

分散メモリ型ベクトル並列計算機上での 高速ソーティングアルゴリズム

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

本稿では、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 Retry アルゴリズムを用いて行う。並列化の特長は、プロセッサ間でのデータの通信処理における次の2点である。1. プロセッサ間での通信時間がプロセッサ数に依らずほぼ一定である、2. 通信データを圧縮することにより、通信時間の短縮を図る。これらの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、NPB(NAS Parallel Benchmarks) の中の IS(Integer Sort) ベンチマークを用いて評価を行ったところ、Class C のデータに対して、32CPU で 106.6msec の実行時間となり、非常に高速であることが確認された。

A Fast Integer Sorting Algorithm for Distributed-Memory Parallel Vector Supercomputers

Eiji Yokoyama[†] Koichi Yasuoka^{††} Yasuo Okabe[†] Masanori Kanazawa^{††}

In this paper, we propose a fast sorting algorithm for distributed-memory parallel vector supercomputers. Our algorithm is based on the bucket sort, and uses "retry algorithm" which is effective for a vector processor for calculating a histogram of a sequence of keys. Our parallelization technique depends on two key ideas as follows: (1) The processor-to-processor communication time is almost independent of the number of processors, (2) The processor-to-processor communication time is reduced by data compression. Using the algorithm, we have implemented a parallel program on Fujitsu VPP 800. We have evaluated its performance on Integer Sort benchmark of NAS Parallel Benchmarks, and obtained 106.6 msec execution time with 32 processors.

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ひとつであり、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、並列化の分野においても、高速なソーティングアルゴリズムについて多くの研究が行われてきている [1, 2]。

本稿では、分散メモリ型並列計算機上でソーティングを高速に行う手法について述べる。本アルゴリズムは、バケツソートを基本とする。バケツソートはベクトル実行においては、最も高速なアルゴリズムのひとつとして知られている [3, 4]。本稿で提案する並列バケツソートでは、まず各プロセッサが、ローカルヒストグラムを計算し、その後、ローカルヒストグラムをプロセッサ間で通信し、グローバルヒストグラムを得て、ジェネラルランクを計算し、各キーに順位をつけることになる。このアルゴリズム

の詳細は4章で述べる。

本稿で対象とするような、分散メモリ型の並列計算機上でのアルゴリズムの並列化において、プロセッサ間の通信が最も重大なボトルネックになることが言われている [6, 7]。つまり、通信時間を抑えることが、並列アルゴリズムの高速化の大きな要因であると言える。そこで、本稿では、バケツソートの並列化におけるプロセッサ間の通信時間を抑え、並列化による効果をより高めるための二つの手法を提案する。一つは、計算処理の負担を各プロセッサで均等にし、プロセッサ数の増大に伴う通信処理時間の増大を抑える手法であり、もう一つは、通信データを圧縮することにより、通信時間を短縮する手法である。

実際に、本稿で提案する手法を Fujitsu VPP800 上で実装し、並列計算機の性能評価の指標として広く用いられている NAS Parallel Benchmarks(NPB) [8] の中のベンチマーク IS(Integer Sort) を用いて、評価を行った。

以下、2章で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た後、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を抑える手法について述べる。さらに5章でそ

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

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

これらの手法を実装し、実行時間の評価、考察を行い、6章でまとめを行う。

2 整数ソーティングの問題の定義

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

ソーティング N 個のキー: $\{K_i \mid i = 0, 1, \dots, N-1\}$ をソートするとは、 $K_i \leq K_{i+1} \leq K_{i+2} \dots$ を満たすように並べ替える処理である。ここで、 K_i は、 $0 \leq K_i \leq \text{MAXKEY} - 1$ を満たす整数、 MAXKEY はキーの最大値である。

ランキング ランクとはキー列がソートされたときの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、このキー列はソートされたことになる。

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。本稿で扱うソーティングでは、同じ値を持つキー間での付け方は任意とする。

本稿では、主にランキングの並列化について議論する。

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, 4] 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。この時、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 (図1参照)。
- 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 ($\text{keyden}(0) + \text{keyden}(1) + \dots + \text{keyden}(k)$) 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。

!ヒストグラムの生成

```
do i=0, N-1
  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=0, N-1
  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

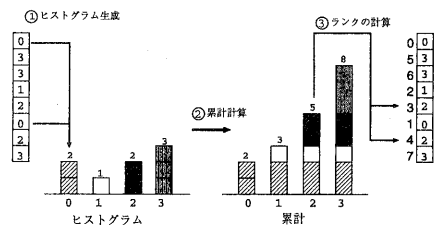


図2: バケツソートの実行の様子

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。始めに、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: 各プロセッサ毎に、初期状態で割り当てられたキー列: $\{\text{key}(i) \mid i = 0, 1, \dots, N/P-1\}$ のヒストグラム (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章参照)。
- 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。この時同時に、全体のキー列 $\{K_i \mid i = 0, 1, \dots, N-1\}$ のヒストグラム (以下、グローバルヒストグラムと呼ぶ) も求める (4.3章参照)。
- ③ ジェネラルランクの計算: ②で求めたグローバルヒストグラムから、ジェネラルランクを計算する。ここで、ジェネラルランクとは、グロ

ーバルヒストグラム (*global_keyden*) の累計、つまり値 k に対する累計 ($global_keyden(0) + global_keyden(1) + \dots + global_keyden(k)$) を $k = 0, 1, \dots, MAXKEY - 1$ の各値について計算したものである (4.4 章参照)。

- ④ ランクの計算: ②、③で生成されたジェネラルランクおよび、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 (4.5 章参照)。

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 [5] を用いて、各プロセッサ毎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、バケット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るため、並列化による効果は完全にプロセッサ台数に比例する。

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

ローカルヒストグラムの部分和は、後述のランクの計算 (4.5 章参照) 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。

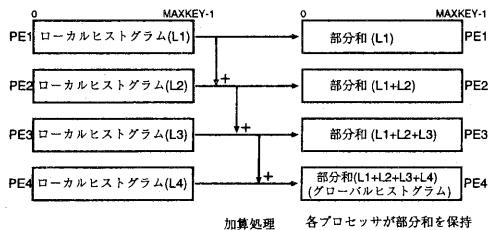


図 3: ローカルヒストグラムの部分和 (PE 数 4)

部分和の計算の並列化とそれに伴う通信処理において、並列化の効果を得るためには以下の二点が重要となる。

1. 各プロセッサが均等に計算を負担する。
2. プロセッサ数に依らず、通信時間がほぼ一定である。

以下では実際に分散メモリ型並列計算機上で、部分和の計算を行うアルゴリズムについて述べる。

部分和の計算の概要

本アルゴリズムで部分和を求めるために用いる手法を図 4 に示し、概要を以下に述べる。

- ① ローカルヒストグラムの通信: 各プロセッサは、ローカルヒストグラムをプロセッサ数で分割し、他のプロセッサと巡回的に、自分が計算を担当

する部分のローカルヒストグラムを通信する。(具体的な通信方法は次節参照。)

- 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 計算後の部分和の通信: 各プロセッサは、①と同様に他のプロセッサと巡回的に通信を行い、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。

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 and を戻してやることで、各プロセッサが必要とする部分 and の計算を完了する。この結果、通信は部分 and の計算の前後に一度ずつ、二回発生し、その通信時間はプロセッサ台数を P とすると、 $(P-1)/P$ に比例する。つまり、通信時間は P に関わらずほぼ一定になるといえる。次節では、このような通信を実現するための具体的な方法について述べる。

ローカルヒストグラムの通信

上記の様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、大きく二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、ヒストグラムのデータ型を 4Byte integer とすると、 $4 \times MAXKEY/P \times (P-1)$ Byte となり、プロセッサ台数に依らずほぼ一定であることがわかる。また、各プロセッサが計算後に受信する部分 and は $4 \times MAXKEY \times (P-1)/P$ Byte で、やはりプロセッサ台数に依らずほぼ一定である。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、共に $4 \times MAXKEY \times (P-1)/P$ Byte ということになる、プロセッサ台数に依らずほぼ一定であると言える。

この通信を効率良く行うために、各プロセッサは、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 5 参照)。つまり、各プロセッサは部分 and の計算前後の通信において、それぞれ、 $4 \times MAXKEY/P$ のデータ量の送受信を $P-1$ 回行うことになる。

この手法を用いれば、通信時間はプロセッサ数に依らず一定で、かつ通信データ量を必要最小限に抑えることができる。通信時間は、通信データの量に大きく依存するため、この手法を用いれば、通信時間を抑える効果が期待できる。

図 6 に通信のアルゴリズムを、図 5 に通信の様子を示す。

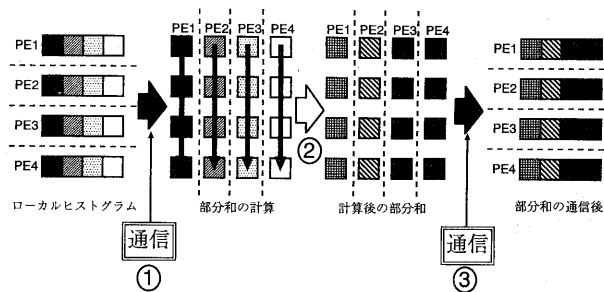


図 4: 部分和の計算の流れ (PE 数 4)

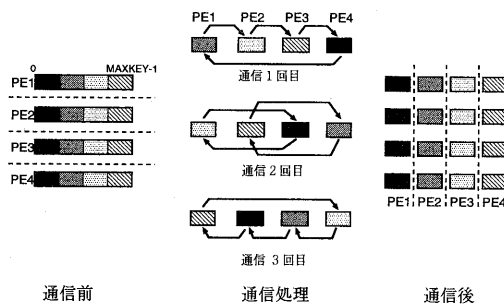


図 5: ローカルヒストグラムの通信

```

do i=1, P-1
  do j=0, MAXKEY/P-1
    rcv(j, pid)=send(j, mod((pid+i), P))
    !pidはプロセッサ番号
  end do
end do

```

図 6: ローカルヒストグラムの通信のアルゴリズム

ヒストグラムの圧縮による通信時間の短縮

すでに述べたように、通信時間は通信データの量に大きく依存する。ここでは、さらに通信時間を抑えるために、通信データを圧縮する手法を提案する。本プログラムで用いているヒストグラムのデータ型は 4Byte integer 型である。ここで、連続する 4 つのヒストグラムの値を、1 つの 4Byte 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、1Byte ずつを割り当て、連続する 4 つの値を 4Byte で表せばヒストグラムは、1/4 に圧縮されることになる。この時、1Byte 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時には、それらのデータのみを、圧縮処理をかけずに、後に通信するようにする。圧縮処理のアルゴリズムを図 8 に、圧縮処理の実行の様子を図 7 に示す。

この手法を用いれば通信データの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。また、通常の処理よりも圧縮処理の時間が余分にかかる

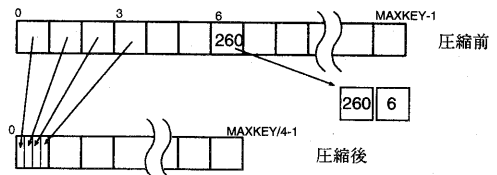


図 7: 圧縮処理

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。

4.4 ジェネラルランクの計算

図 3、図 4 からわかるように、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は、グローバルヒストグラムである。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。ジェネラルランクすなわちグローバルヒストグラムの累計を計算する際に、このデータを用いれば、各プロセッサでそれぞれ、ジェネラルランクの一部を計算することが可能となる。その後、各プロセッサ毎に計算した累計を全プロセッサ間で通信する。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、全プロセッサがジェネラルランクを得るようにする。ジェネラルランクの計算の様子を図 9 に示す。

4.5 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、

```

!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/4-1
  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 8: 圧縮処理のアルゴリズム

ジェネラルランクに加えて、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、ジェネラルランクに、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。図 10にランクの計算の様子を示す。

5 評価

本稿で提案した手法を、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 [9, 10] を用いた。

ソーティングの対象として、NAS Parallel Benchmark(NPB) [8] のベンチマークの一つである IS (Integer Sort) の Class C ($N = 2^{27}$, $MAXKEY = 2^{23}$) に従うデータを用いた。キー値のデータ型は、4Byte 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. 通常ヒストグラムの通信
2. ヒストグラムに圧縮をかけた通信

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、ローカルヒストグラムの通信処理時間のみを測定した結果を表 2 に示す。

表 1 に示したように、本アルゴリズムでは、Class C のデータに対し 32CPU で 106.6msec という実行時間であった。文献 [5] での実行時間が、IS の Class B ($N = 2^{25}$, $MAXKEY = 2^{21}$) のデータに対して NEC SX-4 で 8CPU を用いて 1.05sec であったという結果を考えると、本アルゴリズムは非常に高速であるといえる。

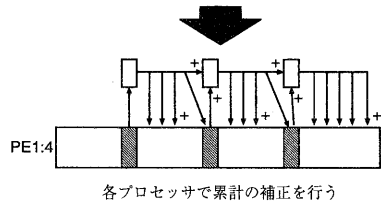
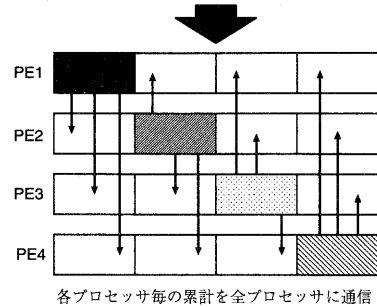
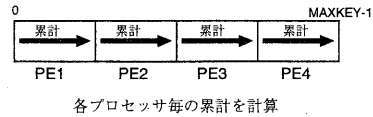


図 9: ジェネラルランクの計算

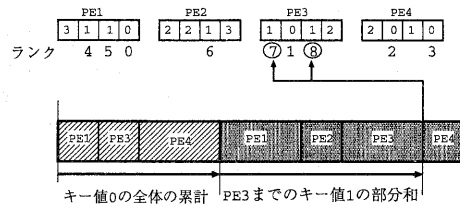


図 10: ランクの計算 (PE 数 4)

表 2 において、通常通信の時間が増加していることがわかる。これは、4.3 章で示したように、通信時間がプロセッサ数 P に対して、 $(P-1)/P$ に比例することに起因する。

また、図 11 において、通常通信を行った場合の処理時間を見ると、プロセッサ数が小さい時は、並列化による十分な台数効果が得られているが、プロセッサ数が大きくなるほど、台数効果が得にくくなることがわかる。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、部分和の計算とジェネラルランクの計算に伴う通信時間が、上述のように $(P-1)/P$ に比例するためである。つまり、プロセッサ数の増加に伴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。

一方、圧縮を用いた場合の処理時間を見ると、前述のように、プロセッサ数の増加に伴い総処理時間

表 1: 実行時間

PE 数	通常 (msec)	圧縮 (msec)
1	1095.0	-
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

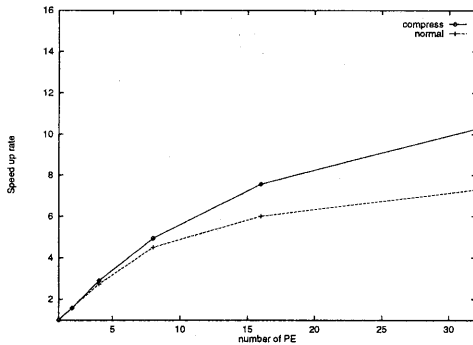


図 11: 並列化による台数効果

に占める通信処理時間の割合が高くなっているため、プロセッサ数が大きくなるほど、圧縮の効果が出ていることがわかる。すなわち、表 2 に示すように、圧縮をおこなった際の通信時間は、プロセッサ数に依らずほぼ一定となるため、通信処理時間による実行時間のロスが、圧縮をかけない場合に比べて減じているのである。また、プロセッサ数が小さい場合に圧縮による効果が出ていないのは、圧縮による通信時間の減少と、圧縮処理にかかる時間が相殺したためだと考えられる。

さらにデータ数を増やして、 $N = 2^{29}$, $MAXKEY = 2^{25}$ で実行時間を計測したところ、表 3 に示すような結果が得られた。この結果、本手法ではデータの大きさに関わらず、32CPU で約 10 倍の台数効果が得られることが示された。

6 まとめ

本稿では、分散メモリ型並列計算機上で整数ソーティングを高速に実行する手法について述べた。

分散メモリ型の並列計算機でバケツソートを行う際に、プロセッサ間の通信処理をほぼ一定に抑える手法を提案した。また、通信データを圧縮することによって、通信時間を減少させる手法を用いて、さらなる高速化を図った。

表 2: ローカルヒストグラムの通信処理時間

PE 数	通常 (msec)	圧縮 (msec)
2	34.1	34.4
4	52.2	34.4
8	56.0	34.5
16	68.1	34.9
32	69.3	37.0

表 3: $N = 2^{29}$, $MAXKEY = 2^{25}$ での実行時間

PE 数	通常 (msec)	圧縮 (msec)
1	4230	-
2	2681	2646
4	1608	1476
8	1079	873
16	838	566
32	692	417

上記の手法を分散メモリ型並列計算機 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。

参考文献

- [1] G.E.Bellicchio, C.E.Leiserson, B.M.Maggs, C.G.Plaxton, S.J.Smith : An experimental analysis of parallel sorting algorithms, Theory of computing system, vol.31,no.2, pp.135-167 (1998).
- [2] T.Grun, M.A.Hillebrand : NAS integer sort on multi-threaded shared memory machines, Proceedings of 4th International Euro-par Conference, pp.999-1009 (1998).
- [3] 津田 孝夫 : 数値処理プログラミング, 岩波書店 (1988).
- [4] 石浦 業岐佐, 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).
- [5] 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).
- [6] C.Kruskal, L.Rudolph, M.Smir : A complexity theory of efficient parallel algorithms, Theoretical Computer Science, vol.71, no.1, pp.95-132 (1990).
- [7] D.E.Culler, R.M.Karp, D.A.Patterson, A.Shahay, K.E.Schauser, E.Santos, R.Subramonian, T.von Eicken : LogP: towards a realistic model of parallel computation, Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp.1-12 (1993).
- [8] D.Bailey, E.Barszcz, J.Barton, D.Browning, R.Carter, L.Dagum, R.Fatoohi, S.Fineberg, P.Frederickson, T.Lasinski, R.Schreiber, H.Simon, V.Venkatakrishnan and S.Weeratunga : The NAS parallel benchmarks, RNR Technical Report, RNR-94-007 (1994).
- [9] (株) 富士通 : UXP/V Fortran90/VPP 使用手引書 V10 用, J2U5-0080-03(1997).
- [10] (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).