Map Sort: マルチコアプロセッサに向けたスケーラブルなソートアルゴリズム

枝廣 正人† 山下 慶子‡

† NEC システムデバイス研究所 〒229-1198 神奈川県相模原市下九沢 1120
‡ NEC ソリューション開発研究本部 〒305-8501 茨城県つくば市御幸が丘 34
E-mail: † eda@bp.jp.nec.com

あらまし マルチコア向けの並列ソートアルゴリズムMap Sortを提案する。今後単体CPUの性能向上が鈍化し、プロセッサがマルチコアによって性能向上する時代では、並列対応されていないソフトウェアは計算機が進歩しても性能は向上しない。従って単体CPUでは従来と同等処理時間で、かつ並列CPUではスケーラブルに性能向上するようなアルゴリズムが必須となるが、我々はそれをスケーラブルアルゴリズムとよんでいる。本論文ではソート問題を取り上げ、新しいスケーラブルアルゴリズムMap Sortを提案する。Map Sortの時間に関する計算複雑度は*N*個のデータ、*P*台のCPUで$O((N/P)\log N)$であり、単体CPU上での下界値$O(N\log N)$の$(1/P)$である。また計算機実験の結果、単体CPU上のクイックソートと比較し、単体CPUでは同等性能、4CPUでは3倍の性能向上であることが示された。

キーワード マルチコア、並列、ソート、アルゴリズム

Map Sort: A Scalable Sorting Algorithm for Multi-Core Processors

Masato EDAHIRO† and Yoshiko YAMASHITA‡

† System Devices Res. Labs, NEC Corporation, 1120 Shimokuzawa, Sagamihara, 229-1198 Japan
‡ Solutions Dev. Labs, NEC Corporation, 34 Miyukigaoka, Tsukuba, 305-8501 Japan
E-mail: † eda@bp.jp.nec.com

Abstract A sorting algorithm, called *Map Sort*, is proposed for multi-core processors. The proposed algorithm uses a *map* from subsets of input data to intervals in data range. Using the map, our algorithm is executed in parallel with multiple CPUs and has $O((N/P)\log N)$ time complexity for N data and P CPUs. Experimental results show that, in comparison with quick sort on a single CPU, processing time of Map Sort is comparable on a single CPU and three times faster on four CPUs.

Keyword Multi-Core, Parallelism, Sorting, Algorithm

1. はじめに

今後の半導体デバイスプロセスにおいては、微細化は進むものの動作周波数を上げつつ電力を下げることがますます難しくなってきており、装置の電力制約を考慮すると単体CPUの動作周波数向上はあまり期待できない。そのため現在パソコンや高性能組込システムにおいて始まっているように[1, 2]、単体CPUの動作周波数よりはマルチコア化など並列により性能を高めていく傾向が強まると考えられている。従って並列化されていないソフトウェアは装置の進化に合わせた性能向上は従来ほど期待できず、並列化されたソフトウェアのみが性能向上していくことになる。

このような時代において、我々は以下の三点の性質を持つアルゴリズムが重要であると考え、*スケーラブルなアルゴリズム*とよんでいる。

1）P台のCPUを用いたときに時間に関する計算複雑度が(1/P)

2）単体CPUでの実行時間は、単体CPU向けに最適化された従来のアルゴリズムと同等

3）複数CPUでの並列性能向上は高いことが望まれる

メディア処理や科学技術計算においては本質的に並列性を含むものが多く、並列アルゴリズムの研究も進んでいるが、いわゆる基本アルゴリズムや、経路探索、辞書サーチなどのグラフ・ネットワークアルゴリズム[3]は並列性が抽出しにくいものが多い。

ソート処理はそのような基本アルゴリズムの中で代表的な問題である。これまで多くの並列ソートアルゴリズムが提案されてきているが[4]、上記の意味で優れたスケーブルアルゴリズムは提案されていない。クイックソートは単体プロセッサ向けでは最も効率の良いアルゴリズムと言われているが、並列化すると再帰的な分割の最初の数回は十分な並列性能が出せない。マージソートは理論上最も良いアルゴリズムであるが、近年の計算機システムで用いるとメモリ上のコピー回数が多いためクイックソートよりも実行時間性能が良くない。またソート済み配列のマージ処理を複数CPUで行うアルゴリズムが複雑であることも実行時間を遅くする原因となっている。並列計算機上での計算複雑度が優れたアルゴリズムについてもバイトニックソート、Odd-Evenマージソートなど数多くの提案があるが、$O(N)$台のCPUを仮定しており、現実的なCPU数ではCPU数に見合った並列実行性能は見込めない。

本論文では、スケーラブルなソートアルゴリズム*Map Sort*を提案する。Map Sortでは入力データ配列を複数部分配列に分割し、データが分布する範囲を複数区間に分割する。そして複数部分配列から複数区間へのマップ(*Map*)を考える。マップの計算、複数部分配列から複数区間へのデータコピー、複数区間ごとのソート処理をそれぞれ並列に行うことができ、結果として$O((N/P) \log N)$の時間複雑度を達成する。

計算機実験の結果、Map Sort は単体 CPU ではクイックソートと同等性能を達成し、4 CPU では単体 CPU 上のクイックソートと比較して 3 倍の性能向上を達成できることがわかった。

2. 用語

N個のデータからなる全順序集合Sがあたえられたとき、*ソート*とは昇順(または降順)に並べ替える処理である。本論文では、ソート処理をP個のコアを持つSMP (Symmetric Parallel Processing)、共有メモリ型のマルチ(コア)プロセッサで解くことを考える。またPはNよりもはるかに小さいと仮定する。

提案するアルゴリズムでは、与えられたデータ集合SをM個の部分集合$\{S_1, ..., S_M\}$に分割し、すべてのデータを含む範囲をL個の区間に分割する。MおよびLはm、lを小さい定数とするとき$M=mP$、$L=lP$のように選ぶことが多い。実際のプログラムでは入力データ配列をM個の部分配列に分割すると都合がよい。また、出力データ配列は、ソート後の状態を考えると、L個の区間に対応してL個の部分配列に分割されることになる。提案アルゴリズムでは出力データ配列のL個の部分配列のそれぞれがさらにM個の部分配列に分割されるが、それにより入力データ配列において異なる部分配列に属するデータは、たとえ同じ区間に属していたとしても並行して転送することができる。この並行データ転送のために提案アルゴリズムでは二つの2次元($M \times L$)配列を用いる。まず入力部分配列S_iに含まれ、j番目の区間に属するデータの集合をD_{ij}とする。このとき***カウント2次元配列*** $C=\{c_{ij}\}$を $c_{ij}=|D_{ij}|$で定義する。また***マップ2次元配列*** $Q=\{q_{ij}\}$を D_{ij}を保持すべき出力データ配列上の位置として定義する。

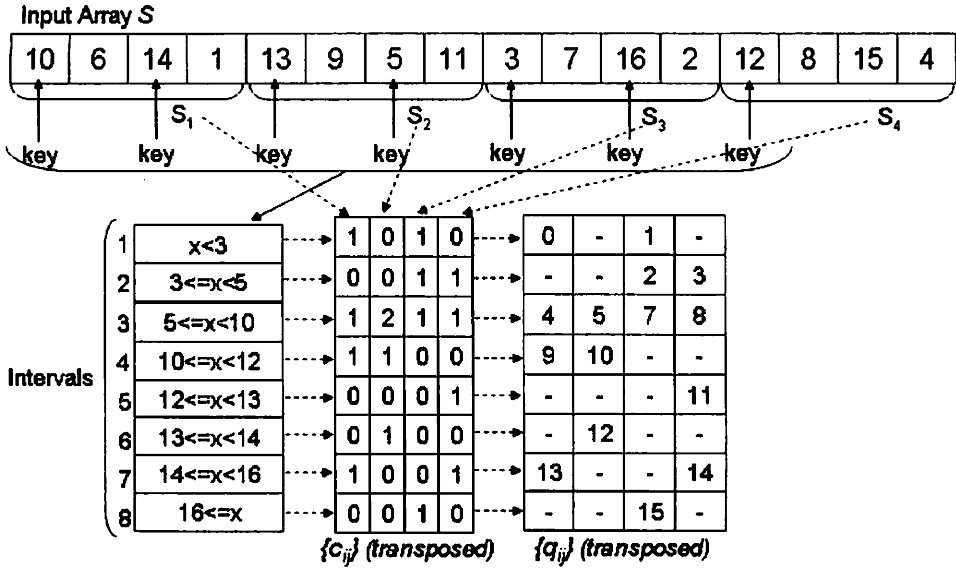
3. Map Sort

Map Sortの概要は下記のようになる。

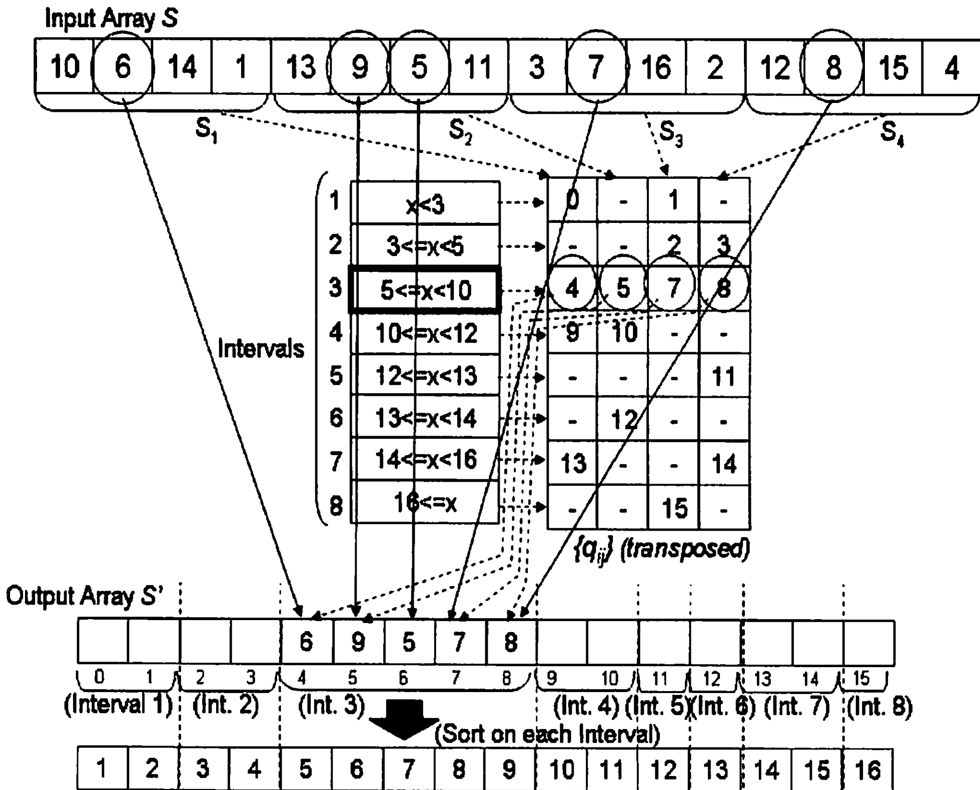
> Algorithm *MapSort*
> Input: *S[0..N-1]*
> Output: *S'[0..N-1]* (sorted array)
>
> Step 1: 入力データ S から $L-1$ 個の基準値を計算し、それらの基準値を境界とする L 個の区間を計算
> Step 2: データ $d \in S_i$ に対し、d を含む区間(j 番目とする)を求め、カウント2次元配列 c_{ij} の値を更新(部分集合 S_i ごとに並行処理可能)
> Step 3: P 個の CPU により Q を計算
> Step 4: データ $d \in S_i$ に対し、d を含む区間(j 番目とする)を求め、出力データ配列 S'上の位置 $q_{ij} \in Q$ に d をコピーし、q_{ij} に 1 を加算(部分集合 S_i ごとに並行処理可能)
> Step 5: 出力データ配列 S'上の各区間に対応する部分配列をソート(各区間のソートはクイックソートなど単体 CPU 向けに最適化されたソートアルゴリズムを用い、区間ごとに並行処理可能)

Fig. 1 (a) に示すような16データの例を用いて提案アルゴリズムを説明する。ここでM=4、L=8とする。Step 1ではSから7 (=L-1)個の基準値(key)を選び8個の区間(Interval)を作る。Step 2では入力データSを4(= M)分割し、各部分配列S_iに対し各区間に入るデータ数c_{ij}を数え上げる。Step 2の処理は各部分配列S_iごとに並行処理可能である。

そしてStep 3ではq_{ij}を次のように計算する。c_{ij}は部分集合S_iに属しj番目の区間に含まれるデータ数であるため、j番目の区間に含まれるデータの総和は $\sum_{i'=1}^{M} c_{i'j}$



(a) Input data S, $(L-1)$ keys, L intervals, c_{ij} and q_{ij} for $M=4$, $L=8$. No value for q_{ij} when $c_{ij}=0$.



(b) Data in S_i within j-th interval is moved to the position of output array S' indicated by q_{ij}. Then, each sub-array of the output array associated with intervals is sorted.

Fig. 1. Proposed Algorithm.

となる。Fig. 1の例では、1番目の区間に2個、2番目の区間に2個のデータが存在する。従って出力データ配列上でj番目の区間に含まれるデータは $\sum_{j'=1}^{j-1}\sum_{i'=1}^{M} c_{i'j'}$ から開始されることになる。Fig. 1の例では1番目の区間を出力データ配列の位置0、2番目の区間を位置2、3番目の区間を位置4から開始すればよい。これよりj番目の区間に含まれ部分集合S_iに属するデータを $\sum_{j'=1}^{j-1}\sum_{i'=1}^{M} c_{i'j'} + \sum_{i'=1}^{i-1} c_{i'j}$ から開始すると重なりなくデータ転送でき、これをq_{ij}とする。Fig. 1の例では上述のように3番目の区間は位置4から開始するためq_{13}=4となり、S_1に含まれるデータはここから開始する。そしてc_{13}=1であるためq_{23}=5となり、S_2に含まれるデータは位置5から開始すればよい。なお図では、c_{ij}=0のときにはS_iに含まれj番目の区間に属するデータが存在しないためq_{ij}を記載していない。Step 3の処理はMxL行列上の集計処理であり、並行処理が可能である。ただしM、Lの値が小さいため単体プロセッサで実行した方が高速になる場合も多い。

次にStep 4において、入力データ配列S上のデータをすべて出力データ配列に転送する。例えば部分配列S_iに属するデータがj番目の区間に含まれるとき、出力データ配列上の位置q_{ij}にコピーすると共に (Fig. 1(b))、q_{ij}に1を加算する。このときStep 3の計算方法により、q_{ij}と$q_{i+1,j}$ (または$q_{1,j-1}$)の間のデータ数がc_{ij}であることから、すべてのデータは重なることなくコピーできることが重要である。Fig. 1の例では、上述のように3番目の区間に含まれるデータは、S_1は位置4から、S_2は位置5から、S_3は位置7から、S_4は位置8からデータ転送することにより重なりなく転送できる。このためStep 4についても各部分配列S_iごとに並行処理可能である。

さて、Step 4の終了後は、出力データ配列は各区間に対応する部分配列に分けられることがわかる。よってStep 5においてそれぞれの区間に対応する部分配列に対して複数プロセッサを用いて並行してソートすることにより、全体のソートが完了する(Fig. 1 (b))。

このアルゴリズムにおいては基準値の選び方が重要である。なぜならば基準値の選び方によりStep 4の結果生成されるL個の部分配列に偏りができ、Step 5を並列実行した場合の負荷バランスが悪くなるためである。そのため、例えばL個よりも多くのサンプルを取得し、簡単な統計処理を行った上で基準値を計算する方が結果的に高速になる場合が多い。また、特に元のデータが一様分布、正規分布など既知の場合には効果が大きくなる。

次に計算複雑度について考える。L、Mを$O(P)$とする。Pは実際上定数であるが$O(\sqrt{N})$とする。空間複雑度は、出力データ配列に$O(N)$、マップ情報に$O(P^2)$であるため、全体で$O(N)$となる。

時間複雑度は、Step 1は選択アルゴリズムにも依存するが $O(P)$個のデータを選びソートして区間を設定するため $O(P \log P)$となる。Step 2において各部分配列は $O(N/P)$の長さであり、各データに対して区間を計算する処理に $O(\log P)$必要であるため、P台のCPUを用いて $O((N/P) \log P)$となる。Step 3におけるマップデータ構造の作成は、P台のCPUを用いて$O(P)$でできる。Step 4もStep 2と同様の処理になるため、P台のCPUを用いて $O((N/P) \log P)$となる。最後にStep 5は、各区間に対応する部分配列が平均的に $O(N/P)$の長さであるため、ソート処理は $O((N/P) \log (N/P))$となるが、Pのオーダーを考えると $O((N/P) \log N)$となる。従って全体の時間複雑度は $O((N/P) \log N)$となり、ソート処理の時間複雑度 $O(N \log N)$の$(1/P)$となる。

4. 計算機実験

提案アルゴリズムを評価するため、OpenMPディレクティブを用いたC言語を用いてプログラム化し、Intelクアッドコアプロセッサ QX6700 (2.66GHz)を用いたパソコン上で計算機実験を行った。各実験においては乱数により生成された整数データ集合を10通り用意し、平均を取ることにより実行時間を測定した。今回の実験ではデータ数は10^4-10^7とした。また比較のためクイックソートもプログラム化した。単体CPU上のクイックソートは[3]に基づくプログラムとし、複数CPUの場合には再帰的な配列分割をスレッド化することにより並列化を行った。なおスレッドを数多く発生させるとオーバーヘッドが大きくなり、かえって実行時間が遅くなるため、最大スレッド数を制限するようにした。

Fig. 2はM=L=32における提案アルゴリズムとクイックソートの実行時間を示している。Fig. 3ではPを変化させたときのMap Sortの実行時間を計測した。Map Sortは、2PE (Processing Element)の時に約1.8倍、4PEでは約3.5倍の並列性能を達成した。Fig. 4ではクイックソートとの比較を示した。単体CPU上のクイックソートと比較して、並列クイックソートは4PEで1.6倍であるのに対し、提案アルゴリズムは4PEで約3倍の性能向上を達成した。またMap Sortでは、パラメタM、Lに対し、単体CPUでの実行速度と並列加速性能がトレ

ードオフの関係にあるが、Fig. 5はLの値に対する変化を示している。Lの値が小さいとき、単体CPUでの実行時間は短いが並列加速は悪くなる。Lの値が大きいと並列加速は良くなってくるが、オーバーヘッドが大きくなるため単体CPUでの実行時間は遅くなる。また、さらにLの値を大きくすると、単体CPU、複数CPUのどちらの性能も悪くなる。この結果より、Pの値に応じてM、Lの値を変えることにより優れた性能が達成できることがわかる。例えばP=1のときL=1、P=2のときL=16、P=4のときL=32とすることにより、単体CPUではクイックソートと同等性能、4 CPUでは3倍の性能を達成し、スケーラブルなアルゴリズムとなることがわかる。

5. まとめ

本論文では、新しい並列ソートアルゴリズムMap Sortを提案した。このアルゴリズムはマルチコア上でスケーラブルな性能を達成する。P台のCPUを用いたとき単体CPU上のソートアルゴリズムと比較して時間複雑度が$(1/P)$となり$O((N/P) \log N)$となる。また計算機実験の結果、単体CPU上のクイックソートと比較し、単体CPUでの性能は同等で、複数CPUでは4 CPUで3倍の性能を達成できることがわかった。

文献

[1] Intel, *Multi-Core Processors. Making the Move to Quad-Core and Beyond*, White Paper, 2006.

[2] P. Middleton, "A New ARMv6 Symmetric Multiprocessing Core", *Embedded Processor Forum*, 2004.

[3] R. Sedgewick, *Algorithms in C*, Addison-Wesley, Reading, MA, 1990.

[4] D. Bitton, D. J. DeWitt, D. K. Hsiao, and J. Menon, "A Taxonomy of Parallel Sorting," *ACM Computing Surveys*, Vol. 16 (1984), No. 3, pp.287-318.

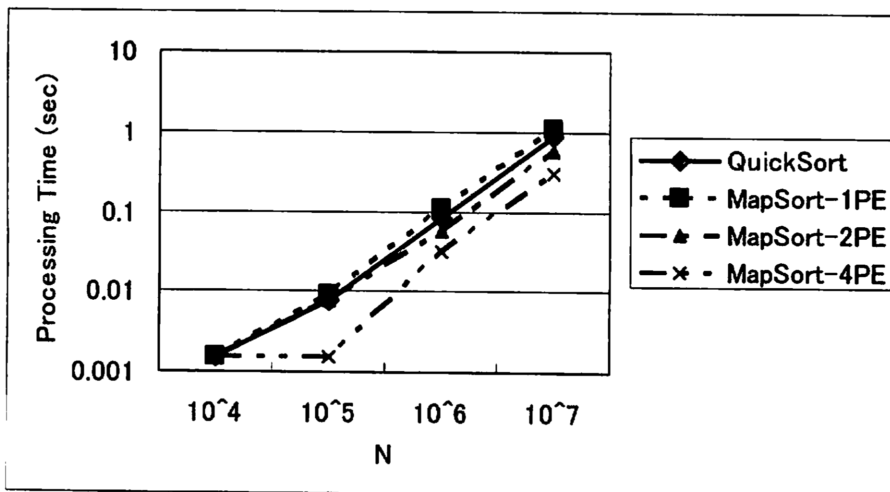


Fig. 2. Processing Time for Quick Sort on 1 PE and Map Sort

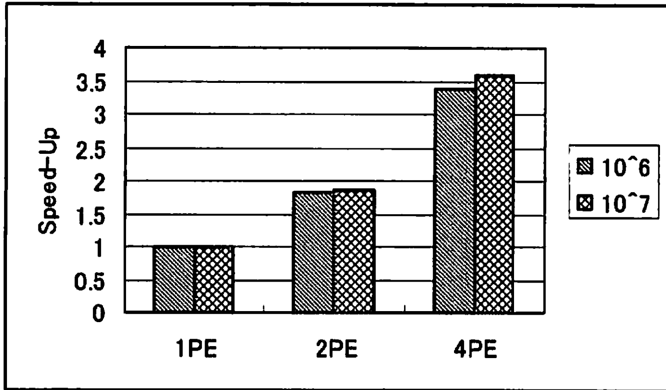


Fig. 3. Speed-Up for Map Sort (Normalized by Map Sort-1PE)

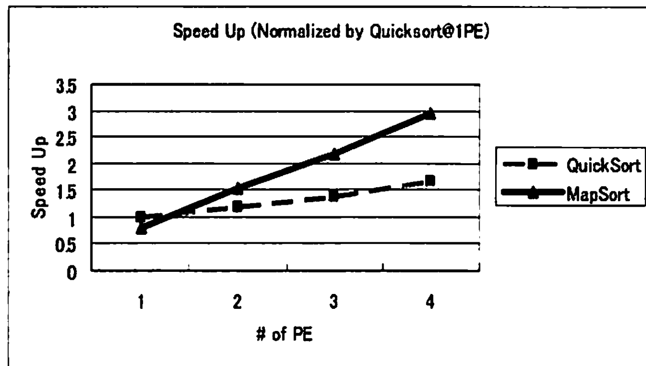


Fig. 4. Speed-Up for Quick Sort and Map Sort
(Normalized by Quick Sort-1PE, $N=10^7$)

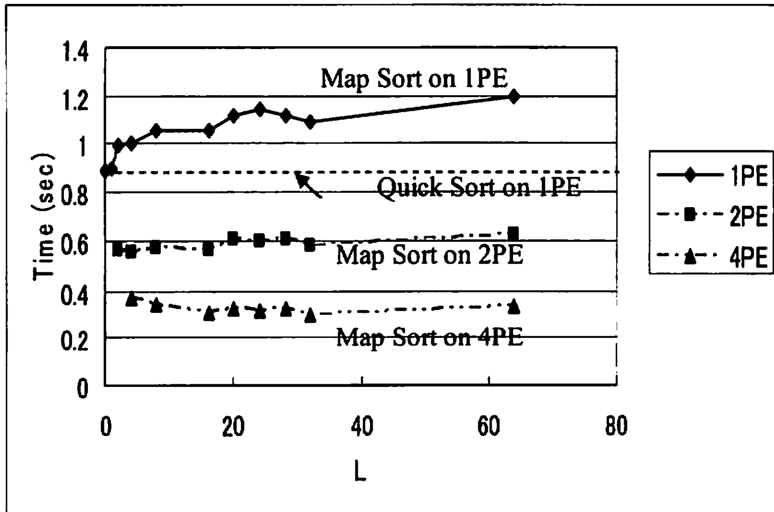


Fig. 5. Processing Time for several L values ($N=10^7$)