Vol.2020-HPC-175 No.4
2020/7/30

適応的分割法によるPARADISの高速化

尾城 拓真[1,a)]　宮崎 崇史[2,b)]　清水 伸幸[2,c)]　川島 英之[1,d)]

概要：メニーコアの有する高並列性を活用する並列 sort アルゴリズムに PARADIS がある．PARADIS はほかの並列 sort 法と比較しても高い並列性を示すことが論文に示されているが, 性能劣化を引き起こす挙動に関する解析は不十分である．本研究では PARADIS を再実装し，特定の場合において PARADIS の処理が逐次的に行われてしまうケースを明らかにし, そのケースに対応するための新しい PARADIS の手法, 適応的分割法を提案する．適応的分割法は, 従来の PARADIS では不可能であった Repair フェーズ内のバケット中の操作を並列化することを可能にした．適応的分割法を導入した PARADIS は実験の結果, 最大で 57%の性能向上を実現した．

Accelerate PARADIS by adaptive partitioning

Abstract: PARADIS is fast parallel in-place radix sort. In PARADIS paper, author show PARADIS sclable in terms of parallelism, but analysis of behavior causing performance degredation is insufficient. In this paper, we reimplement PARADIS, and clarify cases where PARADIS is performed sequentially in specific cases. Then, We propose a new PARADIS method adaptive partitioning method, to deal with the case. The adaptive partitioning method achive parallelizing operations in repair phase, which was not possible with original PARADIS. As a result of experiments, PARADIS which intoroduced the adaptive partitioning method, achieved a performance improvement of 57%.

1. はじめに

1.1 背景

計算機により扱われるデータ量は増加の一途を辿っている．すばる望遠鏡は毎晩 300GB の観測データを生成しており，気象予測シミュレーションでは 30 秒間で 2.5TB のデータが生成されることもある．今後 2030 年には毎日 100EB のデータが生成されるとの予測もある [4]．これらのデータから知識を発見する処理の一環として，sort は重要な役割を果たす．例えば宇宙空間におけるハローとマターの 2 点相関量計算を高速化したり [8]，ゲリラ豪雨予測におけるデータ同化処理を高速化するために，並列 sort が使われている．

半導体技術の進展に伴う CPU コア数の増加とメモリサイズの巨大化に伴い，並列 sort の重要度は一層高まっている．近年提案された手法としては，Satish 法 [10], PARADIS [3]，RADULS [7][6], Regions Sort [9] 等がある．これらの手法はいずれも radix sort である．この中で PARADIS と Regions sort は in-place radix sort を並列化している点に特徴がある．PARADIS は sort を行うにあたり 5 つのフェーズを必要とする．それらは図 1 に示されるように Build Histogram, Partition Array, Permute Data (Permute), Repair Data (Repair), Recurse Sort である．この全てのステージにおいて処理は並列化されている．そのため PARADIS はメニーコア環境を十分に活用した並列化処理が可能である．

1.2 研究課題

ある並列処理タスクについて，僅かな部分でも逐次化がされてしまえばタスク全体の性能が急激に劣化することが知られている [1]．PARADIS は基本的には全ステージでの並列化処理が可能なように設計されている．しかしながら，それはあくまでも基本的な状況に限られる．sort 実行の 1 つである repair フェーズにおいて，データがある種の分布をたまたま構成する場合には，並列化が困難になり逐次化してしまう．このことは端的に原論文の図 9 で

[1] 慶應義塾大学
[2] ヤフー株式会社
[a)] t.albicilla@gmail.com
[b)] takmiyaz@yahoo-corp.jp
[c)] nobushim@yahoo-corp.jp
[d)] river@sfc.keio.ac.jp

も示されている．このような現象を観察かつ解析するためには，スレッド数，データサイズなど様々な条件における PARADIS の挙動を解析することが必要である．この解析を行うためにはアルゴリズムレベルのみならずコードレベルでの実験結果ならびにデータアイテム変動経過の解析が求められる．ところが PARADIS は著者を含めて誰も実装コードを公開していないために，そのような解析を実施することは困難である．

1.3 貢献

上記の問題を解決するために，まず，我々は PARADIS をスクラッチから再実装した．PARADIS は American Flag Sort を並列化したものとみなせるため，まずは single thread における American Flag Sort の高速な実装として知られる kxsort[12] と同等なシングルスレッド版の PARADIS を実装した．これを踏まえ，マルチスレッドにおいて原論文の示す性能と比較して妥当な性能を示す PARADIS を実装した．本研究で使用したコードは github[*1] にて公開している．

PARADIS はスレッドにより分割された領域内に，swap する対象 (wrong elements) が集中するという状況が発生すると，性能劣化を示す現象を我々は発見した．この現象は PARADIS が安定性能を示せる範囲の限界を示すものである．PARADIS の論文では一例があげられていたもののこの状況に対する良い解法は原論文では述べられていない．この状況は PARADIS の repair フェーズの処理を逐次化させてしまうものであり並列処理の観点からは避けなければならない．

我々は最後にこの問題に対する解決策を提案する．従来の PARADIS アルゴリズムでは repair フェーズの実行中に swap 対象の位置が変動するために，逐次化せざるを得ない．我々はこれに対して wrong elements の位置が permute フェーズにおいて一定の規則に従い集約されることに注目した．それを利用することで並列化ができるケースが多くなるようにした．これを踏まえてデータの分布に応じて効果的に並列化が行えるケースであれば swap 処理の並列化を行い, 行えないケースであれば逐次的に swap 処理を行う．本拡張によって追加のメモリ領域を使用することはないので PARADIS の特徴である省メモリという特性を損なうことはない．この手法を適応的分割法と名付けた．適応的分割法により PARADIS の性能を最大で倍以上向上できることを実験を通して示す．

1.4 構成

本論文の構成は以下の通りである．2 章では PARADIS の基礎である American Flag Sort について述べる．3 章では PARADIS の詳細について述べる．4 章では提案手法を述べる．5 章では実験結果を述べる．最後に 6 章で結論を述べる．

[*1] https://github.com/albicilla/paradis_experiment

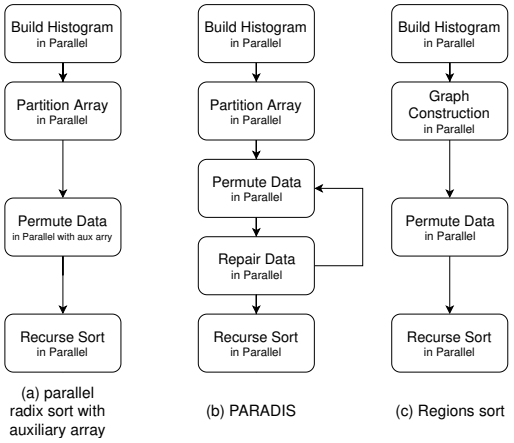

図 1 並列 radix sort アルゴリズムの比較 ([3] より引用したものを改変)

2. 準備

2.1 Parallel Radix Sort

Paralell radix sort として知られているものはそれぞれ LSD(Least Significant Digit) であるか MSD(Most Significant Digit) であるかどうかで分けることができる．LSD の場合は下の桁から counting sort を適用し, MSD の場合は上の桁から counting sort を適用していく．parallel radix sort には, 図 1 に示したように, radix sort に補助配列を用いる場合 (a) や PARADIS(b) のように補助配列を必要としないもの, Regions sort(c) のようにグラフを用いるものがある．(a) で代表的なものは RADULS, Satish 法であり, どちらも LSD radix sort である．(b)(c) はいずれも MSD radix sort である．

2.2 American Flag Sort

American Flag Sort[2] は Bentley らによって提案された radix sort の一種である．radix sort としての特徴としては MSD(most significant digit) radix sort であるため, 上位桁から counting sort を適用していく．また, in-place アルゴリズムであるため, sort 対象の配列に加えてそれと同等以上の配列を確保する必要がないため, 他の radix sort に比べ省メモリであるという特徴がある．American Flag Sort では, 並び替えの操作をバケットと呼ばれるそれぞれの領域にそこにあるべき key を入れ替えて行く操作を繰り返すことで行う．各バケットのサイズは sort 対象のデータに対してヒストグラムを作ることで求める．それぞれのバケットはそれぞれの key の基数に対応しており、正しい基数の領域への入れ替え操作が完了した時、同様の key の入れ替え操作を次のレベルで行う．正しい基数の領域への入れ替え操作が完了したかどうかは, 各バケットの入れ替えてあ

Algorithm 1 American flag sort($d[N]$,l)

```
1: //Build histogram
2: for i = start ... end do
3:    cnt[determineBucket(d[i], l))] + +
4: //calc ghi gti
5: // B is set of each bucket
6: for i ∈ B do
7:    gh_i = Σ_{j<i} cnt[j]
8:    gt_i = Σ_{j<=i} cnt[j]
9: //Permute
10: for i ∈ B do
11:    while gh_i < gt_i do
12:       v = d[gh_i]
13:       while determineBucket(v)! = i do
14:          swap(v, d[gh_{determineBucket[v]} + +])
15:       d[gh_i + +] = v
16: if l < L − 1 then
17:    for i ∈ B do
18:       AmericanFlagSort(d[M_i], l + 1)
```

Algorithm 2 PARADIS($d[N]$,l,P)

```
1: for p ∈ P do in parallel
2:    Build local histogram for each partition
3: end for
4: Synchronization
5: Build glocal Histgram
6: while Remain wrong element do
7:    for p ∈ P  do in parallel
8:       PARADIS_Permute(p)
9:    end for
10:   Synchronization
11:   for p ∈ P do in parallel
12:      for i ∈ B_p do
13:         PARADIS_Repair(i)
14:   end for
15:   Synchronization
16: if l < L − 1 then
17:    for i ∈ B do in parallel
18:       PARADIS(d[M_i], l + 1, P_i)
19:    end for
```

る key の先頭位置を示すインデックスが各バケットの終端位置を示すインデックスと等しくなったことで判断する．

American Flag Sort のアルゴリズムを Algorithm 1 に示す.([3] より引用したものを改変)

表 1 記号の意味

変数名	意味
N	データサイズ
L	必要な再帰の深さレベル
l	現在の再帰の深さレベル
P	プロセッサの集合
$d[]$	sort 対象データ
gh_i	バケット i, 先頭インデックス
gt_i	バケット i, 末尾インデックス
ph_i^p	バケット i，スレッド p，先頭インデックス
pt_i^p	バケット i, スレッド p, 末尾インデックス
M_i	i 番目の partition されたバケット

3. PARADIS

3.1 概略

PARADIS[3] は 2015 年に Cho らによって提案された並列 radix sort である．PARADIS は in-place な MSD radix sort を並列にした初めてのアルゴリズムである．PARADIS の基礎となっている radix sort は American Flag Sort である．すでに公開されている複数の実装において American Flag Sort は key 数が 64 以上の時において std::sort よりも高速に動作することが報告されている [2][5][12] ．また，省メモリであることから American Flag Sort の並列化は性能面と実用面の両面において効果が期待ができる．

PARADIS には 5 つのフェーズがあるが，それらは大きく 3 つのステージにおける並列化を American Flag Sort に対して行うとみなせる．それらのステージとフェーズの関係は次のようにまとめられる．

- ヒストグラム構築と分割．(Build Histogram and Partition)
- データ入れ替え操作．(Permute and Repair)
- 再帰による次レベルへの移行．(Recursion)

ヒストグラムの構築は元のデータを等分割し, 分割された領域のローカルなヒストグラムを並列に計算し最後にそれらを足し合わせることでグローバルなヒストグラムを得る．入れ替え操作の並列化はそのままではできないため, 二つの段階に分けて行う．それが Permute フェーズと Repair フェーズである．

入れ替え操作の具体的な並列化手法は次の節で述べる．

再帰による次のレベルの操作は再帰で呼び出す単位で並列化を行う．PARADIS のアルゴリズムを Algorithm 2 に示す．([3] より引用したものを改変)

3.2 Permute

American Flag Sort の入れ替え操作を並列に行うために PARADIS においては入れ替え操作を二つのフェーズに分けてそれを繰り返し適用していく．一つ目のフェーズが PARADIS_Permute であり, 二つ目のフェーズが PARADIS_Repair である．PARADIS_Permute においてはデータの配列をバケットごとに等分割することを考える. 図 3 は 2 スレッド時における操作を示している. 図中の黒矢印はヒストグラムによって計算された各バケットの境界を意味している. この情報をもとに PARADIS_Permute フェーズではバケットを分割する. これは図 3 における二段目に相当する．分割されたそれぞれのバケットの要素は同じ n 番目の分割されたバケットの要素のものと同じスレッドに属している. このため, 異なるバケット間であっても同じ n 番目の分割されたバケットの要素どうしであればデータの

Algorithm 3 PARADIS_Permute(p)

```
1: for i ∈ B do
2:   head = ph_i^p
3:   while head < pt_i^p do
4:     v = d[head]
5:     k = determineBucket(v)
6:     while k! = i && ph_k^p < pt_k^p do
7:       swap(v, d[ph_k^p + +])
8:       k = b(v)
9:     if k==i then
10:      d[head + +] = d[ph_i^p]
11:      d[ph_i^p + +] = v
12:    else
13:      d[head + +] = v
```

Algorithm 4 PARADIS_Repair(i)

```
1: tail = gt_i
2: for p ∈ P do
3:   head = ph_i^p
4:   while head < pt_i^p&&head < tail do
5:     v = d[head + +]
6:     if b(v)! = i then
7:       while head <= tail do
8:         w = d[− − tail]
9:         if b(w)! = i then
10:          d[head − 1] = w
11:          d[tail] = v
12:          break
13: gh_i = tail
```

入れ替えが可能である. しかし, これだけだと分割によって各バケットにある key の分布が元の分布と異なるため sort を完了することができない. この時, PARADIS_Permute フェーズで sort を完了できず誤った位置に残されてしまった key を wrong element と呼ぶ. Wrong elements を適切な位置に置くために次の節で述べる PARADIS_Repair フェーズが必要となる. PARADIS_Permute のアルゴリズムを Algorithm 3 に示す.([3] より引用したものを改変)

3.3 Repair

PARADIS_Permute フェーズの終了後, PARADIS_Repair フェーズを行う. PARADIS_Repair フェーズ後に wrong element が残っている場合は再び PARADIS_Permute を行う. PARADIS_Repair フェーズでは, 各スレッドごとで行なった PARADIS_Permute フェーズで発生した wrong elements を一箇所にまとめる操作を行う. この操作によって, 次の PARADIS_Permute フェーズではその前に行うよりも少ない数に対して同様の操作を繰り返していけば良いということになる. 要素数が十分に小さくなった時は並列化するオーバーヘッドが大きくなるためシングルスレッドで実行する. Permute フェーズをシングルスレッドでおこなうことは元の American Flag Sort における入れ替え操作と同等であり, この一連の permute, repair 操作は無限ループに陥ることなく終了する. 従来の PARADIS において repair 操作においての並列化はバケット単位で行なっているのみである. このため, バケットのサイズのバランスが取れていない場合, 性能が劣化し, 逐次化してしまい並列であるメリットを活かせなくなってしまう可能性がある.

図 2 は 16 スレッドにおいて実行した, 各フェーズにおける PARADIS の実行時間を示している. 図より, repair フェーズの時間 (repair) はヒストグラムの構築の時間 (buildHist) やデータの入れ替え操作の時間 (permute) よりも相対的にとても短いことがわかる. そのため, 従来の PARADIS では repair フェーズにおける逐次化は大きな問題にならないとしている.

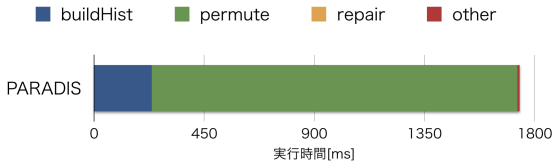

図 2 PARADIS における実行時間内訳

最後に PARADIS_Repair のアルゴリズムを Algorithm 4 に示す. ([3] より引用したものを改変)

3.4 ロードバランス

再帰の際に, 呼び出す関数に渡す sort 対象範囲の大きさに応じて呼び出される関数において, 並列に実行するスレッド数を制御する. これは PARADIS ではロードバランスと呼ばれる. このような再帰操作の並列化は OpenMP では task 構文を用いて実装ができる.

ロードバランスによる並列に実行するスレッド数の決定方法について述べる. PARADIS におけるロードバランスは再帰によって関数を呼び出す直前に渡す引数に基づいて行われる. ロードバランスにおけるスレッド数の決定は計算量に基づいて行う. 現在の実行しているスレッド数を P とし, 求める i 番目のバケットに対して呼び出されるスレッド数を P_i, バケットの要素数を C_i とする. r をバケットに含まれる最大の数とした時, 必要な recursion のレベルが $\log_B r$ であるから, そのバケットの計算量は $O(C_i \log_B r)$ である. r は動的な値であり実際の r を毎回求めるのは計算コストが大きいため, このバケットの計算量を $O(C_i \log_B C_i)$ とみなすと, 求める P_i は $P_i = P \times (C_i \log_B C_i)/(\Sigma_{j \in B} C_j \log_B C_j)$ とすることができる. このロードバランスと呼ばれる手法によって PARADIS は偏った分布 (skew) に対して性能の劣化を抑えることができる.

3.5 計算量

原論文によれば PARADIS の計算量[*2]は, データの件数

[*2] 厳密には Work-Span Model の Work に該当する.

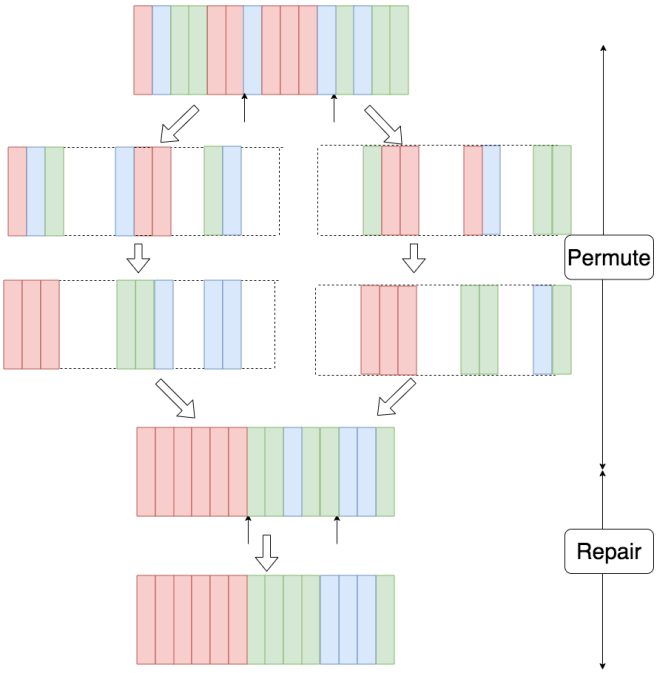

図 3　PARADIS の入れ替え操作

を N，基数を B，データに含まれる数値の最大値を r とした時, 各再帰のレベルでデータを入れ替える操作が $O(N)$ であり, レベルは $\log_B r$ あることから, $O(N \log_B r)$ である．また, PARADIS の span は repair フェーズにおける wrong elements の最大の割合を w, その時動作しているスレッド数を P とおくと $O(N(1/P + w) \log_B r)$ である．

4. 提案

4.1 PARADIS の問題：Block case

従来の PARADIS にはある種のデータ分布に対して性能劣化を引き起こす可能性がある．PARADIS が有する 5 つのフェーズはいずれも並列化可能であることをこれまでで述べたが，ある種のデータ分布を repair フェーズが受け取ると，効果的に repair フェーズを並列化できず, 最悪逐次化されることがある．そのような具体例を図 4 に示す．この図では 2 つのスレッドが同時並列的に処理している様子が描かれている．図中では赤色部分が左端にあるべき key であることを示している．点線に区切られた領域をバケットと呼ぶ．バケットとは第一フェーズである Build Histgram により求められた key のクラスタである．バケットには 2 つの部分があり，1 番目が非赤色，2 番目が赤色になっている．この図では赤色 key が非赤色 key の 10 倍存在していることが示されている．上段に初期状態が示されており，それが permute フェーズを経て中段の状態となり，repair フェーズを経て下段になる様子が示されている．

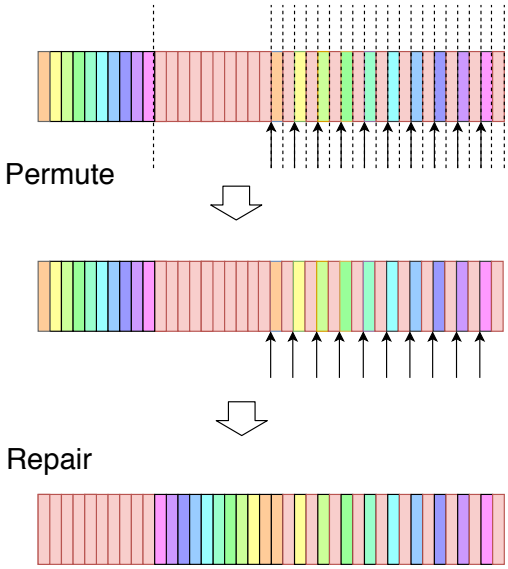

図 4　逐次化をもたらすデータ分布 (block case)

この図では，permute フェーズを経たにも関わらず，上段と中段の状態が変動していないことが示されている．今回のケースでは，repair フェーズで処理するデータ数が全体の 1/4 という莫大な量になってしまっている．

このように repair フェーズで処理すべきデータ量が莫大になってしまう理由は，赤色 key の分布状態に問題があるからである．この赤色 key の分布状態では，permute フェーズで swap 操作がこれ以上できない．この状態では各バケットのパーティションの 2 番目に赤色の key がクラスタ化して存在しているために, 2 番目のパーティション同士で入れ替えの操作ができない．このような permute を妨げる key を我々は permute blocker と呼び, permute blocker が頻発するケースを block case と呼ぶことにする．

Block case では repair フェーズで各バケットに存在する wrong element の数に，バケット毎の偏りが生じる可能性がある．このバケット毎の wrong element の偏りにより，スレッド間の同期に際しての待ち時間が延びることになる．なぜならスレッド毎に担当するバケットの処理量が異なることになるからである．偏りが強い場合には，結果的に repair 処理が逐次化されることになる．これによって PARADIS アルゴリズムの並列度が急激に劣化してしまう．Permute blocker がもたらすバケット毎の偏りに対処できるよう PARADIS を拡張しなければ，permute blocker に起因する性能劣化を避けられない．

4.2 提案手法：PARADIS_Repair_AP

本節では提案手法である PARADIS_Repair_AP を説明する．PARADIS_Repair_AP は前節の permute blocker が引き起こす問題を解決するものである．

従来の PARADIS_Repair においてこの問題が解決できない理由は，wrong elements の移動先が，repair 実行前に決

まらず, 実行時に決定されるからである．従来の PARADIS の repair フェーズでは各バケットの右端に wrong elements を寄せる．この wrong elements を移動するとき，移動先の領域にも wrong elements が存在するかもしれない．そのため各 wrong element の移動先インデックスの情報が必要になる．この移動先インデックスは repair フェーズ実行中に変わる可能性がある．なぜなら repair 処理で実行する swap 先のデータが wrong element ならば swap の意味がないからである．このような場合，swap 先のインデックスをデクリメントして，改めて swap 処理を実行する．このためにバケット内における repair 処理は並列化できない．

上記の問題を解決するために，我々は各バケットにおける swap 先インデックスを事前に求める手法を提案する．そもそも従来手法の問題は，swap 先が不適切であることが，swap 実行の直前でなければわからないことである．ところがその情報は repair フェーズにおいて permute フェーズにおいて効率よく求めることが可能なことを，我々は発見した．これは permute フェーズにおいて各バケットの wrong elements がバケット内の各パーティションの右端に集まる性質を利用するものである．Algorithm 5 に適応的分割法に基づく PARADIS_Repair のアルゴリズムである PARADIS_Repair_AP を示す．今後 PARADIS_Repair の代わりに PARADIS_Repair_AP を適用した PARADIS を PARADIS_AP と表記する．

図 5 と Algorithm 5 を用いて提案手法 PARADIS_Repair_AP の計算手順を説明する． 図 5 の黒矢印はバケットの境界, 点線は各パーティション, 白矢印は wrong elements の移動先を示している．まず, 各バケットについての処理は独立のためバケットごとに並列化を行う．次に, 各バケットについてそれぞれの分割されたパーティションの中において並列化を行うことを考える．これは図 5 において, 左端のバケットで各パーティション内で 2 並列で処理を行うことを意味する．そのパーティションまででいくつ wrong element を目標位置に動かしたかを記録する変数を $nthAftKey$ とする.(Line 1)．各パーティションの wrong element のインデックスを ii とすると, それぞれの wrong elements の行き先は各パーティションで左から何番目であるかを示す数 $(ii - ph_i^p)$, k 番目のバケットの開始位置 gh_i を用いて $gh_i + (ii - ph_i^p) + nthAftKey$ と計算できる (Line 5,Line 9)．このように, permute フェーズにおける swap 移動先位置の計算には既存の PARADIS_Permute を修正する必要はなく，追加のデータ構造は不要である．そのため, この拡張によって PARADIS の省メモリであるという特性を損なうことはない．

一方で, wrong elements の分布状態によっては並列化できない可能性もある．すなわち, 各パーティションの処理中に wrong elements を動かす先を決める直前において, その wrong elements の数 (Algorithm5 における $pt_i^p - ph_i^p$)

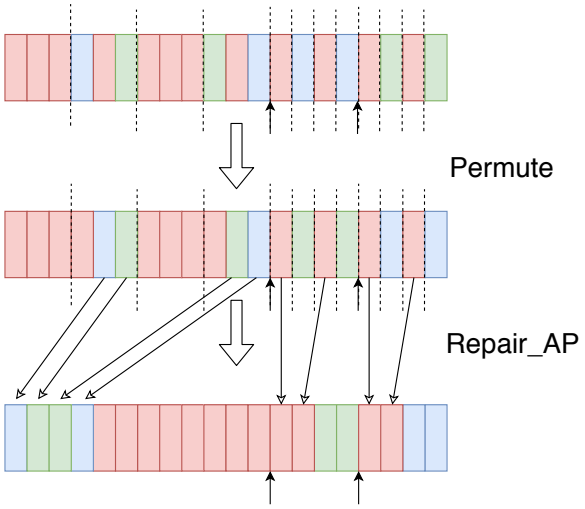

図 5　PARADIS_Repair_AP

が wrong elements の左端より左側にある correct elements の数 (Algorithm5 における $ph_i^p - nthAftKey$) より多い場合はこのままでは並列化不能のため, 並列化をしない (Line 4)．

このように, 提案手法を用いれば，swap 先に wrong element が存在する可能性がなくなり, バケット内のパーティション内での並列化が可能となる．従って，各バケットの何番目の wrong element がどこにあるべきかが PARADIS_Repair 処理の swap 前にわかる．各 wrong element の移動先が各バケットの何番目かが決まれば, 同一バケットの他のスレッドがすでに wrong elements を swap した位置に対して swap を行うことはない．

並列化されたループ内のさらなる並列化となってしまうため，ここで新しくスレッドを呼び出すオーバーヘッドがうまれるデメリットある．そのため, swap 回数がしきい値以下の時は並列化による高速化よりもスレッド生成のオーバーヘッドが大きくなるため並列化をしない (Line 4)．しかし, 近年の CPU は instruction レベルの並列性を CPU での計算において効率的に実行することが期待できる [11] ため, たとえオーバーヘッドが上回り, スレッドによる明示的な並列化ができない場合においてもこの改善は有効な可能性がある．

5. 実験

提案手法の有効性を評価するために，提案手法を実装し，実験を行った．実装は C/C++言語を用いた．コード行数は 500 程度である．並列化には OpenMP を利用した．実験環境は Amazon EC2 x1.32xlarge を利用した．環境は次の通りである．Intel(R) Xeon(R) CPU E7-8880 v3 @ 2.30GHz , コア数：128)，コンパイラ：icc ，最適化オプション -O2(デフォルト)．実験においては各測定を 5 回行い, その平均をとった．本環境において int 型は 32bit であ

Algorithm 5 PARADIS_Repair_AP(i)

```
1: nthAftKey = 0
2: for p = 0 ... processNum do
3:   if ph_i^p − nthAftKey ≥ pt_i^p − ph_i^p && threshold ≤ pt_i^p − ph_i^p
   then
4:     for ii = ph_i^p ... pt_i^p do in parallel
5:       swap(arr[gh_i + (ii − ph_i^p) + nthAftKey], arr[ii])
6:     end for
7:   else
8:     for ii = ph_i^p ... pt_i^p do
9:       swap(arr[gh_i + (ii − ph_i^p) + nthAftKey], arr[ii])
10:  nthAftKey+ = (pt_i^p − ph_i^p)
11: gt_i = gh_i + nthAftKey
```

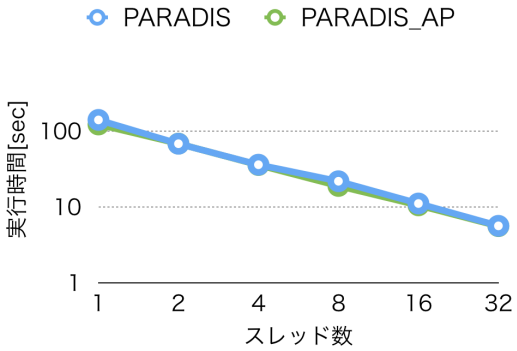

図 6　一様分布における PARADIS のスレッド数に対する実行時間

る．基数は PARADIS の原論文と同じく 256 とした．

5.1　permute blocker が少ないデータ分布に対する実験

図 6 に一様分布から生成された int 型配列に対する sort 実行時間を示す．本実験において PARADIS_AP は十分な数の wrong elements がないため，repair phase でのバケット内の並列化をしていない．図からわかるように PRADIS_AP は PARADIS 同様にスケールする．それに加えて PARADIS_AP の性能は PARADIS と同様である．これより我々の拡張は block case でない場合において PARADIS に悪影響を及ぼしていないことがわかる．

5.2　逐次化をもたらすデータ分布に対する実験

PARADIS_AP の性能を調査するため，PARADIS_Permute の n 番目どうしでしか入れ替えることができないという特性を利用し，多くの permute blocker を発生させ，さらに，発生する wrong element がバケット単位で偏るようなケース，block case を作成した．今回使用した block case ではデータ件数 N に対して N/4 個の permute blocker が発生する．図 7 に実験結果を示す．実験結果より，PARADIS_AP が PARADIS よりも常に高性能である点が観察される．また，図 6 の実験結果と比較すると，その高性能の度合いには明確な差があることも観察される．この理由は提案手法 PARADIS_AP が permute blocker に対処する術を備えているからであると考えら

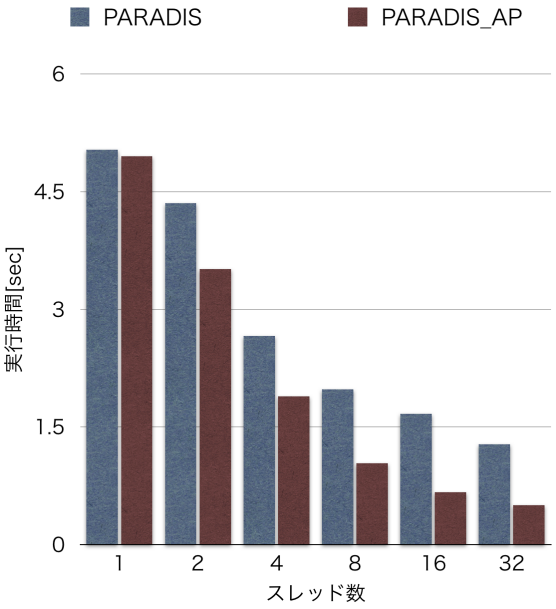

図 7　Block case におけるスレッド数に対する実行時間の変化

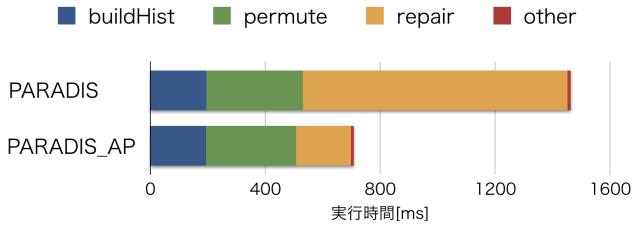

図 8　Block case における実行時間内訳

れる．64 スレッド以上の結果が未記載であるのは repair フェーズにおけるバケット内での並列化数を 4 に設定しており，今回使用したマシンは 128core のため，一つのフェーズの間に利用される core 数が今回の環境においては不足してしまうからである．本実験では Repair_AP における並列するかどうかの閾値は 10^7 とした．図 8 に 16 スレッドにおいて sort を実行した時の PARADIS と PARADIS_AP の実行時間の内訳を示す．permute フェーズにおける swap 操作の回数が permute blocker によって減るため，全体に対して permute フェーズの占める時間が図 2 のような他のケースに比べて相対的に短くなっている．また，block case においては全体の実行時間に占める repair フェーズの時間割合が爆発的に増え，従来の PARADIS の性能を劣化させている．一方で今回の提案手法である PARADIS_AP は repair フェーズの時間を適切なバケット内の分割による並列化処理によって削減することに成功していることがわかる．この並列化処理の有無が図 7 の実験結果における有意な差分の理由だと考えられる．

6.　結論

本論文では，まず最新の並列 radix sort アルゴリズムで

ある PARADIS を再実装し, それが多くの場合で優れた性能を示す一方で一部のケースでは逐次実行と変わらなくなってしまうことを検証した. 次に従来の PARADIS の弱点である block case に対処するための新しい手法, 適用的分割法を提案し, 実装および, 検証を行なった. 提案手法を取り入れた PARADIS_AP は block case 以外について PARADIS と同等以上の性能が確認でき, block case において最大で 57%の性能改善を確認できた.

謝辞 本研究の一部は JSPS 科研費 JP19H04117 による。

参考文献

[1] G. M. Amdahl. Computer architecture and amdahl's law. *Computer*, Vol. 46, No. 12, pp. 38–46, dec 2013.

[2] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Software: Practice and Experience*, Vol. 23, No. 11, pp. 1249–1265, 1993.

[3] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, and Ruchir Puri. PARADIS: an efficient parallel algorithm for in-place radix sort. *Proceedings of the VLDB Endowment*, Vol. 8, No. 12, pp. 1518–1529, 2015.

[4] Jamie Farnes, Ben Mort, Fred Dulwich, Stef Salvini, and Wes Armour. Science Pipelines for the Square Kilometre Array. *Galaxies*, Vol. 6, No. 4, p. 120, 2018.

[5] gorset. Is radix sort faster than quicksort for integer arrays? http://erik.gorset.no/2011/04/radix-sort-is-faster-than-quicksort.html.

[6] Marek Kokot, Sebastian Deorowicz, and Agnieszka Debudaj-Grabysz. Sorting data on ultra-large scale with RADULS: New incarnation of radix sort. *Communications in Computer and Information Science*, Vol. 716, pp. 235–245, 2017.

[7] Marek Kokot, Sebastian Deorowicz, and Maciej Długosz. Even faster sorting of (not only) integers. *Advances in Intelligent Systems and Computing*, Vol. 659, pp. 481–491, 2018.

[8] Ryuya Mitsuhashi, Hideyuki Kawashima, Takahiro Nishimichi, and Osamu Tatebe. Three-dimensional spatial join count exploiting CPU optimized STR r-tree. In *2016 IEEE International Conference on Big Data, Big-Data 2016, Washington DC, USA, December 5-8, 2016*, pp. 2938–2947. IEEE Computer Society, 2016.

[9] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. Theoretically-efficient and practical parallel in-place radix sorting. In *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 213–224, 2019.

[10] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, No. January, pp. 351–362, 2010.

[11] Malte Skarupke. I Wrote a Faster Sorting Algorithm. https://probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm/.

[12] voutcn. kxsort Fast, in-place radix sort. https://github.com/voutcn/kxsort.