Vol.2022-HPC-184 No.1
2022/5/11

スレッド並列化されたソートの富岳上での性能評価

徳植 智之[1,a)]　石山 智明[2]

概要：ソートは最も基本的なアルゴリズムの 1 つである．一方，スーパーコンピュータ「富岳」の 1 ノードは 48CPU コアで構成されており，高度に並列化されたソートプログラムが重要である．しかし，「富岳」上で高速かつ高い並列化性能をもつソートプログラムはあまり報告されていない．本研究ではサンプルソートをベースにした 2 種類のスレッド並列ソートアルゴリズムを実装し，「富岳」上で性能比較を行った．ひとつめは，入力列を複数のブロックに分けてそれぞれソートし，各ブロックから等間隔にサンプリングを行いピボットを選ぶ．ふたつめは，ピボットの選び方が異なり，全てのパーティションの要素数が等しくなるようなピボットを二分探索によって選ぶ．それぞれに対し，逐次ソートやマージのアルゴリズムを変えて性能を比較した．その結果，条件分岐を減らすことで高速化されたクイックソート（Block Quick sort）を逐次ソートに，selection tree をマージに用いたふたつめのサンプルソートが，さまざまな入力に対し安定して高い速度性能および並列化性能を示した．

1. はじめに

ソートは最も基本的なアルゴリズムの 1 つであり，様々な場面で高速なソートプログラムが求められる．一般に高速な逐次ソートアルゴリズムとして，クイックソート [2] が知られている．

スーパーコンピュータ「富岳」の計算ノードは 48 個の CPU コアで構成されている．したがって，「富岳」上で高速にソートするためには，高度にスレッド並列化されたソートプログラムが重要である．しかし，「富岳」上で高速かつ高い並列化性能をもつソートプログラムはあまり報告されていない．

本研究の目的は「富岳」上で高速に実行できる汎用的なソートプログラムを作成することである．そのために，サンプルソートをベースにした 2 種類のスレッド並列ソートアルゴリズムを実装し性能を比較した．さらに，それぞれのアルゴリズムに対し逐次ソートやマージのアルゴリズムを変えて「富岳」上で性能を比較した．

2. クイックソート

N 個の要素から成る入力列 $A_0, \ldots, A_{N-1}$ からピボット P を選び，ある $k \in \{0, \ldots, N-1\}$ に対して $A_0, \ldots, A_k \leq P$ かつ $A_k, \ldots, A_{N-1} \geq P$ が成り立つように要素を並べ替えることをパーティショニングと呼ぶ．クイックソートは，パーティショニングを再帰的に繰り返して入力列全体をソートする．クイックソートの平均計算量は $O(N \log N)$ である．

パーティショニングの際，常に入力列の最大値または最小値がピボットとして選ばれる場合，クイックソートの計算量は $O\left(N^2\right)$ である．そのため，クイックソートの最悪計算量を $O(N \log N)$ に改善したイントロソート [4] が使われることも多い．イントロソートとは，クイックソートの再帰呼出しの深さが予め設定した最大値に達したら，ヒープソートに切り替えるソートアルゴリズムである．深さの最大値が $O(\log N)$ ならば，イントロソートの最悪計算量は $O(N \log N)$ である．

プログラムが条件分岐を含む場合でも演算パイプラインを活用するために，プロセッサは条件分岐の分岐先を予測し実行する．しかし，現代のプロセッサの演算パイプラインは長く，予測が外れるとプログラムの性能は下がる．Block Quick sort [1] では，分岐予測ミスによるクイックソートの性能低下を防ぐために，ブロックパーティショニングを行う．一般的なパーティショニング [2] では，入力列を両端から走査しながら要素をピボットと比較し，交換していく．一方，ブロックパーティショニングは，交換すべき要素の位置をバッファに保存しておき，走査が終わったら交換する．要素をピボットと比較し位置をバッファに保存する操作は ARMv8 の場合，CSET 命令や CINC 命令を使って実行でき，パーティショニングにおける条件分岐の数を減らせる．

Pattern Defeating Quick sort [5] は，入力列を「ピボッ

[1] 千葉大学 融合理工学府 数学情報科学専攻 情報科学コース
[2] 千葉大学統合情報センター
[a)] t.tokuue@chiba-u.jp

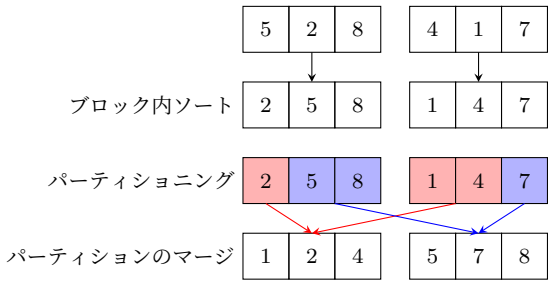


図 1: サンプルソートをベースにした並列ソートアルゴリズム．まず入力列を 2 つのブロックに分け，それぞれソートする．次にピボットとして 4 が選ばれたとしてパーティショニングを行う．最後に各ブロックからパーティションを集めてマージする．

ト未満」，「ピボットと同じ」，「ピボットより大きい」の 3 つに分けるパーティショニングを行う．これにより，キーの種類数 k が十分小さい場合 $O(Nk)$ でソートできる．また，ヒープソートに切り替える精度がイントロソートよりも高くなるように，要素数の偏りが大きいパーティショニング（bad partition）の数が $\lfloor \log N \rfloor$ となったら，ヒープソートに切り替える．

3. サンプルソート

サンプルソートは，並列実行可能なソートアルゴリズムの 1 つである．サンプルソートでは，入力列から複数のピボットを選び，パーティショニングを行う．

サンプルソートをベースにした並列ソートアルゴリズム [6,7] では，N 個の要素から成る列 A を，次の 4 つの手順でソートする．

(1) ブロック内ソート
(2) ピボット選択
(3) パーティショニング
(4) パーティションのマージ

まず，A を n_B 個のブロックに分けて，それぞれ逐次ソートする（手順 1）．ブロックとは，要素数 $\lceil N/n_\mathrm{B} \rceil$ の連続部分列である．手順 2 で $n_\mathrm{P}-1$ 個のピボット $P_1, \ldots, P_{n_\mathrm{P}-1}$ を選び，手順 3 で各ブロックを n_P 個のパーティションに分ける．$P_0 = -\infty$，$P_{n_\mathrm{P}} = \infty$ とすると，k 番目 $(0 \le k < n_\mathrm{P})$ のパーティションは，$P_k < x \le P_{k+1}$ を満たす要素 x から成る連続部分列である．手順 4 で全てのブロックの k 番目 $(0 \le k < n_\mathrm{P})$ のパーティションをマージして順に並べると，ソートが完了する．手順 1 と手順 3 は異なるブロックについて，手順 2 は異なるピボットについて並列に実行できる．手順 4 で異なるパーティションのマージは並列に実行できるが，出力列に書き込む際の先頭からのオフセットは逐次計算する．

$A = \{5,2,8,4,1,7\}$，$n_\mathrm{B} = 2$，$n_\mathrm{P} = 2$ の場合の実行例を**図 1** に示す．A を要素数 3 のブロックに分けてソートした後，ピボットとして $P_1 = 4$ が選ばれるとする．各ブロックを「4 以下」，「4 より大きい」の 2 つのパーティションに分け，それぞれマージして順に並べると，ソート列 $\{1,2,4,5,7,8\}$ が得られる．

データ構造を使うと，n_B 個のソート列を効率的にマージできる．各ソート列の先頭を二分ヒープや selection tree [3] で管理し，最小の要素を順に取り出していくと，1 つのソート列が得られる．最小要素の取得と更新は $O(\log n_\mathrm{B})$ でできるため，要素の総数を n_M とすると，マージの計算量は $O(n_\mathrm{M} \log n_\mathrm{B})$ である．

データ構造を使わずにマージする方法として，マージの代わりにソートする方法が挙げられる．n_B 個のソート列をメモリ上で連続するように並べて逐次ソートすると，マージが完了する．この場合，計算量は $O(n_\mathrm{M} \log n_\mathrm{M})$ となってしまうが，メモリアクセスが局所化されるため，キャッシュ効率は高くなると考えられる．

異なるパーティションを並列にマージする場合，各パーティションの合計要素数が均等になるようにピボットを選ばなければ，並列化効率が下がる可能性がある．Parallel Sorting by Regular Sampling (PSRS) [6] では，各ブロックから $n_\mathrm{P}-1$ 個ずつサンプルを取ってソートし，サンプルから $n_\mathrm{P}-1$ 個のピボットを選ぶ．サンプルやピボットをソート列から等間隔に選ぶことで，各パーティションの合計要素数が均等になるようにする．ピボット選択の計算量は，サンプルを逐次ソートする計算量と一致するため，$O(n_\mathrm{B} n_\mathrm{P} \log(n_\mathrm{B} n_\mathrm{P}))$ である．

「富岳」上で実行可能なサンプルソートの実装である jmakino sort[*1] では，ランダムにサンプリングを行い，PSRS と同様の方法でピボットを選ぶ．サンプルの総数が N に比例しスレッド数 t に反比例するため，ピボット選択の計算量は $O(N/t \log(N/t))$ である．

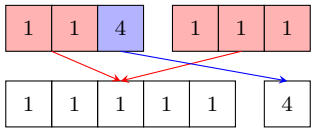
重複要素が大量にある場合，ピボットによるパーティショニングでは，パーティションの合計要素数が均等にならないことがある．例えば $A = \{1,1,4,1,1,1\}$ の場合，ピボットをどのように選んでもパーティションの合計要素数の最大値は 5 以上となる（**図 2a**）．パーティションの合計要素数に偏りが生じると，マージにかかる時間のばらつきが大きくなるため，スレッド数を増やした時に並列化効率が下がる．そこで，Parallel Sorting using Exact Splitting (PSES) [7] は式 (1) を満たすようにピボット P_k を選び，c_k を式 (2) で定める．

$$|\{x \in A \mid x < P_k\}| \le k\frac{N}{n_\mathrm{P}} \le |\{x \in A \mid x \le P_k\}| \quad (1)$$

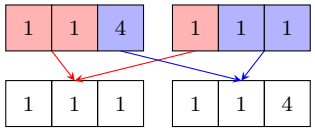
$$c_k = k\frac{N}{n_\mathrm{P}} - |\{x \in A \mid x < P_k\}| \quad (2)$$

k 番目 $(0 \le k < n_\mathrm{P})$ のパーティションには，$P_k < x < P_{k+1}$ を満たす要素 x に加え，$y = P_{k+1}$ を満たす要素 y を c_k 個含

[*1] https://github.com/jmakino/sortlib



(a) ピボット $P_1 = 1$ によるパーティショニング．1 番目のパーティションに含まれる要素が 2 番目のパーティションに含まれる要素よりも多いため，マージ時間の差が大きくなる．



(b) PSES で $P_1 = 1$，$c_1 = 3$ とした場合のパーティショニング．1 番目のパーティションには，1 未満の要素と 3 つの 1 が含まれる．2 番目のパーティションには，残り 2 つの 1 と 1 より大きい要素が含まれる．どちらのパーティションも合計要素数が 3 であるため，マージ時間の差が小さくなる．

図 2: 重複要素が大量にある場合のパーティショニングの違い．

表 1: 「富岳」の計算ノード単体性能

計算コア数	動作周波数	メモリ容量	メモリバンド幅
48	2.00 GHz	32 GB	1024 GB/s

める．これにより，パーティションの合計要素数は N/n_{P} となる．この方法は，重複要素がどれだけあっても，パーティションの合計要素数を均等にできる．式 (1) を満たす P_k を見つけるために，A の要素に対して二分探索を行う．ブロックごとに P_k 未満の要素と P_k 以下の要素の個数を数えて式 (1) を判定すると，P_k と c_k は $O(n_{\mathrm{B}} \log N \log\lceil N/n_{\mathrm{B}} \rceil)$ で求められる．$A = \{1, 1, 4, 1, 1, 1\}$，$n_{\mathrm{P}} = 2$ の例では，1 未満の要素は 0 個，1 以下の要素は 5 個ある．このとき $0 \leq N/n_{\mathrm{P}} = 3 \leq 5$ が成り立つので，$P_1 = 1$，$c_1 = 3$ とすれば図 **2b** のように各パーティションの合計要素数を 3 にできる．

4. 性能評価

4.1 実験環境と問題設定

本研究では，理化学研究所の計算科学研究センターが保有するスーパーコンピュータ「富岳」を使用して，並列ソートアルゴリズムの性能を評価した．「富岳」の計算ノード単体性能を**表 1** に示す．

並列ソートアルゴリズムとして PSRS と PSES，マージのためのデータ構造として selection tree を実装した．並列ソートアルゴリズムのブロック数 n_{B} とパーティション数 n_{P} は，いずれもスレッド数と一致させる．プログラムは C++ で記述し，富士通コンパイラ FCC 4.7.0 の clang モードでコンパイルした．コンパイラの最適化オプションは`-Ofast` を指定する．また，プログラムのスレッド並列化には OpenMP を使用した．

6 種類の入力列（**表 2**）について，擬似乱数のシードを変えて 20 回ソートし，平均経過時間によってアルゴリズムの性能を評価する．UniformInt は 0 以上 $2^{32} - 1$ 以下の整数値をとる一様乱数，UniformFloat は 0 以上 1 未満の浮動小数点数をとる一様乱数で入力列を初期化する．AlmostSorted は，0 から $N - 1$ を順に並べた後，ランダムに選ばれた $\sqrt{N}$ 個の要素の位置を交換する．Duplicate3 は 0 以上 2 以下の整数値をとる一様乱数で初期化する．とりうる値は 3 通りであるため，要素に多くの重複が生じる．Pair と Particle は，ソート用のキー（uint64_t）とデータから成る構造体の配列である．Pair はデータとして配列のインデックスを持つ．Particle は，N 個の粒子を何らかのキーでソートする場合の性能を評価するために，3 次元空間内の粒子データ（質量，位置，速度，加速度，ポテンシャル）を持つ．これらは重力多体シミュレーションにおいて粒子を表す構造体の典型的な構成要素である．どちらもソート用のキーは 0 以上 $2^{64} - 1$ 以下の整数値をとる一様乱数で初期化する．

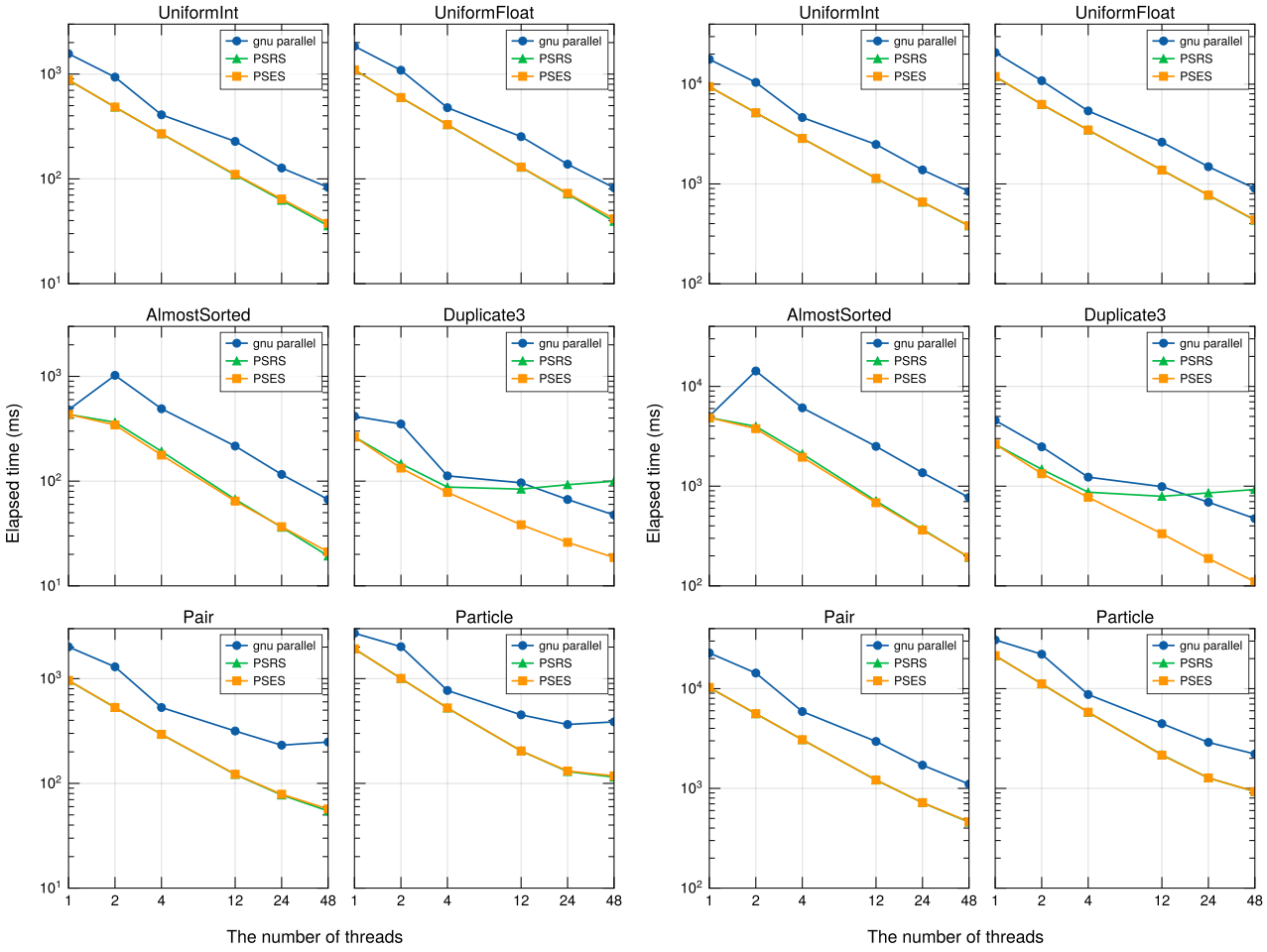
表 2: 入力列の種類とデータサイズ

種類	型	サイズ（Byte）
UniformInt	uint32_t	4
UniformFloat	float	4
AlmostSorted	uint32_t	4
Duplicate3	uint32_t	4
Pair	構造体	16
Particle	構造体	96

4.2 並列化の違い

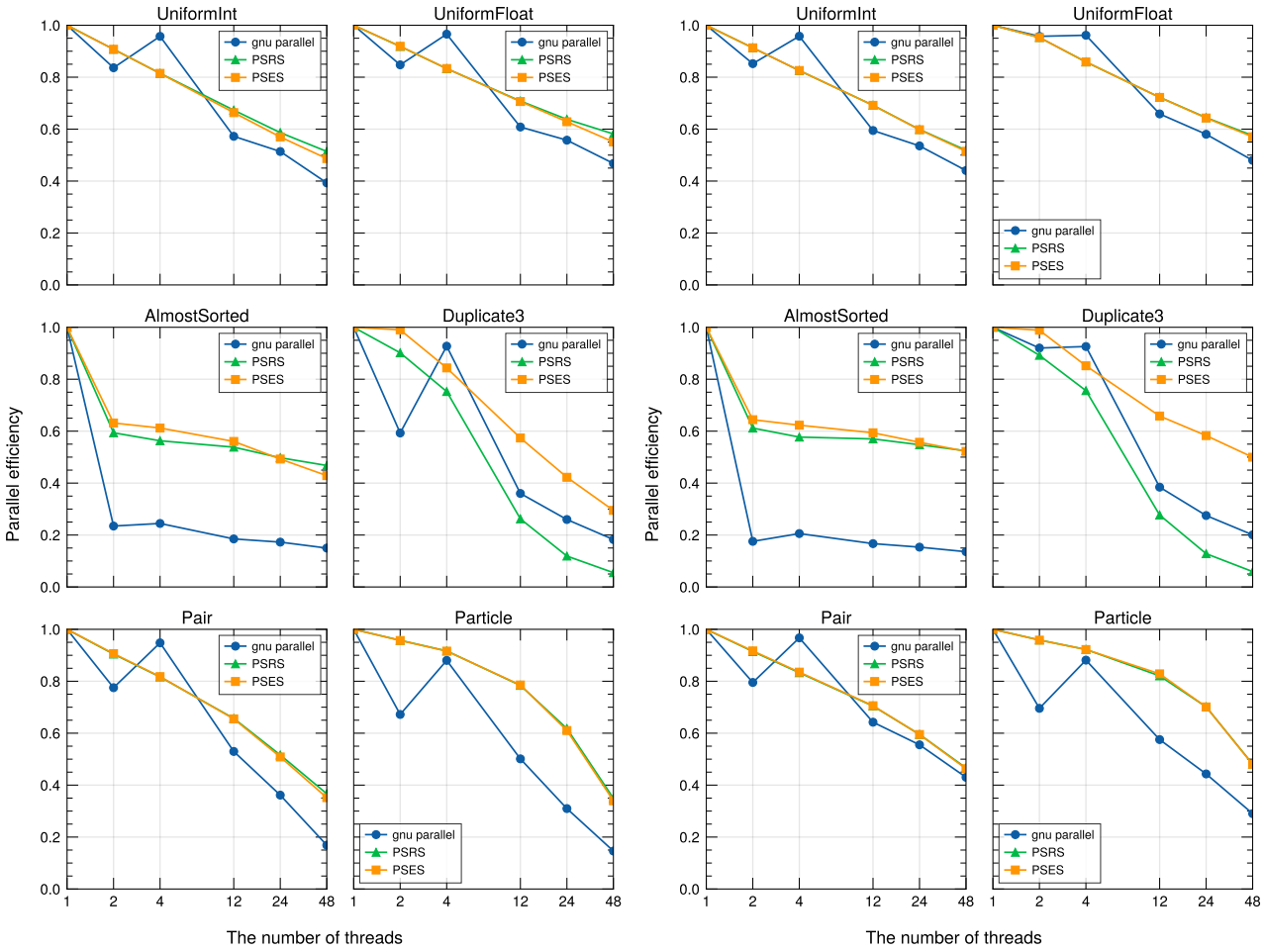
異なる並列ソートアルゴリズムの比較を図 **3** に示す．PSRS と PSES の逐次ソートには Block Quick sort，パーティションのマージには selection tree を用いた．__gnu_parallel::sort は GNU C++ 標準ライブラリ libstdc++ の parallel mode [8] に含まれる Parallel Multiway Mergesort [9] である．基本的なソートの手順は本研究で実装した 2 つのアルゴリズムと同様であり，逐次ソートには std::sort，k 個（$k > 4$）のソート列のマージには selection tree を用いている．

$N = 10^7$ の Pair と Particle に対して，__gnu_parallel::sort は 24 スレッド以上でスレッド数を増やしてもソート時間が短くならないが，PSRS と PSES は 48 スレッドまでソート時間が単調減少する．いずれの入力に対しても，本研究で実装した PSES は標準ライブラリの__gnu_parallel::sort と比べて 48 スレッドで 2 倍以上高速だった．PSRS は，要素の重複が少ない入力に対しては PSES とほとんど同じ結果だったが，Duplicate3 に対しては 4 スレッド以上でスレッド数を増やしてもソート時間が短くならない．これは，スレッド数が要素の種類数より大きくなると，どのようにピボットを選んでも，各スレッドがマージするパーティショ



(a) $N = 10^7$ (b) $N = 10^8$

図 3: 異なる並列ソートアルゴリズムによるソート時間．本研究で実装した PSRS と PSES に加え，libstdc++ の__gnu_parallel::sort を使う場合を比較する．PSRS と PSES の逐次ソートには Block Quick sort，パーティションのマージには selection tree を使う．



(a) $N = 10^7$ (b) $N = 10^8$

図 4: 異なる並列ソートアルゴリズムの並列化効率．本研究で実装した PSRS と PSES に加え，libstdc++ の__gnu_parallel::sort を使う場合を比較する．PSRS と PSES の逐次ソートには Block Quick sort，パーティションのマージには selection tree を使う．

ンの合計要素数に偏りが生じるためである．これらの結果は $N = 10^8$ においても同様だった．

それぞれの並列ソートアルゴリズムについて，並列化効率を図 **4** に示す．$N = 10^8$ の場合，いずれの入力に対しても 12 スレッド以上の並列化効率は PSES が最も高く，48 スレッドで 0.5 程度の並列化効率を維持している．PSRS の並列化効率は Duplicate3 以外では PSES と同程度だが，Duplicate3 に対しては 48 スレッドで PSES の約 1/10 の並列化効率である．Duplicate3 以外では__gnu_parallel::sort の並列化効率が最も低く，特に AlmostSorted に対する並列化効率が低い．これは他の入力と比べて 1 スレッドの場合の逐次ソートが高速であるためと考えられる．

Pair と Particle に対しては全てのアルゴリズムで $N = 10^7$ に比べ $N = 10^8$ の場合，スレッド数を増やした時の並列化効率が高くなる．同じスレッド数で N を大きくすると，各スレッドが並列に実行する部分（ブロック内ソートやパーティションのマージ）にかかる時間が長くなり，並列実行できない部分にかかる時間の割合が小さくなる．特に，データサイズが大きい Pair や Particle では，データのコピーに時間がかかるため，N を大きくすると並列化効率が高くなると考えられる．

4.3 逐次ソートの違い

ブロック内ソートに使用する逐次ソートアルゴリズムを変えた時の PSES のソート時間を図 **5** に示す．いずれの場合もパーティションのマージには selection tree を使い，逐次ソートとしてイントロソート，Pattern Defeating Quick sort，Block Quick sort の 3 種類を比較する．イントロソートは GNU C++ 標準ライブラリ libstdc++ の std::sort，Pattern Defeating Quick sort は Boost ライブラリの boost::sort::pdqsort を使う．Block Quick sort の実装[*2]にはピボットの選び方などが異なる複数のバージョンがあるが，その中の blocked_double_pivot_check_mosqrt::sort を使う．

ほとんどの入力に対して，Block Quick sort を使う場合が高速であるが，Duplicate3 に対しては Pattern Defeating Quick sort を使う場合のソート時間が最も短くなった．これは，要素の種類数が少ないとき高速にソートできるためと考えられる．

4.4 マージの違い

パーティションのマージ方法を変えた時の PSES のソート時間を図 **6** に示す．ブロック内ソートには std::sort を使い，パーティションのマージ方法として二分ヒープを使う場合と selection tree を使う場合，マージの代わりにソートする場合の 3 種類を比較する．二分ヒープには std::priority_queue，ソートには std::sort を使う．

[*2] https://github.com/weissan/BlockQuicksort

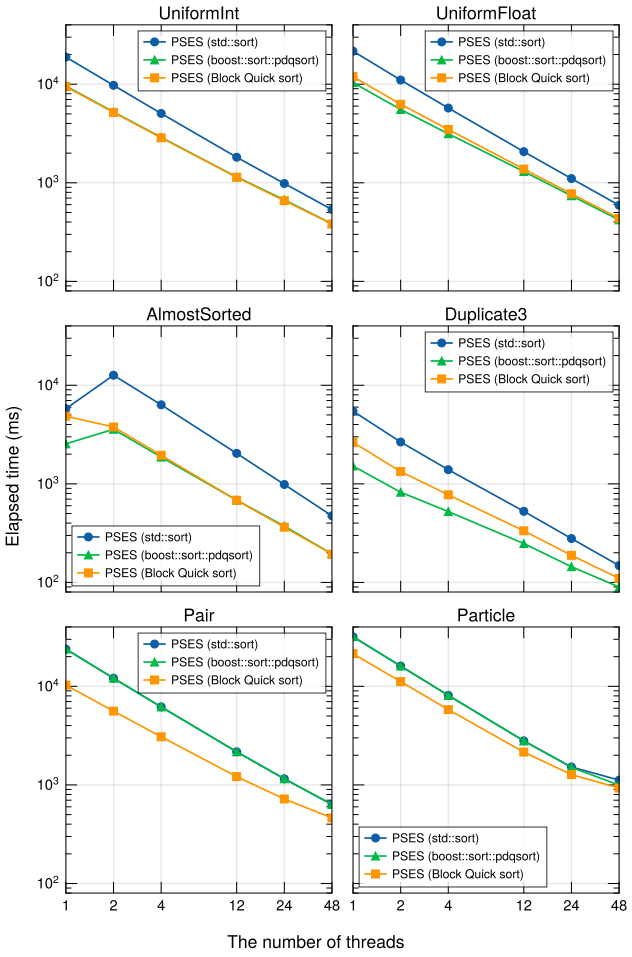


図 5: ブロック内ソートに使用する逐次ソートアルゴリズムを変えた時の PSES のソート時間（$N = 10^8$）．逐次ソートとして libstdc++ の std::sort，Boost ライブラリ の boost::sort::pdqsort，Block Quick sort を使う場合を比較する．パーティションのマージには selection tree を使う．

いずれの入力に対しても，本研究で実装した selection tree を使ってマージする場合が最も高速だった．std::sort を使う場合，AlmostSorted 以外の数列に対して，24 スレッド以上のソート時間は selection tree を使う場合とあまり変わらない．これは，外部のデータ構造を使わずキャッシュ効率に優れるためと考えられる．一方，要素の移動回数が多くなるため，Pair や Particle のように要素のデータサイズが大きい場合には selection tree を使う場合より遅くなる．std::priority_queue を使う場合，4 スレッド以下でスレッド数を増やしてもソート時間が短くならず，数列のソートは std::sort を使う場合よりも遅くなった．これは，標準ライブラリのコンテナが「富岳」上で効率的に実行することを想定した実装になっていないためと考えられる．

5. おわりに

本研究では，「富岳」上で実行可能な 2 種類のスレッド

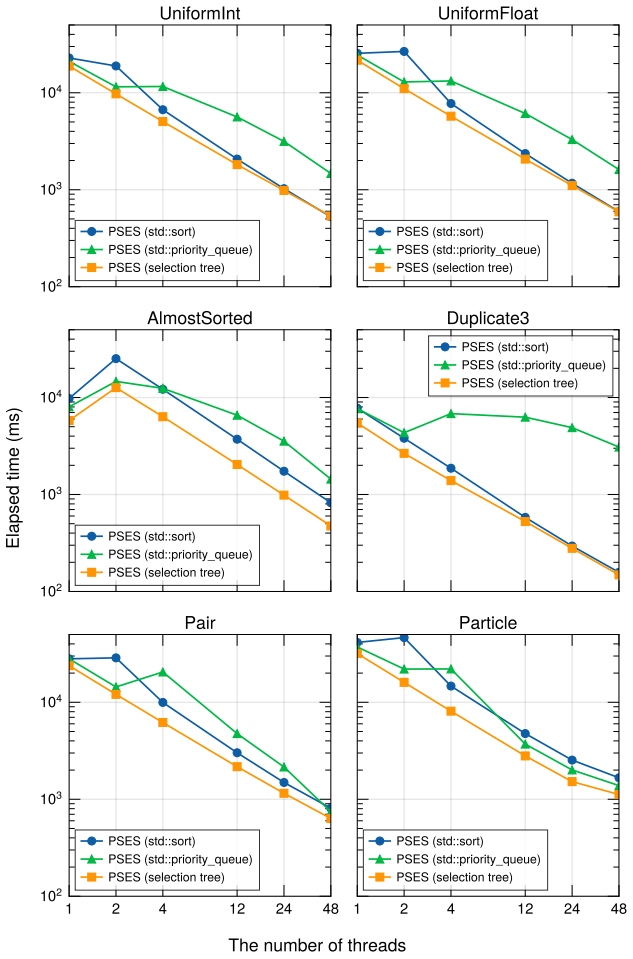


図 6: パーティションのマージ方法を変えた時の PSES のソート時間（$N = 10^8$）．libstdc++ の std::priority_queue を使う場合と selection tree を使う場合に加えて，マージの代わりに std::sort でソートする場合を比較する．ブロック内ソートには std::sort を使う．

並列ソートプログラムを実装した．ひとつめは，入力列を複数のブロックに分けてそれぞれソートし，各ブロックから等間隔にサンプリングを行いピボットを選ぶ．ふたつめは，ピボットの選び方が異なり，全てのパーティションの要素数が等しくなるようなピボットを二分探索によって選ぶ．それぞれに対し，逐次ソートやマージのアルゴリズムを変えて性能を比較した．その結果，条件分岐を減らすことで高速化されたクイックソート（Block Quick sort）を逐次ソートに，selection tree をマージに用いたふたつめのサンプルソートが，さまざまな入力に対し安定して高い速度性能および並列化性能を示した．

今後の課題として，in-place な並列ソートの実装が挙げられる．本研究で実装したスレッド並列ソートプログラムは，いずれも入力列と出力列がメモリ上で異なる領域にあることを前提にしている．すなわち入力列と同じ大きさの出力列をメモリ上に確保しておく必要があり，メモリ使用量が大きい．1 計算コアあたりのメモリ容量が小さい「富岳」上で実行するソートプログラムならば，in-place であることが望ましい場合もあると考えられる．

謝辞 本研究は，「富岳」成果創出加速プログラム「宇宙の構造形成と進化から惑星表層環境変動までの統一的描像の構築」（JPMXP1020200109）および計算基礎科学連携拠点（JICFuS）の元で実施した．また，本研究は千葉大学国際高等研究基幹研究支援プログラムの支援を受けて実施した．

参考文献

[1] Edelkamp, S. and Weiß, A.: BlockQuicksort: Avoiding Branch Mispredictions in Quicksort, *ACM J. Exp. Algorithmics*, Vol. 24 (2019).
[2] Hoare, C. A. R.: Quicksort, *The Computer Journal*, Vol. 5, No. 1, pp. 10–16 (1962).
[3] Knuth, D. E.: *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*, Addison Wesley Longman Publishing Co., Inc., USA (1998).
[4] MUSSER, D. R.: Introspective Sorting and Selection Algorithms, *Software: Practice and Experience*, Vol. 27, No. 8, pp. 983–993 (1997).
[5] Peters, O. R. L.: Pattern-defeating Quicksort, *CoRR*, Vol. abs/2106.05123 (2021).
[6] Shi, H. and Schaeffer, J.: Parallel sorting by regular sampling, *Journal of Parallel and Distributed Computing*, Vol. 14, No. 4, pp. 361–372 (1992).
[7] Siebert, C. and Wolf, F. G. E.: A scalable parallel sorting algorithm using exact splitting, Technical report, Aachen (2011).
[8] Singler, J. and Konsik, B.: The GNU Libstdc++ Parallel Mode: Software Engineering Considerations, *Proceedings of the 1st International Workshop on Multicore Software Engineering*, IWMSE '08, New York, NY, USA, Association for Computing Machinery, p. 15–22 (2008).
[9] Singler, J., Sanders, P. and Putze, F.: MCSTL: The Multi-Core Standard Template Library, *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, Euro-Par'07, Berlin, Heidelberg, Springer-Verlag, p. 682–694 (2007).