなり, かつ扱うデータが大きいためメモリバンド幅の上限に律速されてしまっているといった原因が大きいと考えられる．特に 16 コア以上の実験ではその影響が顕著である.

次に, はずれ値一つを混ぜたソート列に対しての性能についての考察を行う．quick sort は移動する値の要素が激減することから, このようなほぼソートされた列に対しての相性が良い．そのため実行時間は大きく加速している.bitonic sort は, アルゴリズム自体の相性は良くない物の, ある程度粒度が小さくなるまで行なった初期分割の末端においてで quick sort を行っているため, その部分の実行速度の加速の分, 実行時間は短縮されている. また, Cilk sort も初期分割の quick sort での加速の他, マージに際しソートが行われている場合条件分岐を行わずにマージを行えるため, 短縮されている．saw sort は swap ベースのソートであるため, flag の位置が両端に合った場合その部分のマージ操作をデータの読み書き含め一切飛ばすことが可能であり, このような既にソートされている列に対しては圧倒的な性能を持つ. データの移動が減るため, メモリバンド幅やキャッシュミスの影響も低減することが可能になると言えるため, 並列実行への影響も良いと言える．実行時間が短すぎるため, 16M の要素数においてはスケーラビリティの正確な測定が行えないが, 256M の要素数の場合 24 コアで逐次実行の 15 倍近くとランダム列に対する性能よりもスケーラビリティが上昇している.

そして, 幅 100 の一様乱数を加えたソート列に対する各ソートの性能に注目する. 粒子法などに代表される問題で, 頻出するこのような全体的にはソートがされているという配列に対しても移動すべき値の数が減るため, quick sort は相性が良い. 一方で bitonic sort と Cilk sort はこのような列に対してはランダム列とほぼ同程度の操作が必要となるため, 初期分割の quick sort 分の加速に留まる. スケーラビリティをグラフにしたものを要素数 16M と 256M についてそれぞれ図 7.3 と図 7.4 に示す. 要素数 16M において, quick sort はボトルネックとなる初回の操作の時間が短縮されたことにより, スケーラビリティが上昇している他, saw sort は実行時間は短縮されていながらも, ランダム列に対する場合と比較して遜色ない並列性能を出すことが出来ている. 要素数 256M においても, 上記の理由から quick sort はスケーラビリティが上昇する他, 計算量の削減から遅延の影響が減った saw sort もスケーラビリティが向上している. 残り二つのソートアルゴリズムはランダム列に対しての性能と大差ない実行時間となった. これらの結果から, saw sort はこのような型のほぼソートされた列に対しても適応力があることが示されたと言える.

16M 要素に対してのメモリ使用量の比較を図 7.5 に, 256M 要素に対してのメモリ使用量の比較を図 7.6 に示す. なお, 配列の並びによってメモリの使用量が変化することはほぼなかったため, ランダム列に対しての実行のメモリ使用量を参照した. グラフからは少し読み取りづらいが, quick sort と saw sort のメモリ使用量は bitonic sort とほぼ変化しない. 図 7.5 において quick sort に比べ bitonic sort と saw sort のメモリ使用量がやや多いのは, 再帰呼び出しの回数が多いためと考えられる. また, 全体としてグラフが右肩上がりなのはコア数の増加によってタスク並列ライブラリ MassiveThreads が使用するメモリの量が増加しているためである. これらの増分は定数であるため, データサイズが大きくなった図 7.6 ではこれらの影響が図 7.5 よりも相対的に小さくなることが確認出来る.

ベースが原始的な merge sort である Cilk sort は一時配列として配列の長さと同じだけのメモリを必要するため, 他の swap ベースのソートの倍のメモリを消費することが分かる.saw sort もマージの際の末端でメモリ領域を使用して merge sort を行っており, 先に述べたとおり本実験では $2^{15}$ からこの切り替えを行うが, オーダーにほぼ影響しない程度のメモリ使用でソートを完了していることが分かる. データ量が大きくなればなるほど, Cilk sort や Sample sort 等の空間計算量 $O(n)$ のソートアルゴリズムと比較して $O(\log n)$ で済むソートアルゴリズムの優位性が上がるため, 提案手法の持つ強みが確認出来たと言える.

表 7.2. ランダム列に対する各ソートの性能

| コア数 | bitonic sort | | quick sort | | cilk sort | | saw sort | |
|---|---|---|---|---|---|---|---|---|
| | 16M[sec] | 256M[sec] | 16M[sec] | 256M[sec] | 16M[sec] | 256M[sec] | 16M[sec] | 256M[sec] |
| 1 | 7.382 | 188.452 | 2.555 | 48.642 | 2.786 | 55.041 | 2.895 | 60.738 |
| 2 | 3.771 | 96.799 | 1.334 | 25.199 | 1.407 | 27.851 | 1.464 | 31.084 |
| 4 | 1.929 | 50.111 | 0.725 | 13.898 | 0.701 | 14.113 | 0.735 | 16.058 |
| 6 | 1.342 | 34.857 | 0.529 | 10.236 | 0.482 | 9.660 | 0.503 | 11.284 |
| 8 | 1.042 | 26.991 | 0.442 | 8.470 | 0.368 | 7.458 | 0.398 | 9.028 |
| 12 | 0.756 | 19.702 | 0.364 | 6.882 | 0.262 | 5.373 | 0.278 | 7.021 |
| 16 | 0.607 | 16.230 | 0.337 | 6.363 | 0.217 | 4.771 | 0.220 | 6.339 |
| 24 | 0.466 | 13.092 | 0.333 | 6.006 | 0.185 | 4.606 | 0.174 | 6.087 |

表 7.3. はずれ値一つのソート列に対する各ソートの性能

| コア数 | bitonic sort | | quick sort | | cilk sort | | saw sort | |
|---|---|---|---|---|---|---|---|---|
| | 16M[sec] | 256M[sec] | 16M[sec] | 256M[sec] | 16M[sec] | 256M[sec] | 16M[sec] | 256M[sec] |
| 1 | 5.401 | 143.262 | 1.524 | 27.389 | 0.745 | 16.511 | 0.408 | 6.382 |
| 2 | 2.763 | 73.799 | 0.792 | 14.098 | 0.411 | 9.157 | 0.211 | 3.272 |
| 4 | 1.435 | 38.650 | 0.419 | 7.456 | 0.251 | 5.762 | 0.107 | 1.683 |
| 6 | 0.996 | 26.957 | 0.302 | 5.238 | 0.215 | 5.101 | 0.077 | 1.157 |
| 8 | 0.798 | 21.459 | 0.241 | 4.202 | 0.189 | 4.823 | 0.061 | 0.918 |
| 12 | 0.588 | 15.945 | 0.184 | 3.167 | 0.192 | 4.974 | 0.044 | 0.676 |
| 16 | 0.485 | 13.194 | 0.154 | 2.687 | 0.204 | 5.335 | 0.038 | 0.554 |
| 24 | 0.388 | 11.407 | 0.123 | 2.219 | 0.221 | 5.957 | 0.043 | 0.450 |

表 7.4. 幅 100 の一様乱数を加えたソート列に対する各ソートの性能

| コア数 | bitonic sort | | quick sort | | cilk sort | | saw sort | |
|---|---|---|---|---|---|---|---|---|
| | 16M[sec] | 256M[sec] | 16M[sec] | 256M[sec] | 16M[sec] | 256M[sec] | 16M[sec] | 256M[sec] |
| 1 | 6.670 | 165.410 | 2.264 | 37.816 | 2.695 | 46.841 | 2.449 | 43.671 |
| 2 | 3.404 | 84.998 | 1.161 | 19.539 | 1.371 | 24.134 | 1.235 | 22.126 |
| 4 | 1.761 | 44.091 | 0.603 | 10.152 | 0.709 | 12.568 | 0.625 | 11.432 |
| 6 | 1.227 | 30.813 | 0.421 | 7.093 | 0.512 | 8.964 | 0.431 | 7.779 |
| 8 | 0.966 | 23.838 | 0.330 | 5.532 | 0.401 | 7.183 | 0.328 | 6.163 |
| 12 | 0.695 | 17.761 | 0.241 | 4.129 | 0.298 | 5.610 | 0.232 | 4.347 |
| 16 | 0.581 | 14.641 | 0.200 | 3.392 | 0.261 | 4.916 | 0.183 | 3.745 |
| 24 | 0.444 | 12.274 | 0.154 | 2.715 | 0.209 | 4.750 | 0.143 | 3.097 |

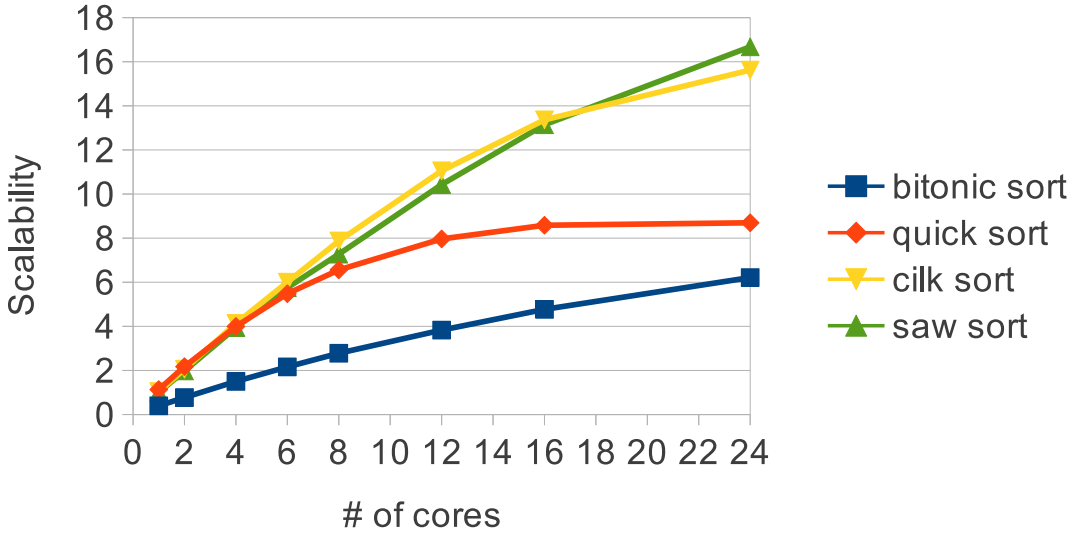


図 7.1. 要素数 16M のランダム列に対する性能

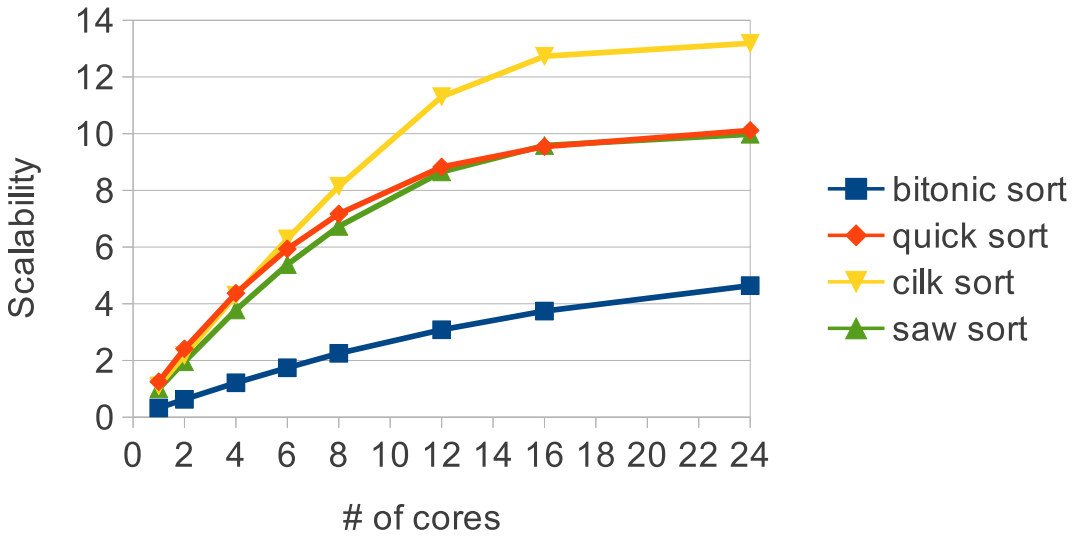


図 7.2. 要素数 256M のランダム列に対する性能

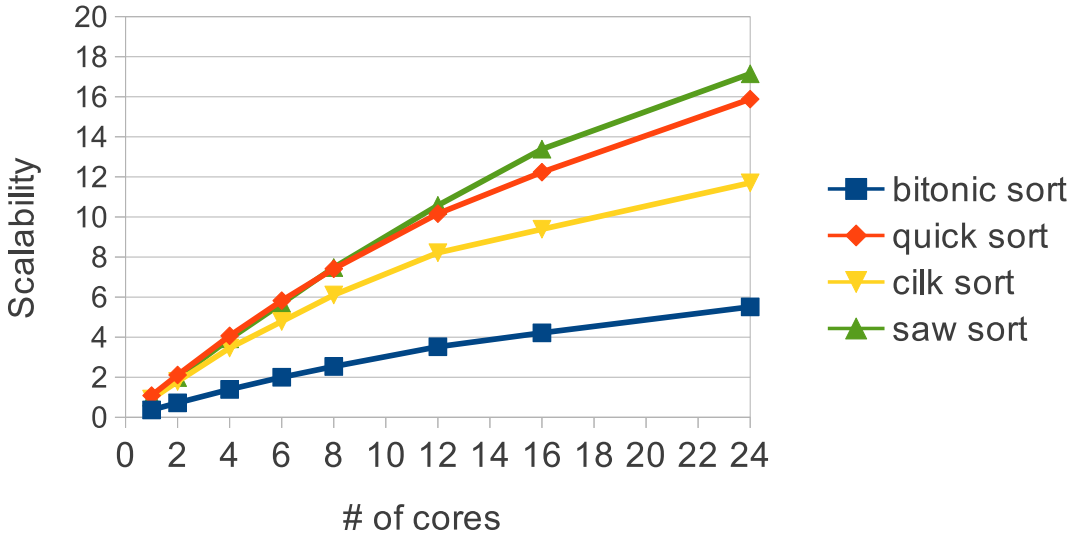


図 7.3. 要素数 16M のソート列に幅 100 の一様乱数を加えた列に対する性能

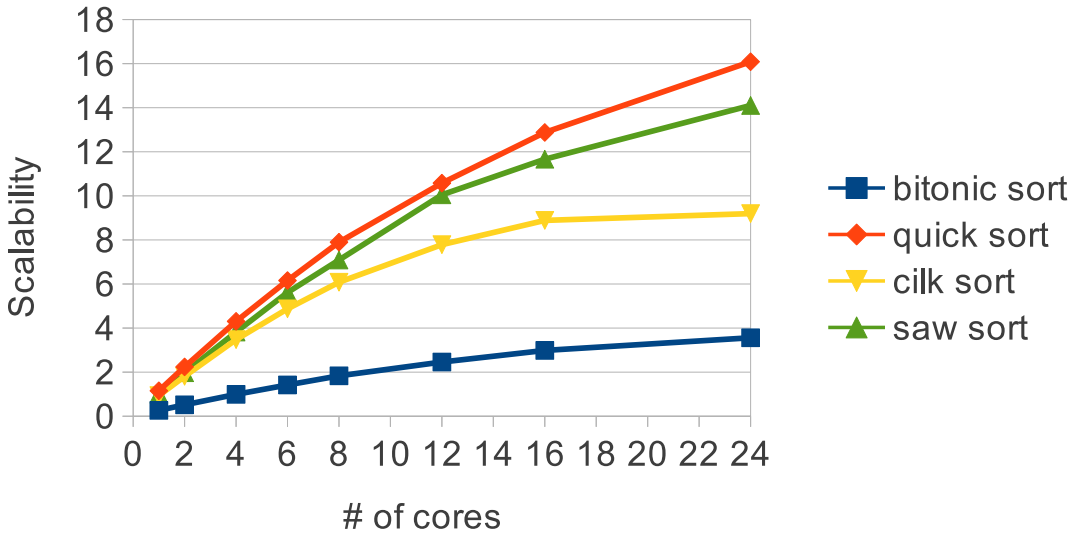


図 7.4. 要素数 256M のソート列に幅 100 の一様乱数を加えた列に対する性能

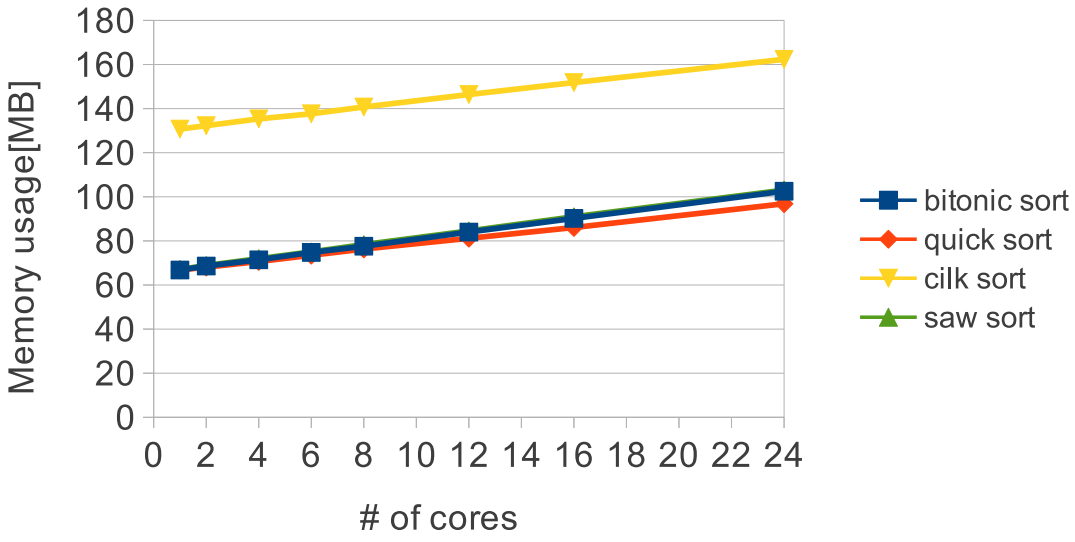


図 7.5. 要素数 16M のランダム列に対するメモリ使用量の比較

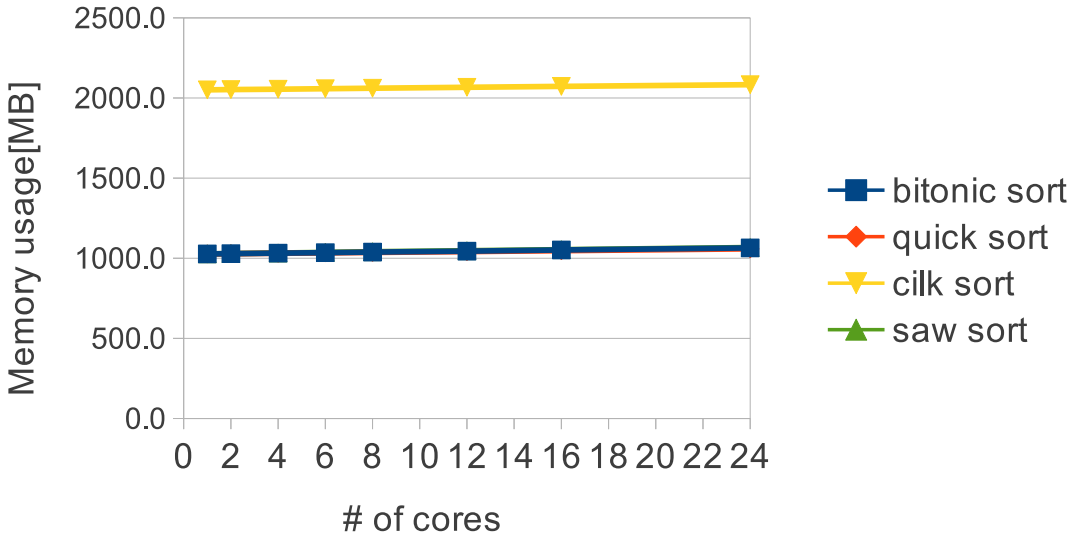


図 7.6. 要素数 256M のランダム列に対するメモリ使用量の比較

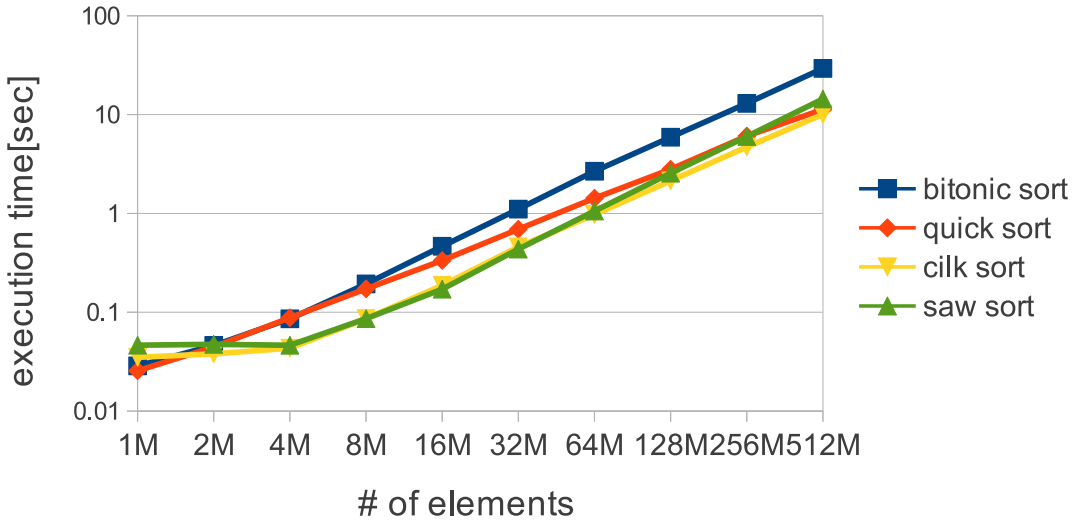


図 7.7. データサイズを変化させた時の並列実行時間

## 7.4　データサイズに対する性能変化

データサイズを変化させた時の実行時間の変化を測定した. 実験ではそれぞれのソートについて 24 コアで並列実行を行い, 要素数を 1M から 512M まで増やしながら実行時間を測定した. 結果を図 7.7 に示す. 全体として要素数が増えると共に実行時間も当然増加するが, bitonic sort は実行時間の増加割合が特に大きい. これは, bitonic sort の $O(n \log^2 n)$ という逐次計算量の多さが要素数 $n$ の増加の影響を特に受けているためと考えられる.bitonic sort をベースとしている saw sort も, 削減をかけているものの増分が計算量 $O(n \log n)$ の quick sort, Cilk sort と比較して多く, より大きなデータを扱う事を考慮する場合, 何らかの対処が必要であると考えられる. 改善策としては, $n$ が大きくなると, マージの打ち切りである $m$ の値を総メモリ使用量に影響のない範囲で大きくすることが可能であるため, $m$ を決定的にせず入力サイズに応じて自動で変化させる事が挙げられる.

## 7.5　既存研究との比較

提案手法の性能を近年の高速並列ソートの研究と比較した. 比較対象は近年の共有メモリ内の高速並列ソートである odd-even merge sort[14] とした. 実装が公開されていないため, 比較対象のデータは論文内の物を用いることとした. 対象の実験環境は Core clock speed3.22GHz を 4 コア積んだマシンであり, 並列化には Pthreads を用いている. 結果を表 7.5 に示す.

表 7.5 から, saw sort は比較対象と比べ逐次で約 8 倍時間がかかっていることが分かる. しかし, 比較対象は SIMD を有効利用しており, SIMD を使用しない場合は括弧内の値となり, 差は 2.5 倍ほどとなる. これに関して, 現在 saw sort のマージの末端で行っている primitive なマージをこの

比較対象のような SIMD を利用する形式にすることで改善を行える. この末端のマージが実行時間において占める割合は多く, 比較対象ほどでは無くとも, 実行時間の短縮を見込むことが可能である.

さらに, 実行コアの性能差が挙げられる. 今回実験に用いたコア単体の周波数は表 7.1 で示したように 2.00GHz であるが, 比較対象は 3.22GHz と, およそ 1.6 倍となっている. この性能差を加えて考慮することで, 差分は 1.5~2 倍程度であると言える.

また, 速度では多少遅くなっているが, 空間計算量に関しては比較対象は merge sort ベースであるため $O(n)$ 必要となるが, 提案手法は $O(\log n + m * p)$ であることから省メモリであり, 比較対象は pthreads を用いた実験環境に特化した並列実装をしているのに対し, 提案手法はタスク並列を採用しているため記述が比較的容易であり, 環境の変化に対する適応性が高いという速度以外の場所での利点があるため, 全てにおいて劣っているということは無いと考えられる.

## 7.6　スチール方針による変化

提案手法である共有キャッシュを考慮したワークスチール方針をスチーリング関数としてプログラムに実装し, スチーリング方針を変更したことによる実行時間やキャッシュミスの変化を調査した. 比較対象は MassiveThreads に実装されている基本的なランダムワークスチーリングである.

6 章で述べたように, 広範囲のデータを扱う必要がある問題をタスク並列で記述する場合, ランダムスチーリングによって共有キャッシュの奪い合いによる遅延が発生する恐れがある. これを提案手法のワークスチーリングを用いることでの改善を図った.

まず予備実験として, L3 キャッシュに収まる配列を 4 つ作り, それらを読み込むだけのタスクを大量に作るだけのプログラムを使い, それらをランダムスチーリングで処理する場合と提案手法によって各配列ごとに属するタスクを別のソケットで並列処理を行った物を比較し, 提案手法が大幅に L3 キャッシュミスと実行時間が削減することを確認した.

これを受けて saw sort に対しての比較実験を行なった. なお, 提案手法のワークスチール方針を用いる時, 6 章で今回採用した手法のため, タスクの作り方がやや変わる. このタスクの作り方の変更自体が, 実験環境に即した物にするための行為であるため, 純粋にスチーリング方針の変更による成果を見るため, 元の実装・タスクの作り方のみを変えたもの・提案スチール方針を採用したものの 3 種類で実験を行なった. 前者二つはランダムワークスチーリングを採用している. 入力は要素数 16M と 256M のランダム列とし, 元の実装の逐次実行の結果を 1 としてスケーラビリティを測定した. 実験結果をそれぞれ図 7.8 と図 7.9 に示す.

表 7.5. odd-even merge sort[14] との比較

| コア数 | saw sort | | comparison | |
|---|---|---|---|---|
| | 16M[sec] | 256M[sec] | 16M[sec] | 256M[sec] |
| 1 | 2.895 | 60.738 | 0.3739(1.2636) | 7.7739(23.5382) |
| 2 | 1.464 | 31.084 | 0.2042 | 4.3703 |
| 4 | 0.735 | 16.058 | 0.1170 | 2.4521 |

両方のグラフとも, コア数が少ない時はほぼ変化がないが, コア数 16 付近から, 提案手法のスケーラビリティが向上していることが分かる. これは先程述べたランダムスチーリングが共有キャッシュミスを誘発するという現象がコア数が多いほど発生しやすいためであり, 提案手法のスチーリング方針によってそのようなキャッシュミスの発生を削減出来たためと考えられる. これを検証するため, perf コマンドによって 24 コア並列実行時のキャッシュミス数の測定を行なった. 値は perf stat コマンドにおける cache-misses の値を採用した. 結果を表 7.6 に示す.

提案スチーリング方針によって, ランダムスチーリングの場合と比較して 20%近くのキャッシュミスを削減している事が読み取れる. またこのキャッシュミスの中の一定の割合は, 元々確実に起こるキャッシュミスであるため, 実際にランダムスチーリングによって発生した余計なミスの削減率はさらに高いと考えられる.

検証を深めるため, 要素数を 1M から 512M 変化させ 24 並列での実行時間を測定し, 各ソートとの比較を行なった. 結果を表 7.7 に示す. rs-saw sort はランダムスチールを用いた saw sort であり, cs-saw sort は提案スチーリングを用いた saw sort の結果を示している. 全ての要素数で提案手法を用いた物がランダムスチールを用いた物よりも実行時間が短くなっている. 表 7.7 をグラフにしたものが図 7.10 である. 図 7.10 要素数が少ない部分での実行時間が改善されていることが分かる. これは実行時間が極端に少ないため, 周回スチールにより一回のスチールの質が向上した影響のためスチール回数が削減されたことによるオーバーヘッドの減少が原因と推測される.

さらに検証を深めるため, MassiveThreads 付属の性能解析ソフトウェア DAG_RECORDER を用いて, 実行時のスチーリングによるタスクの動きを可視化した. 要素数 256M に対する, 24 コア並列実行時のタスクの動きをランダムスチーリングと提案手法のスチーリングについてそれぞれ図 7.11 と図 7.12 に示す.

まず図の読み取り方の説明から行う. グラフ上部にある, total steal が実行時に行われたワークスチールの総計数であり, s-local がその中で同じソケット内でのスチールであった場合である. 図の x 軸, y 軸がそれぞれ「タスクを盗んだコア」「タスクを盗まれたコア」にあたり, その回数が多いほどグラフの色が青から赤に近くなる物となっていり, 最も赤い部分がグラフ上部の max の数である. また, 実験環境は 24 コア 4 ソケットであるため, 6 個分毎に 1 ソケットとなる. タスク並列処理においては, 基本的に生成されたタスクのほとんどはそのままスチールされることなくそのまま実行されるため, 対角線の (=同じコア内で実行された) タスクの数は他の場所に比べ非常に多くなる. そのため max の値は対角線以外で最も大きな値を表示している.

図 7.11 と図 7.12 を比較していくと, まず total の steal 数はほぼ同数となっているが, そのうち socket-local である割合が提案手法は大幅に上昇している. ソートのような広範囲のデータを分割統治法で行う場合, 遠くのコア, ソケット外からのワークスチールはほぼ共有キャッシュミスを生じさせるため, 提案手法がタスクの動きを見ても共有キャッシュミスを削減していることが確認されたと言える.

表 7.6. 24 並列実行時のキャッシュミス数

| 要素数 | normal | task change | steal change |
|---|---|---|---|
| 16M | $1.0 * 10^7$ | $1.0 * 10^7$ | $7.9 * 10^6$ |
| 256M | $3.5 * 10^8$ | $3.6 * 10^8$ | $3.0 * 10^8$ |

さて, 今回の実験では結果としてスケーラビリティが改善したものの, 提案手法はスチーリングの質を良くするため, タスクの生成数に制約をかけた上で, 周回スチーリングを行うなど時間をかけているため, それらが実行時間に悪影響を及ぼす可能性が無いとは言えない. そこで DAG_RECORDER によって実行時間の内訳の解析を行なった. 解析結果をランダムスチーリングと提案手法のスチーリングについてそれぞれ図 7.13 と図 7.14 に示す.

グラフはそれぞれ横軸がタイムステップとなっており, 赤がワークスチールなどではなく実際に実行したいタスクを行っているコアの数である. 今回は 24 並列であるため赤の最も高い場所は 24 にあたる. これが実行したコア数よりも低い部分は並列度が出ていないと言える. そして青は実行可能だが, 実行されていない, スチールが可能である実行待ちのタスクの量を示す. このようなタスクが潤沢な場合, 動的負荷分散が上手く行くため並列性能が高くなると言える. まず図 7.13 に注目すると, 初期にタスクが大量に生成されるためすぐに実働コア数は 24 に達し, タスク数が僅かになった終盤のみやや並列度が下がるものの十分な並列実行は出来ていると考えられる. 一方で図 7.14 から提案手法は実行待ちタスクが無くなり並列度が下がってしまう場面が序盤から散見され, 特に中盤以降は 24 並列実行出来ている場面が殆ど存在しない.

より詳しい分析のため, DAG_RECORDER を用いて実行結果を値として取得した物が表 7.8 である. 表の work は実際に実行したいタスクを行なったクロック数の総計を示しており, delay は実行可能なタスクがあったにも関わらず, 並列度が出ていない部分のクロック数の総計である. また, no work は実行可能なタスクが存在しなかったため, 並列度が出せなかった部分のクロック数の総計である. 表 7.8 から, 256M では提案手法のスチール方針は delay と no work がランダムスチールと比較して大きく増加することが分かる. これは, 提案手法では初期の分割移行は共有キャッシュに収まるまでタスクの生成を制限するためその影響でタスクがランダムスチールよりも少なくなる要素数 256M において並列度が下がったためと考えられる. 単純にタスクが無い状況が増える他, Socket-aware Steal により, 他のソケットにスチール可能なタスクがある場合でもあるソケットのメンバーコアがスチール可能なタスクが同ソケットにないためタスクを実行出来ない場合があるため, no work と delay 両方が増加してしまう.

また, work に注目すると要素数 16M と 256M の双方で提案手法の方が少なくなっていることが分かる. これはランダムスチーリング側が実行全体に対して提案手法の場合より時間がかかっているということを意味し, 即ちそれが共有キャッシュミスによる遅延であると言え, その影響がいかに大きいかを示していると言える. また, 提案手法はグラフの形状は並列実行としてはややお粗末であるが, これは逆に改善の余地がまだ十分に残されていると捉えることが可能である. タスク数を減らしすぎない様な実装で Socket-aware と Cache-aware を考慮したタスクの実行を行えるような改善を施すことが出来れば, 並列度が上昇し, 現状よりもさらに実行時間の短縮が見込めると言える.

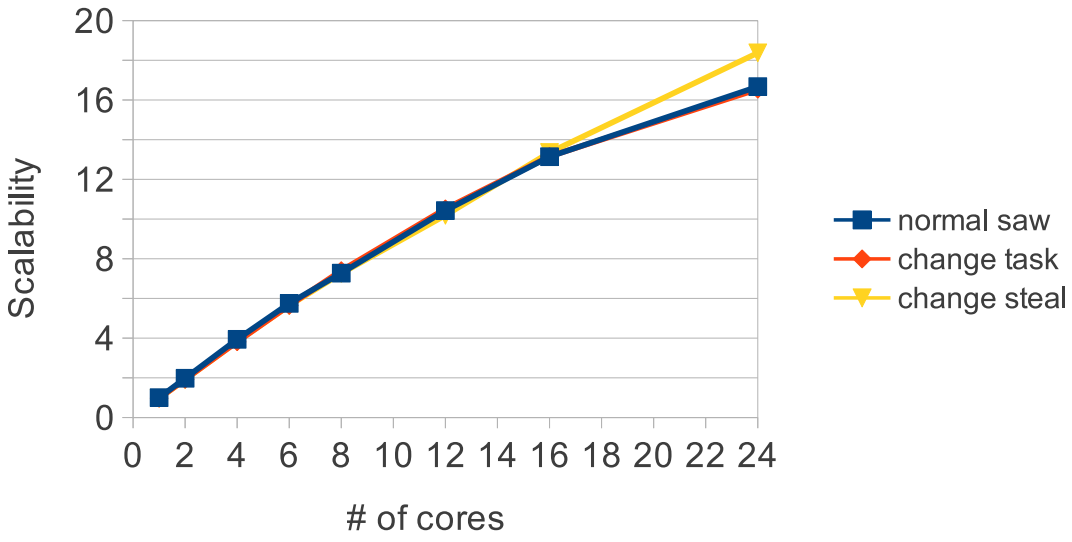


図 7.8. 要素数 16M のランダム列に対する性能

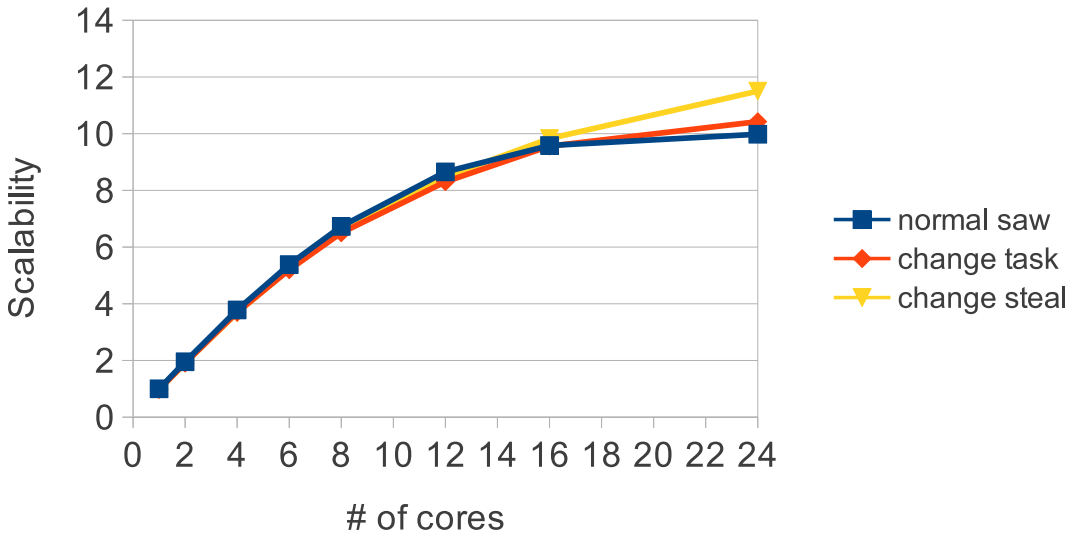


図 7.9. 要素数 256M のランダム列に対する性能

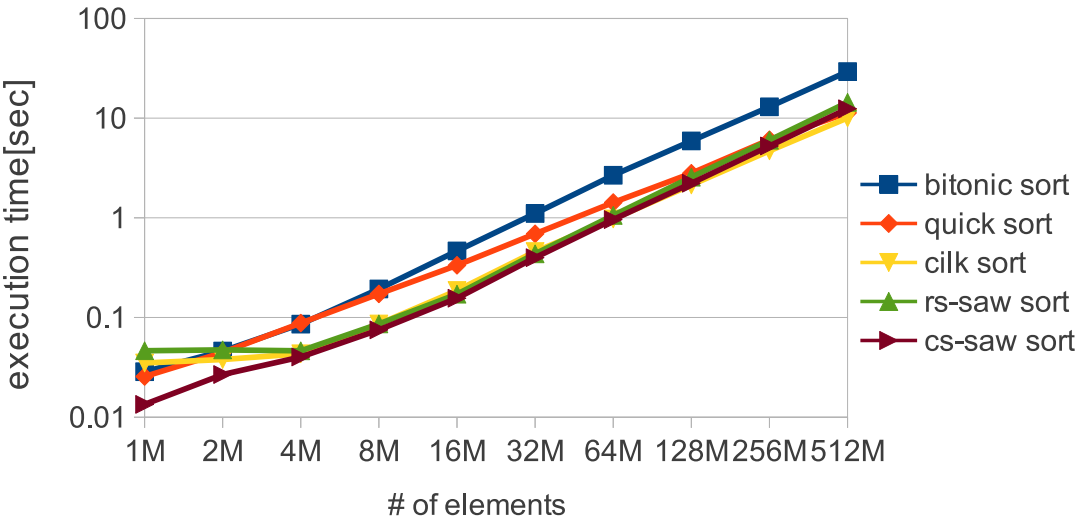


図 7.10. 要素数を変化させた時の並列実行時間

表 7.7. 要素数を変化させた時の並列実行時間

| 要素数 | bitonic sort | quick sort | cilk sort | rs-saw sort | cs-saw sort |
|---|---|---|---|---|---|
| 1M | 0.029 | 0.026 | 0.035 | 0.046 | 0.013 |
| 2M | 0.046 | 0.045 | 0.038 | 0.047 | 0.027 |
| 4M | 0.086 | 0.088 | 0.043 | 0.046 | 0.040 |
| 8M | 0.193 | 0.172 | 0.086 | 0.086 | 0.075 |
| 16M | 0.465 | 0.334 | 0.187 | 0.171 | 0.156 |
| 32M | 1.106 | 0.691 | 0.459 | 0.435 | 0.398 |
| 64M | 2.669 | 1.424 | 0.974 | 1.057 | 0.959 |
| 128M | 5.907 | 2.797 | 2.135 | 2.555 | 2.222 |
| 256M | 12.984 | 6.066 | 4.698 | 6.003 | 5.257 |
| 512M | 29.300 | 11.287 | 10.050 | 14.395 | 12.429 |

表 7.8. スチーリング手法による実行結果の変化

| 方針 | random steal | | proposal steal | |
|---|---|---|---|---|
| 要素数 | 16M[clock] | 256M[clock] | 16M[clock] | 256M[clock] |
| work | $7.4 * 10^9$ | $2.7 * 10^{11}$ | $6.4 * 10^9$ | $1.8 * 10^{11}$ |
| delay | $5.9 * 10^8$ | $3.5 * 10^9$ | $5.9 * 10^8$ | $3.0 * 10^{10}$ |
| no work | $9.2 * 10^8$ | $1.3 * 10^{10}$ | $9.2 * 10^8$ | $3.8 * 10^{10}$ |

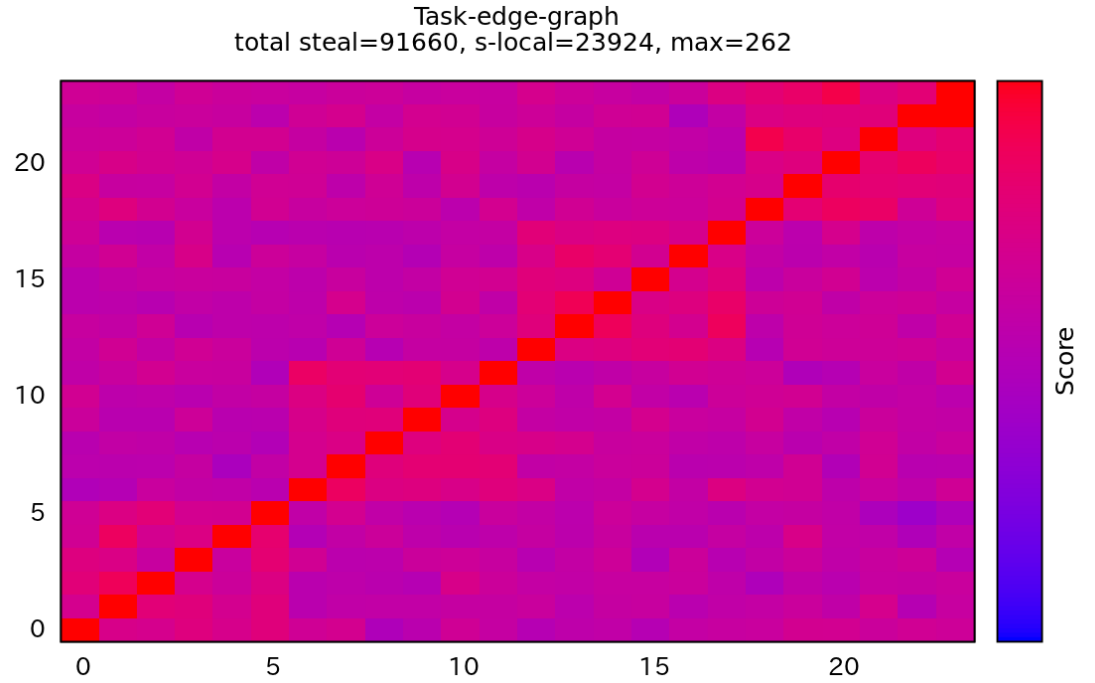


図 7.11. ランダムスチーリングによるタスクの動作

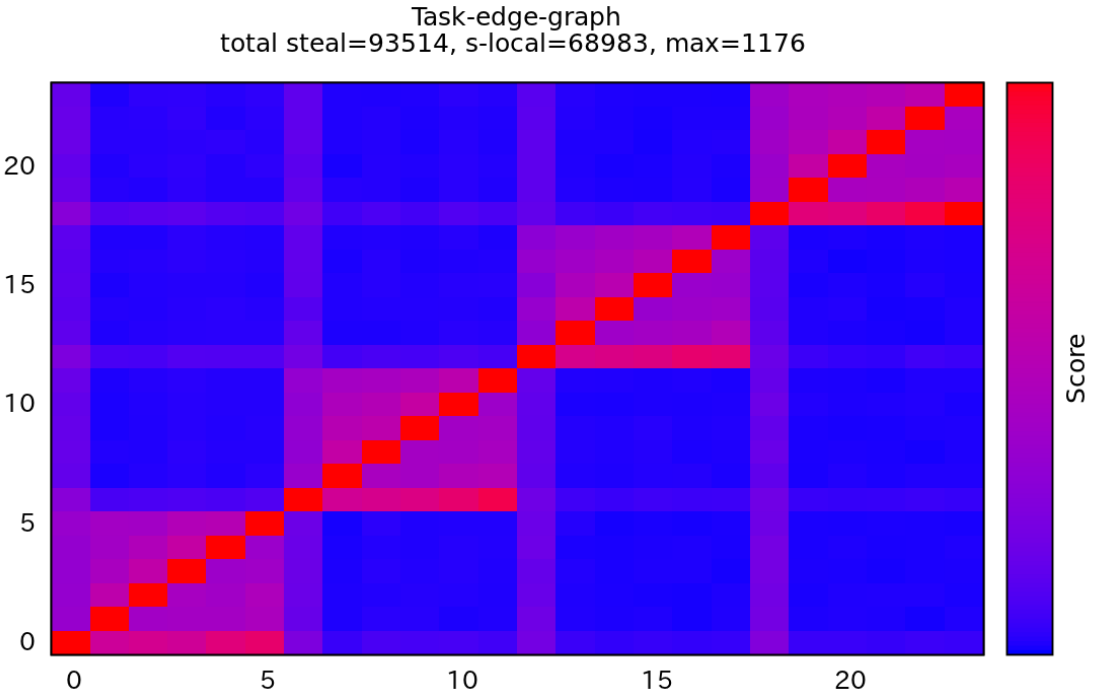


図 7.12. 提案手法のスチーリングによるタスクの動作

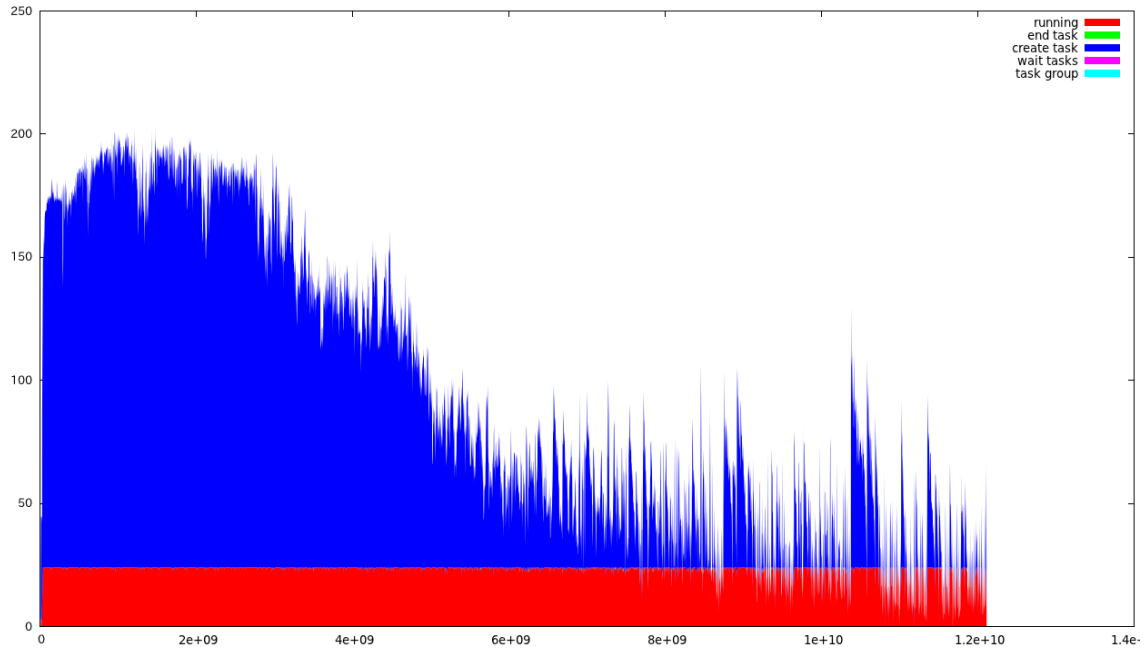


図 7.13. ランダムスチーリングを採用した場合の実行解析

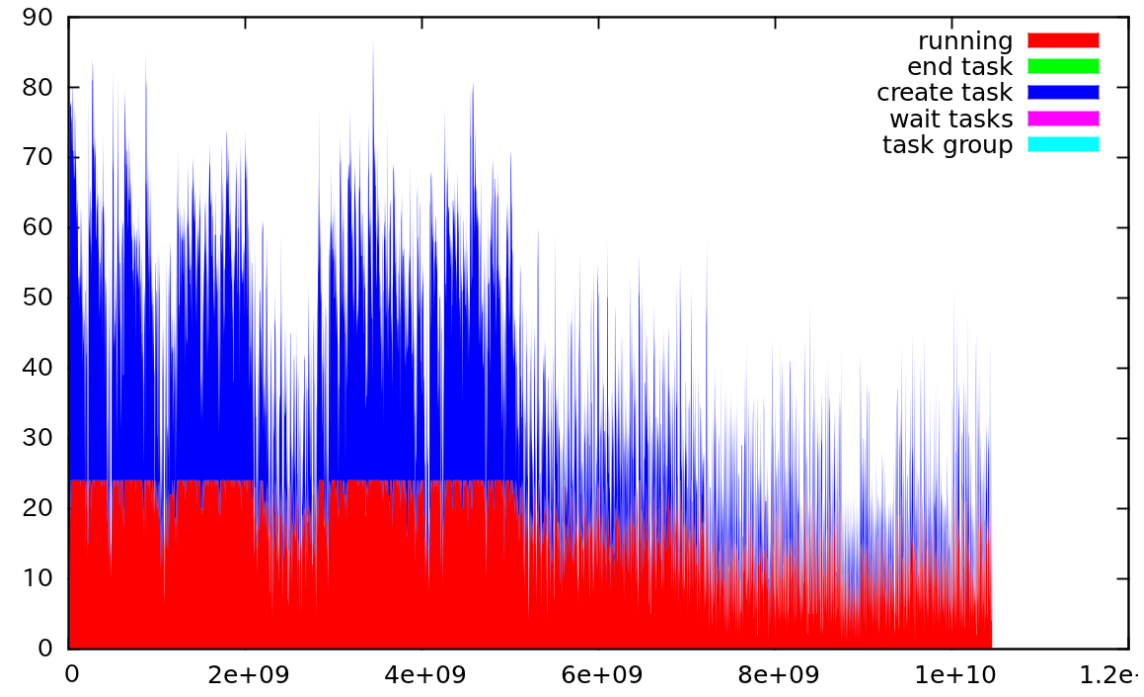


図 7.14. 提案手法のスチーリングを採用した場合の実行解析

# 第8章　結論

## 8.1　まとめ

本研究では, タスク並列処理に適したソートアルゴリズムとして, 分割統治法を用いて記述する省メモリな並列ソートアルゴリズム鋸ソートを提案した.

提案手法は省メモリかつスケーラブルな bitonic sort をベースとし, 実用性を高めるため, 同ソートが苦手とするほぼソートされている列に対するソーティングの速度の改善と逐次計算量の削減を行った. 具体的には bitonic sort が用いる bitonic 列の代わりに二つのソート列の連続と定義する鋸列を提案し, これを再帰的に生成することでソート列に対する適正を上げ, 十分なタスク数が確保出来た場合にいくらかのメモリ領域を使って計算量の少ない merge sort を導入することで, 並列性能と空間計算量に大きな影響を与えること無く逐次計算量の削減を図った.

また, ソートアルゴリズムのような広範囲のデータを扱う問題をタスク並列処理系を用いて解く際に, 通常のランダムスチーリングを行うと共有キャッシュミスが大量に発生してしまう恐れがあるという問題に着目し, タスクのソケット間の割り振りを無駄なく行い, ソケット内でのタスク処理順をキャッシュサイズを考慮した物にするワークスチーリング方針を提案した.

実験では, 共有キャッシュに収まる範囲のデータに対し, 鋸ソートは省メモリ性を保ちつつ高速並列ソートである Cilk sort 以上の結果を残した他, ほぼソートされた列に対しても quick sort 並の高い適正を持つことを確認した. また, 共有キャッシュを大幅に上回るデータに対してはランダムスチーリングを用いると並列性能が落ち込んでしまうが, 提案手法のスチーリング方針を用いることである程度の改善を図ることが出来, 共有キャッシュミスを大幅に削減することが出来た.

一方で絶対性能としては, 最近の高速並列ソートと比較するとまだ遅いという結果になったが, 省メモリである, タスク並列処理に適している事から入力データのばらつきに強い, などといった強みがあり, 実装にも改善の余地がまだあることからこれからの最適化が期待される.

## 8.2　今後の展望

提案手法は, 既存の高速並列ソートと比較して実行速度が遅かった. しかしながら理論計算量はそこまで劣らないため, 実装の最適化によって改善することは可能である. 具体的には, 以下のような改善が考えられる.

- 末端の実装の最適化
  鋸ソートでは, 逐次計算量を削減するため, マージの際にタスク数が十分である末端においては逐次計算量 $O(n \log n)$ の merge sort を行っていたが, 今回の実装は非常に primitive な merge sort を行っていた. この末端のマージは全体の実行時間の中でも結構な割合を占めるため, これを最適化することで実行時間の大幅な改善が期待される. 逐次 merge sort の SIMD

などを利用した高速化最適化は既に研究が進んでおり, それらの実装を取り入れるなどして高速化を図る必要がある.

- メモリバンド幅関連の問題の解消
  並列ソートアルゴリズムにおいて, コア数が増加すると必ず発現する問題として, メモリバンド幅の上限に律速されるといった物があり, 今回も 24 コアでの実行時はキャッシュミス以外にもこちらの影響が少なからずあったと考えられる. これを解決するための手法としては提案手法が merge sort ベースであるため, 2-way でなく n-way でマージを行う multi-way merge の導入が考えられる. 末端に関しては既存の研究の実装を取り入れられる他, 鋸マージも $2^n$-way merge が理論上可能である.

また, 提案ワークスチーリング方針は, 今回の実装では共有キャッシュミスと実行時間の削減には成功したが, 一方で並列度が下がってしまう箇所が出現してしまった. これを解消するため, タスクを十分に作った上で実行順に指向性を持たせる必要があり, 6.3.1 節で述べた手法のうち上二つの手法をとるといった改善が考えられる. その上で, ソートに限らず広範囲のデータを参照する様々なタスク並列プログラムで検証を行い, ある程度の汎用性を持たせていくことが望ましい.

# 謝辞

まず, 本研究を行うにあたり, 研究の方針実装その他もろもろ, 絶大な指導をして下さった田浦健次朗准教授(ティーチャー)に深く感謝します. 知識量などのポテンシャルが低い私を, 学会や行き詰まった時など, 様々な場面で懇切に導いて下さり, 迷走を重ねた研究も形にする事が出来ました. 先生(ティーチャー)が下さった数々の至言は今や私の一割程度を為していると思います.

D3 の中島潤先輩には, MassiveThreads 開発者として, 研究関連で悩んでいたときに快く相談に乗って頂き, また冒険者としての他愛もない会話もして頂きました. 困ったらいつも MassiveThreads[24] のせいにしていましたが, 大抵は私のミスであったことに関して, この場を借りて謝罪させて頂きます. 社会人 D の井上拓さん, D2 の秋山茂樹先輩, 同期の中谷君には技術的なアドバイスを幾度となく頂き, 研究の質を向上させることが出来ました. 同期の林伸也(けんじ)君とは, 良き友人として, ある時には真剣に研究の議論をし, ある時には修士生活の辛さを分かち合い, またある時にはけん玉で互いを高め合いました. その常軌を逸した研究ぶりには幾度と無く閉口させられましたが, 触発される形で私も研究に取り組む気を持つことが出来, 林君のお陰で楽しい修士生活を送ることが出来ました. 同期の Amgaa 君には, ぶれない生き方とはどういうものかという事を身を持って見せて頂き, 今後の人生の参考になりました.M1 の菊地君には, 絵描きとして, M1 年の早水君と西岡君には, 他愛もない話などで良き話し相手になって頂いた他, 研究室運営でお世話になりました. また, 隣の研究室の M2 の板持君, M1 の亀甲(かめこう)君には修士の 2 年間, 幾度となく敢行された私の研究室荒らしに対しいつも快く接して頂きました. 特に板持君は私に生きる意味を与えてくれ, お陰でこの一年私の正気が保たれたと言っても過言ではありません.

その他にも研究生活を送る中で本当に多くの方々にお世話になりました. この場を借りて厚くお礼申し上げます. 最後に, これまで支えてくれた家族に感謝します. 本当にありがとうございました.

# 発表文献

[1] 中澤隆久，田浦健次朗．bitonic sort の高速な並列化．**並列／分散／協調処理に関するサマーワークショップ（*SWoPP2012*）**, 鳥取, 2012/8.

[2] 中澤隆久，田浦健次朗．省メモリでスケーラブルな merge sort アルゴリズム．**並列／分散／協調処理に関するサマーワークショップ（*SWoPP2013*）**, 北九州, 2013/8.

# 参考文献


[1] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Rnadall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, Vol. 30, No.8, pp. 207-216, 1995.

[2] Cray Inc. Chapel specification version 0.92. October 2012.

[3] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp.519 538, New York, NY, USA, 2005. ACM.

[4] Chuck Pheatt. Intelthreading building blocks. *J. Comput. Small Coll.*, Vol. 23, No. 4, pp. 298, 2008.

[5] Eduard Ayguad , Nawal Copty, Alejandro Dueran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Ernesto Su, Priya Unnikrishnan, and Guansong Zhang. A proposal for task parallelism in openmp. *In IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pp.1-12, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Trans. Parallel Distrib. Syst.*, Vol. 2, No. 3, pp. 264280, 1991.

[7] Hoare, C.A.R.: Quicksort. Computer Journal 5(4), 10-15 1962.

[8] K. E. Batcher.: "Sorting networks and their applications, " in Proc. AFIPS SJCC, vol. 32, Montvale, NJ: AFIPS Press, pp.307-314.1968.

[9] 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.

[10] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. J. ACM, 46(5):720748, September 1999.

[11] Message Passing Interface Forum. http://www.mpi-forum.org/.

[12] Xiaochun Ye. : High performance comparison-based sorting algorithm on many-core GPUs :Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on. 19-23 2010.

[13] 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.

[14] 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.

[15] C. Leiserson and A. Plaat.: Programming parallel applications in Cilk. SINEWS: SIAM News, 31, 1998.

[16] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, Vol. 46, No. 5, pp. 720 748, 1999.

[17] 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.

[18] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Theory of Computing Systems*, 2000.

[19] B.S. Chelebus.: A parallel bucket sort. Info. Process. Lett..27(2):57-61, Feburary 1988.

[20] S. J. Lee, M. Jeon, D. Kim, and A. Sohn.: Partitioned Parallel Radix Sort, J. Parallel Distr.Comput. (JPDC ), 62:656 668 2002.

[21] A. Sohn and Y. Kodama.: Load balanced parallel radix sort, in ‘ ‘ Proc. 12th ACM International Conference on Supercomputing, Melbourne, Australia, July 14-17, 1998.

[22] Cheng, D.R., Edelman, A., Gilbert, J.R., Shah, V.: A novel parallel sorting algo-rithm for contemporary architectures. Submitted to ALENEX 2006.

[23] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 297 308, Padua, Italy, June 1996.

[24] 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.

[25] Marsha J. Berger and Joseph E. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. Technical report, Stanford University, 1983.

[26] L. Arge, M. Goodrich, M. Nelson, and N. Sitchinava, “ Fundamental parallel algorithms for private-cache chip multiprocessors, ” in *Proc. 20th ACM SPAA, 2008*, pp. 197 206.

[27] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *IPDPS*, 2010.

[28] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999.

[29] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons. Scheduling irregular parallel computations on hierarchical caches. In *SPAA*, pages 355-366, 2011.

[30] Quan Chen, Minyi Guo, and Zhiyi Huang. CATS: Cache Aware Task-Stealing based on Online Profiling in Multi-socket Multi-core Architectures. In Proceedings of the 26th ACM international conference on Supercomputing, ICS ’12, pp. 163172, New York, NY, USA, 2012. ACM.

[31] Martin Wimmer, Daniel Cederman, Jesper Larsson Traff, and Philippas Tsigas. Work-stealing with Configurable Scheduling Strategies. In Proc. of PPoPP ’13, pp. 315316, 2013.

[32] 大筒裕之, 田浦健次朗. 電力効率を向上させるワークスチーリング手法. Bachelar’s Thesis. 東京大学工学部電子情報工学科 2013.

[33] Nadatur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, Pradeep Dubey, Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. SIGMOD ’10 Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. page 351-362 2010.