

分散メモリ型ベクトル並列計算機上での 高速整数ソーティングアルゴリズムの実装

横山 栄二[†] 安岡 孝一^{††}
岡部 寿男[†] 金澤 正憲^{†††}

分散メモリ型ベクトル並列計算機上での高速なソーティング手法について述べる。本手法は、バケツソートを基本とし、ヒストグラムの計算は、ベクトル実行に優れたアルゴリズムを用いて行う。並列化は、まず各プロセッサがローカルヒストグラムを計算し、次にローカルヒストグラムをキー空間方向に再分散してグローバルヒストグラムを得る。それに基づいてトータルランクを計算し、各キーの順位を得る。ヒストグラムの転送に圧縮されたデータ形式を採用することで、プロセッサ間通信に要する時間を短縮した。以上をデータ並列型言語 VPP Fortran で実装し、分散メモリ型ベクトル並列計算機 Fujitsu VPP800 上で NPB (NAS Parallel Benchmarks) の中の IS (Integer Sort) ベンチマークを用いて評価した。Class C のデータに対して 32 CPU で 106.6 msec の実行時間となり、高速であることが確認された。

Implementation of a Fast Integer Sorting Algorithm on Distributed-memory Vector-parallel Supercomputers

EIJI YOKOYAMA,[†] KOICHI YASUOKA,^{††} YASUO OKABE[†]
and MASANORI KANAZAWA^{†††}

We propose a fast sorting algorithm for VPP distributed-memory parallel vector supercomputers. Our algorithm is based on the bucket sort, and utilizes a vector algorithm for local sorting on each vector processor. Bucket sort is parallelized as follows. First each processor computes the local histogram. Next the local histograms are redistributed blockwise with the key space. Then the processors compute the global histogram in parallel, and get the total rank for each key. We have also made an improvement in transferring data among processors by sending compressed histograms. We have implemented a library code in on VPP Fortran data-parallel language, and have evaluated its performance on Fujitsu VPP 800 in NAS Parallel Benchmark Integer Sort scheme. The time for the Class C instance on VPP800, 32 CPUs is 0.1066 sec. This is much faster than former benchmark records.

1. はじめに

ソーティングは、計算機科学分野における最も基本的な処理の1つであり、データベース、画像処理など様々な分野で応用されている。そのため、ソーティングを高速に実行するアルゴリズムの重要性は非常に高くなっており、理論面からも実用面からも多くの研究が行われてきている^{1),2)}。

本論文では、分散メモリ型並列ベクトル計算機上で

の高速な整数ソーティングアルゴリズムの実装について述べる。採用したアルゴリズムは、ベクトル計算機において高速なアルゴリズムであるバケツソート³⁾を基本としている。バケツソートは、まず各キー値のヒストグラムを計算し、次に各キー値に対するヒストグラムの累計を計算して、それを基にキーに順位をつける。ヒストグラムの計算はそのままではベクトル化できないが、ここでは最も高速なベクトルアルゴリズムとして知られる Retry アルゴリズム⁴⁾を採用した。

バケツソートの並列化では、まず各プロセッサにおけるヒストグラム（ローカルヒストグラム）の計算を行った後、その総和を計算することになる。ここでは、各プロセッサでの処理は文献⁴⁾などで用いられている共有メモリ型並列計算機用アルゴリズムをそのまま分散メモリ型計算機へ適用したうえで、必要となるデー

[†] 京都大学情報学研究科
Graduate School of Informatics, Kyoto University

^{††} 京都大学人文科学研究科
Institute for Research in Humanities, Kyoto University

^{†††} 京都大学大型計算機センター
Data Processing Center, Kyoto University

タ転送時間を短くするためにヒストグラムの転送に圧縮されたデータ形式を採用することで、プロセッサ間通信に要する時間を短縮した。

以上のアルゴリズムを HPF⁵⁾ 類似のデータ並列型言語である VPP Fortran^{6),7)} で実装し、NAS Parallel Benchmarks (NPB)⁸⁾ の中のベンチマーク IS (Integer Sort) を用いて、分散メモリ型ベクトル計算機 Fujitsu VPP800 上で評価を行った。その結果、Class C のデータに対して 32 CPU で 106.6 msec の実行時間となり、高速に実装できていることが確認された。

以下、2 章で整数ソーティング問題の定義を行い、3 章でバケツソートの概要について述べる。4 章でバケツソートの並列化の詳細と、我々の提案する通信時間の圧縮手法について述べる。5 章でそれらの手法の VPP500 上での実装について述べ、6 章で性能評価と考察を行う。

2. 整数ソーティングの問題の定義

本論文で議論の対象とする整数ソーティングを、NAS Parallel Benchmark (NPB) の中のベンチマーク IS に従って次のように定義する。

与えられた N 個のキー $\{K_i \mid i = 0, 1, \dots, N-1\}$ のソーティングとは、 $K_i \leq K_{i+1} \leq K_{i+2} \dots$ を満たすように 1 列に並べ替える処理のことである。ここで、 K_i は、 $0 \leq K_i \leq \text{MAXKEY} - 1$ を満たす整数、MAXKEY はキー値のとりうる種類の数である。ランクとは、キーの集合がソートされたときの各キーに対するインデックスであり、ランキングとは、すべてのキーにランクをつけることである。つまり、初期のソートされていない N 個のキー $\{K_i \mid i = 0, 1, \dots, N-1\}$ のランクが、 $r(0), r(1), \dots, r(N-1)$ だとすると、 $K_{r(0)}, K_{r(1)}, \dots, K_{r(N-1)}$ のような順番に並べ替えられたとき、このキー列はソーティングされたことになる。ソーティングにおいては同じ値を持つキーにもランクをつける（つまり、 $i \neq j$ ならば、 $r(i) \neq r(j)$ である）必要がある。ここでは、同じ値を持つキー間での付け方は任意とする。

本論文では、ランキング、すなわち各キーに対するランクを求めるのに必要な計算時間で性能を評価する。NPB 2 IS (逐次実装 (2.3-serial) および MPI による並列実装 (2.3b2)) では（各キーではなく）各キー値に対してその値を持つ最初のキーのランクのみを計算するようになっているが¹⁾、ここでは NPB 1 での本来の定義に従いすべてのキーに対して異なるランクを割り当てるところまで要求する。

分散メモリ型計算機の場合には、計算終了時点で解である各キーに対するランクがどのプロセッサ上のメモリに保持されているかも定める必要がある。ここでは各キーに対して、そのキーが最初に配置されていたプロセッサ上にそのランクを返すものとする^{*}。

3. バケツソート

本章では、本アルゴリズムの基本となる、バケツソート³⁾について述べる。以下に、単一プロセッサでのバケツソートの概要を述べる。

- ① **ヒストグラムの生成**：初期のソートされていないキー列を走査し、 $0, 1, \dots, \text{MAXKEY} - 1$ の各値を持つキーの個数をカウントし、キー値のヒストグラム *keyden* を生成する。このとき、ランクの計算の際に用いるため、同じ値を持つキー間での出現順位の情報も保存しておく。
- ② **累計の計算**：①で生成されたヒストグラム (*keyden*) の累計を計算する。つまり、値 k に対する *keyden* の累計 ($\text{running_sum}(k) = \text{keyden}(0) + \text{keyden}(1) + \dots + \text{keyden}(k)$) を $k = 0, 1, \dots, \text{MAXKEY} - 1$ の各値について計算する。
- ③ **ランクの計算**：①で記憶しておいた同じ値を持つキー間での出現順位の情報 $\{\text{pos}(i) \mid i = 0, 1, \dots, N-1\}$ と、②で生成された累計 (running_sum) を基に、各キーのランクを計算する。値 k を持つ $\text{key}(i) (i = 0, 1, \dots, N-1)$ のランクを $\text{running_sum}(k) - \text{pos}(i)$ と計算する。

バケツソートのアルゴリズムを図 1 に、実行の様子を図 2 に示す。

4. バケツソートの並列化

本章では、分散メモリ型並列計算機上でのバケツソートの並列化手法について述べる。はじめに、アルゴリズムの概要について述べ、その後、各手順の詳細について述べる。

4.1 アルゴリズムの概要

分散メモリ型並列計算機上でのバケツソートの手順を示す。

- ① **ローカルヒストグラムの計算**：各プロセッサごとに、初期状態で割り当てられたキー列 $\{\text{key}(i) \mid i = 0, 1, \dots, N/P - 1\}$ のヒス

^{*} 各キーに対して同じプロセッサ上に対応するデータが格納されている状況を想定している。

```

!ヒストグラムの生成
do i=1, N
  k = key(i)
  keyden(k) = keyden(k) + 1
  pos(i) = keyden(k) !出現順位の保存
end do

!累計の計算
running_sum(0)=keyden(0)
do k=1, MAXKEY-1
  running_sum(k) = keyden(k) + running_sum(k-1)
end do

!ランクの計算
do i=1, N
  k = key(i)
  rank(i) = running_sum(k) - pos(i)
end do

```

図 1 バケツソートのアルゴリズム

Fig. 1 The algorithm of bucket sort.

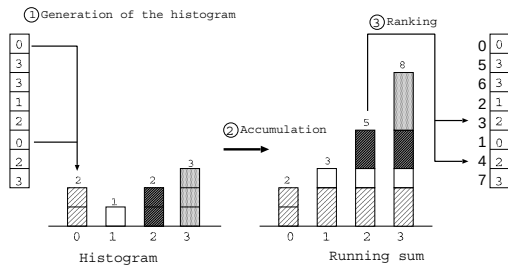


図 2 バケツソートの実行の様子

Fig. 2 Execution of bucket sort.

- トグラム $local_keyden(k, j)$, $k = 0, 1, \dots, MAXKEY - 1$, $j = 0, 1, \dots, P - 1$ (以下, **ローカルヒストグラム**と呼ぶ)を計算する. ここで, P はプロセッサ台数とする (4.2 節参照).
- ② **ローカルヒストグラムの部分和の計算:** ①で生成されたローカルヒストグラムから, **ローカルヒストグラムの部分和** $cumulative_keyden(k, j) = local_keyden(k, 0) + \dots + local_keyden(k, j)$ を計算する. その後, 各プロセッサは, ランクの計算の際に用いるため, 必要な部分和を保持する. このとき同時に, 全体のキー列 $\{K_i \mid i = 0, 1, \dots, N - 1\}$ のヒストグラム $global_keyden = cumulative_keyden(k, P - 1)$ (以下, **グローバルヒストグラム**と呼ぶ)も求める (4.3 節参照).
- ③ **トータルランクの計算:** ②で求めたグローバルヒストグラムから, **トータルランク**を計算する. ここで, トータルランク $total_rank(k)$ とは, 各キー値 $k = 0, 1, \dots, MAXKEY - 1$ についてグ

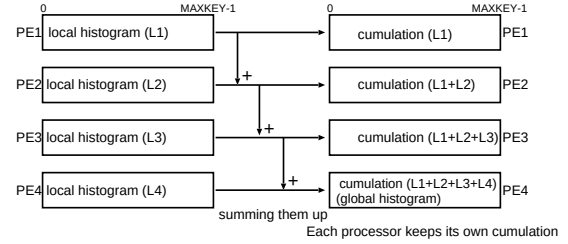


図 3 ローカルヒストグラムの部分和 (PE 数 4)

Fig. 3 Cumulation of the local histogram (#PE is 4).

ローカルヒストグラム ($global_keyden$) の k までの累計 $total_rank(k) = global_keyden(0) + global_keyden(1) + \dots + global_keyden(k)$ を計算したものである (4.5 節参照).

- ④ **ランクの計算:** ②, ③で生成されたトータルランクおよび, ローカルヒストグラムの部分和を基に, 各キーのランクを計算する (4.6 節参照).

4.2 ローカルヒストグラムの生成

本アルゴリズムでは, Retry アルゴリズム⁴⁾を用いて, 各プロセッサごとにローカルヒストグラムの生成を行う. Retry アルゴリズムは, バケツソートにおけるヒストグラムの生成処理を高速にベクトル実行するためのアルゴリズムである. この部分は, 完全にローカルに計算可能であり, 各プロセッサあたりの演算量は $O(N/P)$ で, 並列化すればプロセッサ台数に比例する効果が得られる.

4.3 ローカルヒストグラムの部分和の計算

ローカルヒストグラムの部分和は, 後述のランクの計算 (4.6 節参照) の際に用いられる. 各プロセッサが保持すべき部分和は, 自分と, 自分より小さいインデックスのプロセッサの生成したローカルヒストグラムの和となる. 部分和の概念を図 3 に示す.

部分和の計算は, プロセッサ間で二分木上に行う方法⁹⁾と, 並列化の方向を変更して各キー値に対しては 1 台のプロセッサが計算するようにする方法^{4), 9), 10)}とがあり, 共有メモリ型ベクトル計算機では後者が効率良く実装できる⁴⁾. 分散メモリ型並列計算機では, 並列化の方向を変更する方法は, 初めにデータの再分散が必要となる一方総和計算の途中ではデータ転送が不要となるため, どちらが有利かは一概にはいえない⁹⁾.

以下では, 並列化の方向を変更する方法に基づいて分散メモリ型並列計算機上で実際に部分和の計算を行うアルゴリズムの詳細について述べる. 部分和を求めるために用いる手法を図 4 に示す.

- ① **ローカルヒストグラムの再分散:** ローカルヒストグラム $local_keyden(k, j)$ を, キーの初期配

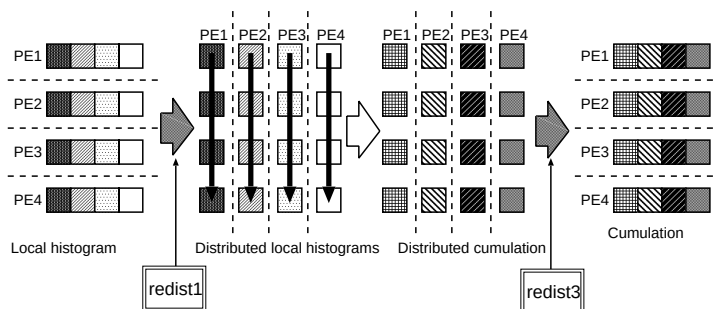


図 4 部分和の計算の流れ (PE 数 4)

Fig. 4 Parallelization of computing cumulation (#PE is 4).

置に沿った分散 (添字 j に関するブロック分散) からキー空間に沿った分散 (添字 k に関するブロック分散) へ再分散する. すなわち各プロセッサがローカルヒストグラムをプロセッサ数で分割し, 他のプロセッサへ, ローカルヒストグラムのそれぞれの計算担当範囲を通信する転置処理を行う (図 4 の redist 1).

- ② **ローカルヒストグラムの部分和の計算**: 各プロセッサは, ①で受信したローカルヒストグラムの一部を用いて, 自分が担当する部分和の計算を行う.
- ③ **計算後の部分和の再分散**: 計算後の部分 $cumulative_sum(k, j)$ を, キー空間に沿った分散 (添字 k に関するブロック分散) からキーの初期配置に沿った分散 (添字 j に関するブロック分散) へ再分散する. すなわち各プロセッサは①と同様に他のプロセッサと通信を行い, ②で計算した部分和の一部を, それを必要とするプロセッサに送信する転置処理を行う (図 4 の redist 3).

この方法の特長は, 部分和の計算処理を各プロセッサが均等に負担することにある. すなわち, ヒストグラム全体をプロセッサ台数分に分割し, 各プロセッサが担当する部分を計算し, その後, 本来その部分 and を必要とするプロセッサに計算後の部分和を戻してやることで, 各プロセッサが必要とする部分和の計算を完了する.

上述のように計算処理の並列化を行うとすると, 各プロセッサは, 計算処理を行う前後に, 大きく 2 回の通信を行うことになる. 図 4 で示したように, 各プロセッサが計算を負担するローカルヒストグラムの受信量は, たとえばヒストグラムのデータ型を 4 byte integer とすると, プロセッサ台数を P として $4 \times \text{MAXKEY} / P \times (P - 1)$ byte となり, また, 各プロセッサが計算後に

受信する部分 and は $4 \times \text{MAXKEY} \times (P - 1) / P$ byte である. つまり, 計算前後の通信の際に各プロセッサが受信するデータ量は $\text{MAXKEY}(1 - 1/P)$ に比例し, プロセッサ台数が十分大きければほぼ一定になる.

4.4 ヒストグラムの圧縮による通信時間の短縮

前節で述べたように, 各プロセッサが送受するデータ量はプロセッサ台数を増してもほとんど一定であるのに対し, 各プロセッサでのローカルな処理の計算時間は並列化によりプロセッサ台数にほぼ反比例して減少することから, プロセッサ台数を増したときには全体の計算時間に対してこの部分の通信時間が支配的になる. すなわち通信時間を短くすることがプロセッサ台数が大きい場合の性能向上の鍵となる. ここでは, さらに通信時間を抑えるために, ヒストグラムを圧縮して通信する手法を提案する.

具体的には次のようにして行う. ヒストグラム中の各キー値の出現回数は, 最大値としては全レコード数と同じになりうるが, そのようなものは全体からみれば例外的である. そこで, 転送の際に用いられるヒストグラムのデータ形式において, 全レコード数に相当する整数値を格納するのに必要なビット数 (たとえば 32 bit) ではなくその定数分の一 (たとえば 8 bit) を割り当てる. そして, それでは表現できなかったエンタリのみ後にまとめて送ることにする.

たとえば, キーの総数 N が 2^{32} 未満であることを想定して, ヒストグラムの各エンタリのデータ型が 4 byte (32 bit) integer 型であるとする. ここで, 連続する 4 つのヒストグラムの値を, 1 つの 4 byte integer 型データに圧縮することを考える. ヒストグラムの各値に, 1 byte (8 bit) ずつを割り当て, 連続する 4 つの値を 4 byte で表せばヒストグラムは, $1/4$ に圧縮されることになる. このとき, 1 byte で表せない値 (つまり 256 以上の値) がヒストグラム内に現れたときには, それらのデータのみを, 圧縮処理をかけずに,

```

!256以上の値を持つものを集める
cur=0
do i=0,MAXKEY/P-1
  if(keyden(i) >= 256) then
    over(cur) = keyden(i)
    place(cur) = i
    keyden(i) = 0
    cur=cur+1
  end if
end do

!圧縮処理
do i=0,MAXKEY/P-1
  compress(i)=0
  do j=0,3
    compress(i) = compress(i) + \
      ishft(keyden(i*4+j), j*8)
  end do
end do

!compress(MAXKEY/P/4)を通信
!over(cur)とplace(cur)を通信

```

図 5 圧縮処理のアルゴリズム

Fig. 5 The algorithm for compression.

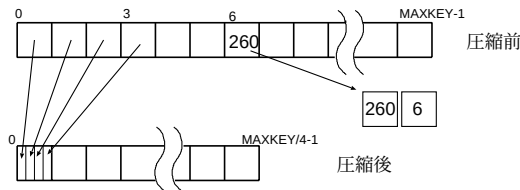


図 6 圧縮処理

Fig. 6 Compressing histogram data.

後に通信するようにする。圧縮処理のアルゴリズムを図 5 に、圧縮処理の実行の様子を図 6 に示す。

この手法を用いれば通信されるデータの量（通信量）が減少することになり、通信時間を減少させることができる。各プロセッサあたりの通信量および圧縮・伸長処理の演算量は $O(\text{MAXKEY}(1-1/P))$ である。通常の処理よりも圧縮および伸長処理の時間が余分にかかることになるが、通信処理と圧縮処理を並列に実行する、つまり、各回の通信中に次回に送信するデータの圧縮処理を行うことによって、圧縮処理時間をできるだけ隠すようにする。

4.5 トータルランクの計算

図 3, 図 4 から分かるように、各プロセッサが、ローカルヒストグラムの部分和の計算の際に計算する、 P 台目のプロセッサ用の部分和 $\text{cumulative_keyden}(k, P-1)$ は、グローバルヒストグラム $\text{global_keyden}(k)$ そのものである。つまり、各プロセッサはグローバルヒストグラムの一部をすでに計算し、保持していることになる。トータルランク total_rank すなわちグロー

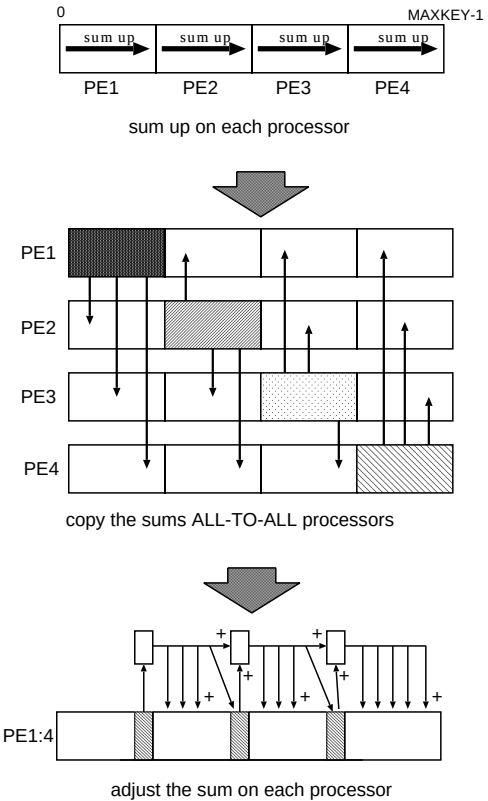


図 7 トータルランクの計算

Fig. 7 Computing total rank.

バルヒストグラム global_keyden の累計を計算する際に、このデータを用いれば、第 j 番目のプロセッサ ($j = 0, P-1$) でそれぞれ、 $j \times \text{MAXKEY}/P \leq k \leq (j+1) \times \text{MAXKEY}/P-1$ に対してトータルランクの一部である $\text{global_keyden}(j \times \text{MAXKEY}/P) + \dots + \text{global_keyden}(k)$ を計算することが可能となる。各プロセッサあたりの演算量は $O(\text{MAXKEY}/P)$ である。

その後、各プロセッサごとに計算した累計を全プロセッサ間で通信する。このときの通信量は各プロセッサあたり $O(\text{MAXKEY})$ である。

ただし、これではグローバルヒストグラム全体の連続した累計にならないので、各プロセッサでローカルに累計の補正を行い、全プロセッサがトータルランクを得るようにする。トータルランクの計算の様子を図 7 に示す。このときの演算量は $O(\text{MAXKEY}(1-1/P))$ である。

4.6 ランクの計算

単一プロセッサの実行の場合は、同じ値を持つキーに対して、ヒストグラムの累計を基準に、出現順位の情報を用いてランクの計算を行う（3 章参照）。

しかし、複数プロセッサでの実行の場合は、各プロ

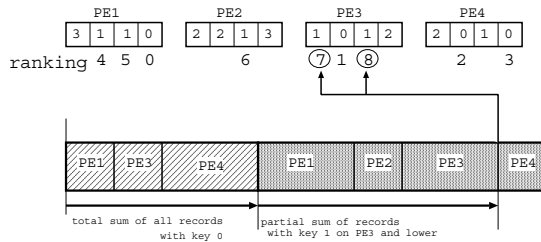


図 8 ランクの計算 (PE 数 4)

Fig. 8 Local ranking (#PE is 4).

セッサは初期状態で割り当てられた部分キー列についてヒストグラムを計算するため、割り当てられた部分キー列内での出現順位の情報しか持たない。このため、全体でのランクの計算を行うためには、トータルランクに加えて、ローカルヒストグラムの部分和の情報が必要となる。各プロセッサは、トータルランクに、ローカルヒストグラムの部分和を加えた値を基準に、出現順位の情報を用いてランクの計算を行う。

この計算は、各プロセッサで独立して行えることから、並列化による効果は、完全にプロセッサ台数に比例する。演算量は各プロセッサあたり $O(N/P)$ である。図 8 にランクの計算の様子を示す。

5. 実 装

提案した手法を、分散メモリ型ベクトル並列計算機 Fujitsu VPP800 上で実装した。実装にはデータ並列型言語 VPP Fortran^{11),12)} を用いた。

4.3 節で述べた部分和の計算における再分散の処理では、各プロセッサが、ローカルヒストグラムをプロセッサ数で分割し、巡回的に、他のプロセッサが計算を負担するローカルヒストグラム情報を通信する。図 9 に通信のアルゴリズムを、図 10 に通信の様子を示す。各プロセッサは部分和の計算前後の転置処理において、それぞれ、 $4 \times \text{MAXKEY}/P$ のデータ量の送受信を $P-1$ 回行うことになる。Fujitsu VPP のようなプロセッサがクロスバ型ネットワークで結合された並列計算機では、通信時間もプロセッサ数によらず一定にできる。

転置処理は、HPF では redistribute 指示行により記述できるものであるが、VPP Fortran の言語仕様には再分散の機能がないため、do ループに spread move 指示行による高速転送モード¹¹⁾ の指定をすることで対処した。

4.5 節で述べた、各プロセッサがキー空間方向で分散されて保持しているトータルランクの一部を全プロ

```
do i=1, P-1
  do j=0, MAXKEY/P-1
    rcv(j, pid)=send(j, mod((pid+i), P))
    !pid is the processor id
  end do
end do
```

図 9 ローカルヒストグラムの再分散のアルゴリズム

Fig. 9 The algorithm for redistribution.

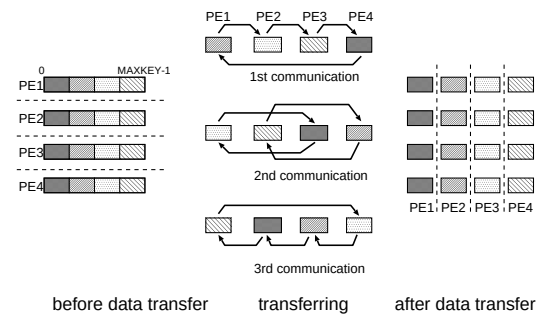


図 10 ローカルヒストグラムの再分散

Fig. 10 Redistributing local histogram.

セッサにコピーする処理^{*}は、VPP Fortran では unify 指示行による独自の転送モード¹¹⁾ で高速に行える。

6. 評 価

実装したコードの性能評価を、京都大学大型計算機センターの VPP800/63 システム上で行った。ソーティングの対象として、NAS Parallel Benchmark (NPB)⁸⁾ のベンチマークの 1 つである IS (Integer Sort) の Class C ($N = 2^{27}$, $\text{MAXKEY} = 2^{23}$) に従うデータを用いた。キー値のデータ型は、4 byte integer 型で、初期状態では分散型メモリシステムの各メモリユニットにそれぞれ N/P 個のキーを割り当てる。以上の条件で以下の 2 つの場合について、実行時間を測定した。

1. ヒストグラムを圧縮せずにそのまま転送
2. ヒストグラムを圧縮して転送

プロセッサ数 1~32 における実行時間の測定結果を表 1 に、並列化による台数効果を図 11 に示す。

非圧縮、圧縮いずれの場合もプロセッサ数が大きくなるほど、台数効果が得にくくなっていることが分かる。これは、ローカルヒストグラムの計算時間と、ランクの計算時間が完全にプロセッサ台数に比例して減少するのに対し、部分和の計算とトータルランクの計算にともない通信されるデータ量が、上述のように $1 - 1/P$ に比例するためである。つまり、プロセッサ

^{*} MPI という ALL-TO-ALL 型の通信である。

表 1 実行時間 (Class C)
Table 1 Execution time (Class C).

PE 数	圧縮なし (msec)	圧縮あり (msec)
1	1,095.0	—
2	690.8	692.5
4	401.1	378.2
8	243.1	222.1
16	182.5	144.8
32	150.3	106.6

表 2 $N = 2^{29}$, $\text{MAXKEY} = 2^{25}$ の実行時間
Table 2 Execution time ($N = 2^{29}$, $\text{MAXKEY} = 2^{25}$).

PE 数	圧縮なし (msec)	圧縮あり (msec)
1	4,230	—
2	2,681	2,646
4	1,608	1,476
8	1,079	873
16	838	566
32	692	417

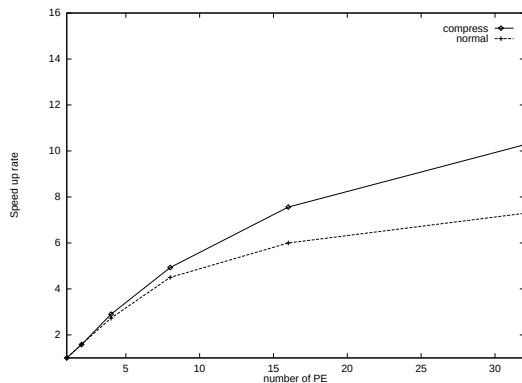


図 11 並列化による台数効果
Fig. 11 Scalability.

数の増加にともない各プロセッサあたりの計算時間が減少し、全体に占める通信処理時間の割合が高くなるために、台数効果が得にくくなっているが、圧縮の効果により 32 CPU の場合で非圧縮の場合の約 1.5 倍、1 CPU との比では 10 倍程度の性能が得られている。

さらにデータ数を増やして、クラス C の 4 倍である $N = 2^{29}$, $\text{MAXKEY} = 2^{25}$ で実行時間を計測したところ、表 2 に示すように同様に 32 CPU で約 10 倍の台数効果が得られた。4 章で示した、アルゴリズムの各部分の演算量、通信量は、いずれも N または MAXKEY の 1 次式であるため、 N と MAXKEY を同時に 4 倍に増した場合の台数効果に変化しないのは、妥当な結果といえる。

ローカルヒストグラムの部分和の計算前後の再分散処理の実行時間の測定結果を表 3 の左列 (NPB 1 のデータ) に示す。非圧縮の場合の再分散処理の時間がプロセッサ数 P に対しておおよそ $1 - 1/P$ に比例して増加していることが分かる。一方、圧縮を行った際の再分散処理の時間は、プロセッサ数によらずほぼ一定で、非圧縮の場合の半分強にまで減っている。圧縮により通信量は $1/4$ になる一方、圧縮・伸長処理にともなう演算が必要となる。プロセッサあたりの通信量、演算量はいずれも $O(\text{MAXKEY}(1 - 1/P))$ であるが、圧縮・伸長処理と通信の重ね合わせの効果により、このような結果になったと考えられる。

NPB IS で提供されている入力データは、ガウス分布に基づくランダムな整数値であり、同じキー値を持つキーの数が 256 個以上あるようなキー値は存在しない。すなわちこれは本アルゴリズムにとって最も都合のよいキー列である。そこで、参考データとして、NPB IS クラス C のキー数 $N = 2^{27}$, キー値の範囲 $\text{MAXKEY} = 2^{23}$ においてヒストグラムの圧縮処理時間が最悪となるような入力データについても実行時間を計測した。

キー数 $N = 2^{27}$ とすると、ヒストグラム内に存在する圧縮処理を適用できないデータ数はたかだか $2^{19} (= 2^{27}/256)$ 個までであり、 2^{19} 種類のキー値に対して、各キー値を持つキーが 256 個ずつ存在するのだが、圧縮処理が適用できないキーが最も多い場合である。このようなキー列を用いて、ヒストグラムに圧縮をかけて通信を行った場合と、圧縮をかけない場合の再分散処理の時間の計測結果を表 3 の右列 (参考入力データ) に示す。

参考入力データの場合についても、圧縮をかけた場合と、通常の通信を行った場合の再分散処理の差はそれほど大きくない。つまり、圧縮処理を行うアルゴリズムを用いたプログラムで、圧縮処理が適応されないデータ数が最大の場合でも、圧縮処理を行わないプログラムと実行時間がほぼ変わらない。これにより、キー数とキー値の範囲の関係が NPB IS で用いられているような典型的なものでは、圧縮処理を行うアルゴリズムを用いたプログラムが、最悪の場合でも圧縮を行わないプログラムの性能を大幅に下回ることはないと期待できる。

他の実装との比較として、本論文の実装の性能を、NPB 2 IS (C 言語で記述) の逐次版 (NPB 2.3-serial) および並列版 (NPB 2.3), ならびに文献 4) の結果と比較したものを表 4 に示す。

まず逐次版における比較では、本論文の実装では、Class C のデータに対し VPP800 1 CPU で 1.095 sec である。本論文がもとにした文献 4) の実装での実行時間が、 $1/4$ の問題サイズである Class B ($N = 2^{25}$, $\text{MAXKEY} = 2^{21}$) において NEC SX-4 1 CPU で

表 3 ローカルヒストグラムの通信処理時間
Table 3 Time for transferring local histogram.

PE 数	(NPB 1 のデータ)		(参考入力データ)	
	圧縮なし (msec)	圧縮あり (msec)	圧縮なし (msec)	圧縮あり (msec)
2	34.1	34.4	34.1	39.3
4	52.2	34.4	52.2	43.7
8	56.0	34.5	56.0	55.0
16	68.1	34.9	68.1	63.3
32	69.3	37.0	69.3	73.8

表 4 実装ごとの逐次および並列性能の比較
Table 4 Comparison of serial and parallel performance among implementations.

クラス	逐次(sec)	並列(sec)	CPU 数	計算機
本論文 (Class C)	1.095	0.1066	32	Fujitsu VPP800
NPB 2.3 (Class C)	119.2	6.33	32	Fujitsu VPP800
文献 4) (Class B)	6.77	1.05	8	NEC SX-4

6.77 sec であることと比較して, VPP800 1 CPU の浮動小数点演算性能が SX-4 の約 4 倍*であることを考慮しても, 単一 CPU によるベクトルアルゴリズムとして十分高速に実装できていることが分かる.

そのうえで本論文のアルゴリズムは, 圧縮を用いた場合, 8 CPU で 222.1 msec (加速率約 5 倍), 32 CPU で 106.6 msec (加速率約 10 倍) という実行時間であった. 共有メモリ型ベクトル並列計算機である SX-4 の場合の 8 CPU での実行時間が 1.05 sec (加速率約 6 倍) であることと比べ, 同一 CPU 数での加速率では及ばないものの, 絶対性能としても分散メモリ型ベクトル並列計算機上での実装としても本アルゴリズムは高速であるといえる**.

ちなみに NPB 2.3 のコードは逐次版, 並列版いずれもほとんどベクトル化されず***, 並列化した場合の加速率は約 18.8 倍と大きいものの, 32 CPU での性能においても我々の実装の 1 CPU の性能に及ばなかった.

7. ま と め

本論文では, バケツソートにおいて, ヒストグラムの転送に圧縮した形式を用いることで, 通信時間を減少させる手法を提案し, それに基づく分散メモリ型ベクトル並列計算機上での整数ソーティングの高速な実装について述べた. Fujitsu VPP800 上で実装し, NPB 中の IS ベンチマークを用いて, 実行時間を計測し, 評価を行った. その結果, 特にプロセッサ数が

小さい場合に並列化による高い効果を示した. また, 特にプロセッサ数が大きい場合に通信データの圧縮による効果が高いことを示した. さらに実行時間についても, 従来のものに比べ, 非常に高速であることが確認された.

今回の実装には Fujitsu 独自のデータ並列言語である VPP Fortran を用いたが, 設計段階ではより汎用性の高い HPF を用いて記述している. VPP800 では HPF 2.0 の処理系も提供されていることから, HPF への移植も検討中である.

謝辞 ベンチマークプログラムの実行にご協力いただいた京都大学大型計算機センター平野彰雄技官, 浅岡香枝技官に感謝します. 多数の有益なコメントをいただいた査読者の方々に深謝します. なお, 本研究は一部京都大学大型計算機センター開発計画による.

参 考 文 献

- 1) Belettoch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G. and Smith, S.J.: An experimental analysis of parallel sorting algorithms, *Theory of Computing System*, Vol.31, No.2, pp.135-167 (1998).
- 2) Grün, T. and Hillebrand, M.A.: NAS integer sort on multi-threaded shared memory machines, *Proc. 4th International Euro-par Conference (Euro-Par'98)*, pp.999-1009 (1998).
- 3) 石浦菜岐佐, 高木直史, 矢島脩三: ベクトル計算機上でのソーティング, 情報処理学会論文誌, Vol.29, No.4, pp.378-385 (1988).
- 4) 村井 均, 末広謙二, 妹尾義樹: 共有メモリ型ベクトル並列計算機上の高速ソーティングアルゴリズム, 情報処理学会論文誌, Vol.39, No.6, pp.1595-1602 (1998).
- 5) High Performance Fortran Forum: *High Per-*

* VPP800 は 8GFlops, SX-4 は 2GFlops.

** 著者らの知る限り, クラス C ベンチマークの性能値として最高速のものである.

*** このコードについては, NPB2 の規約に従い, ベクトル化指示の挿入などソースコードの変更は行っていない.

formance Fortran Language Specification Version 2.0 (1996).

- 6) 岡田 信, 坂本喜則, 浅井 登: VPP500 システムのソフトウェア, 電子情報通信学会論文誌, Vol.J78-D-I, No.2, pp.149-161 (1995).
- 7) 岩下英俊: HPF からみた VPP Fortran, 情報処理, Vol.38, No.2, pp.114-120 (1997).
- 8) Bailey, D., Barszcz, E., Barton, J., Brown-ing, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V. and Weeratunga, S.: The NAS parallel benchmarks, RNR Technical Report, RNR-94-007 (1994).
- 9) Elton, B.H. and Miura, K.: A Vector-Parallel Implementation and Statistical Analysis of the Bucket Sort on a Vector-Parallel Distributed Memory System: Lessons Learned in the Integer Sort NAS Parallel Benchmark, *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing*, pp.782-783 (1995).
- 10) 太田 寛, 西谷康仁, 小林 篤, 布広永示: HPF 処理系 Parallel FORTRAN による NAS Parallel ベンチマークの並列化, 情報処理学会論文誌, Vol.38, No.9, pp.1830-1839 (1997).
- 11) (株)富士通: UXP/V Fortran90/VPP 使用手引書 V10 用, J2U5-0080-03 (1997).
- 12) (株)富士通: UXP/V VPP プログラミングハンドブック V10 用, J2U5-0070-01 (1997).

(平成 13 年 2 月 5 日受付)

(平成 13 年 5 月 24 日採録)



横山 栄二

1998 年京都大学工学部情報工学科卒業。2000 年同大学大学院情報学研究科システム科学専攻修士課程修了。現在松下電器産業株式会社勤務。在学中、並列計算機用ソフトウェアの

開発に従事。



安岡 孝一

1965 年生。1990 年京都大学大学院工学研究科情報工学専攻修士課程修了。同年京都大学大型計算機センター助手, 1997 年同助教授。スーパーコンピュータにおける高速アル

ゴリズムの研究に従事。2000 年京都大学人文科学研究科所属漢字情報研究センター助教授。文字コードおよび漢字情報論の研究に従事。京都大学博士(工学)。電気情報通信学会会員。



岡部 寿男(正会員)

1964 年生。1988 年京都大学大学院工学研究科情報工学専攻修士課程修了。同年同大学工学部助手。同大学大型計算機センター助教授を経て、現在同大学大学院情報学研究科助教

授。博士(工学)。並列・分散アルゴリズム, インターネットワーキング等の研究に従事。電子情報通信学会, システム制御情報学会, 日本ソフトウェア科学会, IEEE, ACM, EATCS 各会員。



金澤 正憲(正会員)

1946 年生。1971 年京都大学大学院工学研究科数理工学専攻修士課程修了。1972 年同大学大型計算機センター助手。同助教授を経て、1995 年同教授, 現在に至る。工学博士。計

算機システム・オペレーティングシステムの方式と性能評価, およびコンピュータネットワーク・分散システムに興味を持っている。電子情報通信学会, ACM, 日本応用数理学会各会員。