共有メモリ型ベクトル並列計算機上の高速整数ソーティングアルゴリズム

村井 均[†] 末広謙二[†] 妹尾義樹[†]

本稿では，共有メモリ型ベクトル並列計算機をターゲットとした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するもので，次の2点を特長とする．1．我々が開発したRetryアルゴリズムによりヒストグラムの生成を高速にベクトル実行する．2．累計の計算において，配列データに対する並列処理の方向を変更することによって，並列実行の効率を高める．NEC SX-4上での評価の結果，従来の作業ベクトルを用いて依存関係を回避する手法に比べて，本手法は4倍近い高速化が達成できた．また，並列実行においては8CPUで約6倍の台数効果が得られた．

Fast Integer Sorting Algorithm for Shared-memory Vector Multiprocessors

HITOSHI MURAI,[†] KENJI SUEHIRO[†] and YOSHIKI SEO[†]

This paper describes a fast sorting algorithm for shared-memory vector multiprocessors. The algorithm, based on the bucket sort, has two major advantages: 1) it exploits the vector facilities by making histograms with a new "vector retry" algorithm, and 2) it enhances parallelism by alternating directions for calculating running sums. An implementation of the algorithm on NEC SX-4 is nearly four times faster in single execution than that of a conventional vector algorithm. It also achieves about six times as high performance in 8-CPU parallel execution as that in single execution.

1. はじめに

ソーティングは，計算機の行う最も基本的なデータ処理の1つであり，高速なソーティングアルゴリズムの開発のためにこれまで多くの研究が行われてきた[1]．さらに，近年では，スーパコンピュータの非数値応用を考えていくうえでも重要視されている[2]．このような理由から，ソーティングを高速に実行できるアルゴリズムの重要性は非常に高くなっており，スーパコンピュータの性能評価の指標として広く用いられているNAS Parallel Benchmark（NPB）[3]にもソーティングを対象としたベンチマークISが含まれている．

本稿では，共有メモリ型ベクトル並列計算機上で，整数ソーティングを高速に実行する手法について述べる．本手法は，基本的なアルゴリズムとしてバケツソート[1]を用いる．バケツソートは，ヒストグラムの生成，累計の計算，順位の計算の3つのステップから成る（3.1節を参照）．このうち，ヒストグラムの生成は一般にはベクトル化できないが，累計の計算と順位の計算は効率良くベクトル実行することができるために，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1つとして知られている[4),5]．本稿で述べる手法は，特にヒストグラムの生成のベクトル化と累計の計算の並列化に新しい方法を用いることによって，バケツソートをさらに高速に実行するものである．

ヒストグラムの生成は，粒子シミュレーションの分野において頻繁に現れるParticle Pusher[6]と呼ばれる処理の一種である．Particle Pusherは，Work Vectorアルゴリズム[7]によってベクトル化が可能ではあるが，十分な性能を得るには多くの記憶領域が必要になるという問題があるうえ，アルゴリズム適用にともない挿入される処理のオーバヘッドが大きいため，ベクトル化の効果が小さい場合が多い．これに対し，我々の開発したRetryアルゴリズムは，ベクトル計算機の特性を利用して同一アドレスへの定義を検出し，この定義に起因する結果不正を再計算によって補正するという方法により，Particle Pusherを少ない記憶領域で非常に高速にベクトル実行することが可能である．

† NEC C&C メディア研究所
NEC C&C Media Research Laoratories

一方，累計の計算は，本質的に逐次的な部分を含んでおり完全に並列化することは不可能であるが，共有メモリの特性を生かして，配列データ（ヒストグラム）に対する並列処理の方向を変更することにより，高速化を達成した．

本手法をNPB ISに適用し，その実行時間をNEC SX-4上で測定した．その結果，本手法は，従来のWork Vectorアルゴリズムによる手法に比べて，実行時間・メモリサイズの点で優れていることが確認できた．また，並列実行においても良好な結果が得られた．

以下，2章で問題の定義を行った後，3章ではバケツソートとWork Vectorアルゴリズム，4章では我々の開発したRetryアルゴリズム，5章ではバケツソートの並列化について述べる．続いて，6章ではそれらの手法の評価を行い，最後に7章でまとめを行う．

2. 整数ソーティング問題の定義

本稿で議論の対象とする整数ソーティングを，NPB ISに従って次のように定義する．

整数ソーティング

ソーティングとは，与えられた N 個のキー：

$\{0 \le key(i) \le MAXKEY-1 \mid i=0,1,\cdots,N-1\}$

を，$key(i) \le key(i+1) \le key(i+2)\cdots$ を満たすように並べ替える処理である．ここで，$key(i)$ は，

$0 \le key(i) \le MAXKEY-1$ $(i=0,1,\cdots,N-1)$

を満たす整数値をとるものとする．

ソーティングは次の2つの処理により実行される．

(1) 1対1のマッピングである順位：

$$rank : \{0,1,\cdots,N-1\} \to \{0,1,\cdots,N-1\}$$

を，

$$rank(i) < rank(j) \Rightarrow key(i) \le key(j)$$

を満たすように求める（すなわち，$rank(i)$ は，N 個のキーを昇順に並べ替えたときの $key(i)$ の先頭からの位置である）．この処理をランキングと呼ぶ．

(2) 得られた順位に従って，キーを昇順に並べ替える．すなわち，

$$key(rank(i)) := key(i)$$
$$\text{forall } i = 0,1,\cdots,N-1$$

を実行する．

ここで，同じ値を持つキーの間にも順位をつける（すなわち，$i \neq j$ ならば，$rank(i) \neq rank(j)$ である．同じ値を持つキーの間での順位の付け方は任意とする）ことに注意されたい．

(2)のキーの並べ替え処理は，共有メモリ型ベクトル並列計算機では，そのままベクトル化/並列化することにより高速に実行できる．したがって，以下では(1)のランキングのベクトル化/並列化について議論する．

3. 従来の手法

3.1 バケツソート

本稿では，次の3つのステップから成るソーティング手法をバケツソートと呼ぶ．

バケツソート

(1) ヒストグラムの生成：
キー列を走査して，各値を持つキーの個数（発生頻度）をそれぞれカウントし，キー値のヒストグラム $keyden$ を生成する．

(2) 累計の計算：
ステップ(1)で生成されたヒストグラムの累計（running sum）を計算する．すなわち，値 k に対する $keyden$ の prefix-sum $p(k) = keyden(0)+keyden(1)+\cdots+keyden(k)$ を，$keyden$ の累計を求めることによって計算する．

(3) 順位の計算：
ステップ(2)で生成された累計から各キーの順位を計算する．すなわち，値 k を持つ $keyden(k)$ 個のキーに対して，順に $p(k)-1, p(k)-2,\cdots$ を順位として与える．

バケツソートの概念図を**図1**に示す．

これらの3つのステップのうち，累計の計算はベクトル化できるが，ヒストグラムの生成と順位の計算はループ内に同一アドレスへの定義（出力依存）が存在する可能性があるため，そのままベクトル化すると正しい結果が得られない（**図2**）．

順位の計算に関しては，次のようにすればベクトル化することが可能である．すなわち，**図3**のプログラムのように，ステップ(1)のヒストグラム生成時に，

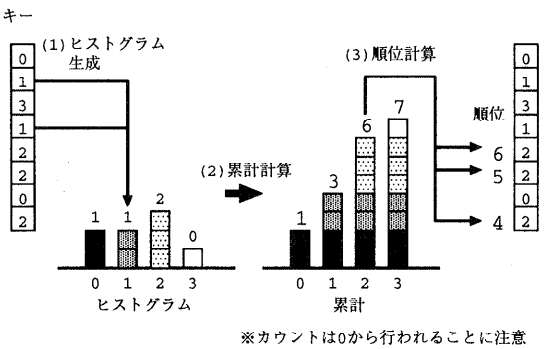


図1 バケツソートの実行の様子
Fig. 1 Execution of bucket sort.

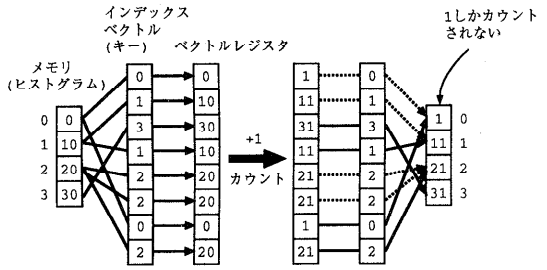


図2 出力依存による結果不正
Fig. 2 Wrong results due to output dependences.

```
{step1: ヒストグラムの生成}
for i:=0 to N-1 do begin
  k := key(i);
  keyden(k) := keyden(k) + 1;
  pos(i) := keyden(k); {出現順位の保存}
end;

{step2: 累計の計算}
{vector loop}
for k:=1 to MAXKEY-1 do begin
  keyden(k) := keyden(k) + keyden(k-1);
end;

{step3: 順位の計算}
{vector loop}
for i:=0 to N-1 do begin
  k := key(i);
  rank(i) := keyden(k) - pos(i);
end;
```

図3 バケツソート
Fig. 3 Bucket sort.

```
for i:=0 to N-1 do begin
  a(L(i)) := a(L(i)) + b(i);
end;
```

図4 Particle Pusher
Fig. 4 Particle Pusher.

同時に，同一の値を持つキーの中での各キーの出現順位を配列 *pos* に求めておき，ステップ(3)では，ステップ(2)で得られた累計からこの出現順位を引いて各キーの順位を求めるようにする[4),5)]．

しかし，ヒストグラム生成のベクトル化は何らかの方法で同一アドレスへの定義を回避しない限り不可能である．

3.2 Work Vector アルゴリズム

バケツソートのステップ(1)で行われるヒストグラムの生成は，**図4**のような総和演算の特殊な場合だと考えられる．このような処理は，Particle Pusher と呼ばれ，粒子シミュレーションの分野においてきわめて頻繁に現れるものである[6)]．

Particle Pusher をベクトル化する方法として，**Work Vector アルゴリズム**（止まり木のアルゴリズム）と呼ばれるものが従来から知られている[7)]．こ

```
for i := 0 to N-1 step K do begin
 {vector loop}
  for ii := 0 to K-1 do begin
      k := key(i+ii);
      wkden(ii, k) := wkden(ii, k) + 1;
      pos(i+ii) := wkden(ii, k);
  end;
end;
{後処理}
```

図5 Work Vector アルゴリズム
Fig. 5 Work Vector Algorithm.

のアルゴリズムは，ベクトル長 K のベクトル実行を可能とするため，総和演算の足し込み先として K 個の作業配列（Work Vector）を用意し，K 個の配列要素の足し込み先をそれぞれ異なる作業配列とすることによって同一アドレスへの定義を回避するものである．ただし，この後，K 個の作業配列を合計して本来の総和演算の結果を求める必要がある．

Work Vector アルゴリズムを用いれば，ヒストグラムの生成を次のようにベクトル化することが可能である．

Work Vector アルゴリズム

(1) ヒストグラムを保持する配列の次元を拡張した作業用ヒストグラム *wkden* を用意する（拡張した次元のサイズを K とする）．K 個ずつのキーが，それぞれ異なる作業用ヒストグラムへカウントされるように，拡張した次元方向でベクトル化を行う．このとき，同一の値を持つキーの中での出現順位を配列 *pos* に求めておく．

(2) K 個の作業用ヒストグラムを合計し，全キーに対するヒストグラムを求める．

(3) (1)で求めた *pos* の値は，各作業配列に足し込まれた N/K 個のキーの中での出現順位なので，これからキー全体での出現順位を求める．

(2)と(3)の処理は，通常のバケツソートの場合には存在しなかったオーバヘッドとなる．

図5に，Work Vector アルゴリズムによってヒストグラムを生成するプログラムを示す．また，**図6**に，Work Vector アルゴリズムによりヒストグラムが生成される様子を示す．

この手法で十分な性能を得るには，拡張する次元のサイズ K を大きくしてベクトル長を長くする必要がある．したがって，性能とメモリサイズはトレードオフの関係になる（ただし，K があまり大きすぎても，(2)，(3)の処理が大きくなり，性能低下の原因となる）．

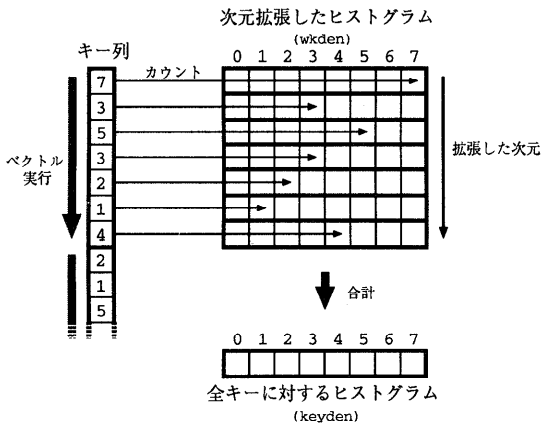


図6 Work Vectorアルゴリズムの実行の様子
Fig. 6 Execution of Work Vector Algorithm.

4. Retryアルゴリズムによるベクトル化

本章では，我々の開発したRetryアルゴリズムについて述べる．Retryアルゴリズムは，ベクトル計算機の特性を利用して同一アドレスへの定義を検出し，この定義に起因する結果不正（カウントの不足）を再計算によって補正するというものである．本アルゴリズムを用いることによって，Particle Pusherを少ない記憶領域で非常に高速にベクトル実行することが可能となる．

3.1節で述べたように，同一の値を持つキーが複数個存在する（**キー値の衝突**と呼ぶ）場合，ループ内に同一アドレスへの定義が存在するため，ヒストグラムの生成をこのままベクトル化することはできない．

そこで，何らかの手法でキー値の衝突の有無を調べる必要がある．Retryアルゴリズムでは次の処理をベクトル実行することによってキー値の衝突を検出する．

キー値の衝突の検出

(1) キー $key(i)$ の値が k であるとき，作業配列 $work(k)$ に i を書き込む．これを，すべてのキーに対して，i の昇順に適用する．値 k を持つキーが複数個存在する場合，$work(k)$ は上書きされていくことになる．

(2) ある値 k を持つキー i について，$work(k)$ と i が等しくなければ，キー値の衝突が生じている（値 k を持つキーが複数個存在する）．

この方法によれば，値 k を持つキーが複数個存在する場合，そのうち最大の番号 i を持つキー（すなわち，最も後に現れるキー）で $work(k)=i$ となり，その他では $work(k)\neq i$ となる．

キー値の衝突を検出するプログラムを**図7**に示す．

```
{vector loop}
for i := 0 to N-1 do begin
  k := key(i);
  work(k) := i;
  if (work(k) <> i) then
    { キー値の衝突あり }
  end;
end;
```

図7 キー値の衝突の検出
Fig. 7 Detection of key value conflicts.

ここで，実行時には計算機のベクトルレジスタ長VLでストリップマイニングが行われることに注意されたい．すなわち，キー値の衝突の検出は，VL個のキーを対象とした処理の繰返しとして実行される．したがって，実際に検出されるのは「VL個のキーの中で起きるキー値の衝突」であるが，以下で述べるRetryアルゴリズムにおいては，この事実はまったく支障にはならない．

このキー値の衝突の検出手法を用いて，Retryアルゴリズムは次のように記述される．

Retryアルゴリズム

(1) 通常のバケツソートのヒストグラム生成を，同一アドレスへの定義を無視して強制的にベクトル実行する．この結果，VL個のキー列の中に値 k を持つキーが1個以上存在すれば，ヒストグラム $keyden(k)$ には1が加えられる．

(2) 先に述べたキー値の衝突の検出法により，キー値の衝突の有無を調べる．その結果，VL個のキー列の中で衝突の起こらないキー（すなわち，他に同じ値を持つキーが存在しないキー）については，(1)のヒストグラムへのカウントによって，正しいヒストグラムが得られている．衝突の起こるキーについては，それらのキーのうち $work(k)=i$ となるもの（すなわち，そのようなキーの中で最後に現れたもの）がヒストグラムにカウントされたと考え，その他のキーについては配列 $fail$ にキー番号を記録しておく．

(3) 配列 $fail$ に記録された各キーを新しいキー列として，本アルゴリズムを適用する．これを，$fail$ に新たに記録されるキーがなくなるまで（すべてのキーに対してヒストグラムへのカウントが終了するまで）繰り返す．

このアルゴリズムによりヒストグラムを生成するプログラムを**図8**に，ヒストグラムが生成される様子を**図9**に示す．配列 key，$fail$ の灰色に塗られている要素が，ヒストグラムにカウントされる．それ以外の要素は $fail$ にキー番号を記録され，以下，全キーに対してカウントが終了するまで同じ操作が繰り返さ

```
ii := 0
{vector loop}
for i := 0 to N-1 do begin
  k := key(i);
  work(k) := i;
  keyden(k) := keyden(k) + 1;
  pos(i) := keyden(k);
  if (work(k) <> i) then begin
    fail(ii) = i;
    ii := ii + 1;
  end;
end;
while (ii > 0) begin
  vl := ii;
  ii := 0;
  {vector loop}
  for i := 0 to vl-1 do begin
    k := key(fail(i));
    work(k) := i;
    keyden(k) := keyden(k) + 1;
    pos(fail(i)) := keyden(k);
    if (work(k) <> i) then begin
      fail(ii) = fail(i);
      ii := ii + 1;
    end;
  end;
end;
```

図8 Retry アルゴリズム
Fig. 8 Retry Algorithm.

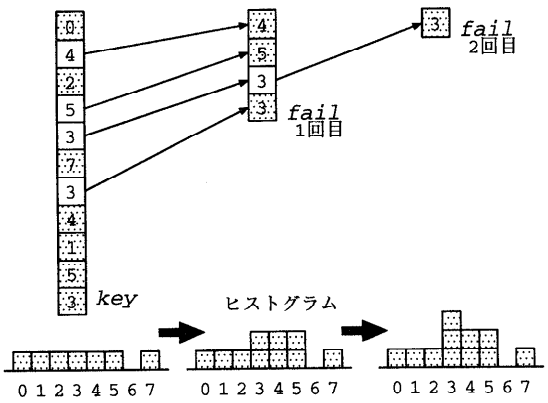


図9 Retry アルゴリズムの実行の様子
Fig. 9 Execution of Retry Algorithm.

れる.

Retry アルゴリズムは，キー値の衝突が解消されるまで繰り返される．たとえば，値 k を持つキーが VL 個のキーの中に 10 個存在すれば，繰返し回数は少なくとも 10 回となる．したがって，キー値の衝突の頻度が大きい場合には効率が低下するが，6.3 節で述べるように，このような場合は稀であると思われる．

5. バケツソートの並列化

本章では，バケツソートを並列化する方法について述べる．対象の計算機としては，共有メモリ型のベクトル並列計算機を想定する．

Retry アルゴリズムを用いたバケツソートは，次のように並列化することができる．

並列バケツソート

(1) **ローカル・ヒストグラムの生成**：
各プロセッサが，Retry アルゴリズムによって，自分の担当するキーに対するヒストグラム（**ローカル・ヒストグラム**と呼ぶ）を並列に生成する．

(2) **部分累計の計算**：

(a) **グローバル・ヒストグラムの計算**：ステップ (1) で生成されたローカル・ヒストグラムから，すべてのキーに対するヒストグラム（**グローバル・ヒストグラム**と呼ぶ）を求める．

(b) **全体の累計の計算**：グローバル・ヒストグラムから全体の累計を計算する．

(c) **部分累計の計算**：全体の累計からプロセッサごとの累計（**部分累計**と呼ぶ）を計算する．

(3) **順位の計算**：
ステップ (2) で得られた部分累計を基準として，各キーの順位を並列に計算する．

このうち，ステップ (1) のローカル・ヒストグラムの生成とステップ (3) の順位の計算は，各プロセッサが自分の担当するキーに対して処理を行うことによって，ベクトル化/並列化することができる．

以下では，ステップ (2) の部分累計の計算を並列化するために用いた手法について述べる．

5.1 (a) グローバル・ヒストグラムの計算

ステップ (2)(a) では，ローカル・ヒストグラムを合計してグローバル・ヒストグラムを計算する．

このとき，以下の 2 点が高速化のポイントとなる．

- 各プロセッサが保持しているデータ（ローカル・ヒストグラム）間で演算を行う必要があるため，各プロセッサがそれぞれ互いに異なるキー値の範囲を保持するようにローカル・ヒストグラムの分割を変更し，その合計を並列に求める．
- ステップ (2)(c) で部分累計を計算するときのために，ローカル・ヒストグラムを合計してグローバル・ヒストグラムを求める際，その途中の結果である部分和を求めておく．

図 10 にグローバル・ヒストグラムの計算の様子を示す．

5.2 (b) 全体の累計の計算

累計を求める処理は，一次回帰演算になるため，完全に並列に実行することは不可能である．そこで，まず，ステップ (2)(a) で得られたグローバル・ヒストグラムを分割し，各プロセッサでその各々の部分の累

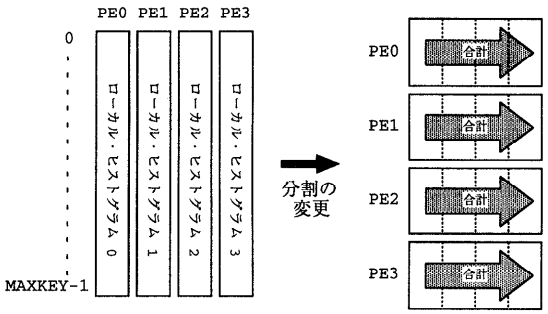


図10 グローバル・ヒストグラムの計算
Fig. 10 Calculation of global histogram.

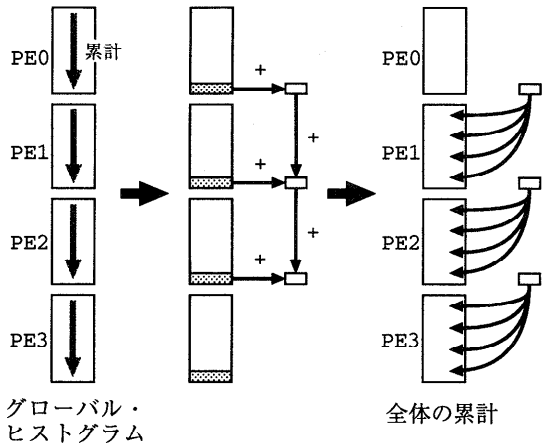


図11 全体の累計の計算
Fig. 11 Calculation of global running sum.

計を並列に求める．

元のヒストグラムで連続する2つの分割においてそれぞれ累計を求め，先行する側（より小さいキー値に対するヒストグラムの側）の最後の要素を，他方のすべての要素に加えると，この2つの分割全体の累計が得られたことになる．各分割の要素に加えるべき値をあらかじめ計算しておけば，加算処理自体は並列に行うことができる（**図11**）．

5.3 (c) 部分累計の計算

ステップ(3)において順位の計算を行う際，単一プロセッサの実行では，ヒストグラムの累計を基準として用いたが，複数プロセッサの実行では，各キーの順位を並列に計算するために，プロセッサごとに基準が必要となる．これには，累計の値を，各プロセッサごとに分割して得られる部分累計を用いる．

部分累計は，全体の累計の値より，グローバル・ヒストグラム生成時に求めておいたヒストグラムの部分和を引くことによって得られる．

部分累計を用いると**図12**のように順位を求めることができる．

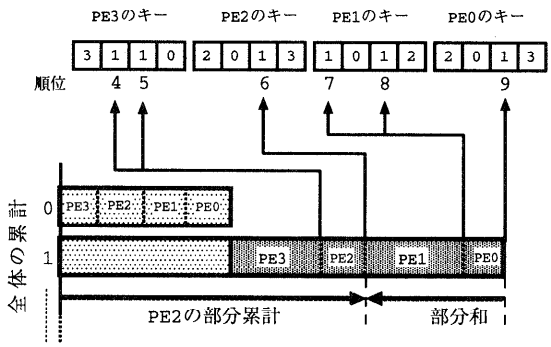


図12 部分累計
Fig. 12 Partial running sum.

6. 評　価

開発した手法の評価をNEC SX-4[8]上で行った．評価用のプログラムは，FORTRAN77/SXコンパイラによってベクトル化/並列化したものである（コンパイルオプションには，ベクトルレジスタ長256の計算機向けの最適化を行うことを指示する-pvctl vr256を指定した）．

6.1 Retryアルゴリズムの評価

Retryアルゴリズムと従来のWork Vectorアルゴリズムによる手法を実際のプログラムに適用し，1CPUでの実行時間および実行に要するメモリサイズを測定・比較した．

対象のプログラムとしては，NAS Parallel Benchmark（NPB）[3]に含まれるベンチマークISのクラスB（$N = 2^{25}$, $MAXKEY = 2^{21}$）を用いた．また，Work Vectorアルゴリズムの拡張次元のサイズとして8，16，32，64の各場合で測定した．その結果を**表1**に示す．なお，Retryアルゴリズムにおいて，Retryの回数は2回であった．

表から，Retryアルゴリズムを用いることにより，拡張次元のサイズを64としたWork Vectorアルゴリズムに比べても4倍近くの性能が得られていることが分かる．メモリサイズに関しても，実行速度の大きく劣る，拡張次元のサイズが8，16のWork Vectorアルゴリズムに比べると大きくなっているものの，拡張次元のサイズが32，64の場合に比べれば小さくなっており，実行時間とメモリサイズの両方において，Retryアルゴリズムは Work Vectorアルゴリズムに比べて優れているといえる．

6.2 並列バケツソートの評価

5章で述べた手法により並列化を行ったプログラムの実行時間を測定した．対象のプログラムとしては，NPB ISのクラスB（$N = 2^{25}$, $MAXKEY = 2^{21}$）

表1 Retry アルゴリズムの性能
Table 1 Performance of Retry Algorithm.

	実行時間 (sec)	メモリサイズ (MB)
Retry	6.77 (38.00)	540
Work Vector (8)	71.55 (3.60)	468
Work Vector (16)	44.11 (5.83)	532
Work Vector (32)	32.48 (7.92)	660
Work Vector (64)	26.20 (9.81)	916
スカラ	257.27 (1.00)	456

実行時間の欄の括弧内はスカラ実行に対する性能比を示す.

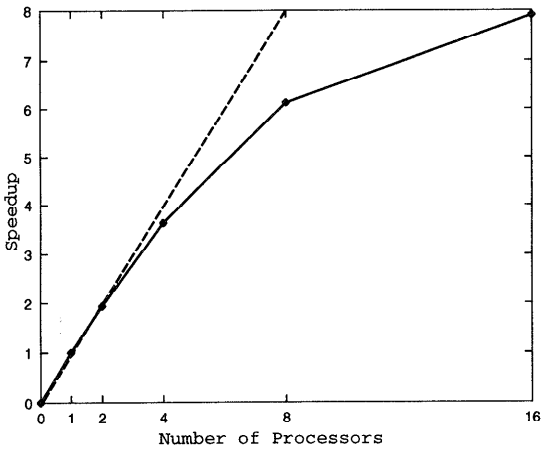


図13 並列バケツソートの性能
Fig. 13 Performance of parallel bucket sort.

を用いた．その結果を**図13**に示す．また，参考として，1～8 CPU 実行時に各ステップの実行に要する時間を**表2**に示す．なお，Retry の回数は，どの CPU 台数でも 2 回であった．

図13から，開発した手法により並列化を行うことで，1～8 CPU の実行では特に優れた性能が得られていることが分かる．並列バケツソートの3つのステップのうち，ヒストグラムの生成と順位の計算は完全に並列化できるため，その実行時間は CPU 台数とともにリニアに減少する．一方，部分累計の計算は，CPU 台数に比例する計算量の増加と並列化による速度向上が相殺した結果，CPU 台数にかかわらず実行時間がほぼ一定になっている．したがって，CPU 台数が増えたときに性能低下が見られるのは，部分累計の計算に要する時間の比率が大きくなってくることとアルゴリズム自体には起因しないその他の要因のためである．

6.3 キー値の衝突の頻度と性能の関係

Retry アルゴリズムと Work Vector アルゴリズムについて，キー値の衝突の頻度と性能の関係を調べた．

対象のプログラムとして NPB IS を用い（ただし，測定値の精度を良くするためにソーティングの回数を

表2 各部分の実行時間
Table 2 Execution time of each step.

	ヒストグラム (sec)	部分累計 (sec)	順位 (sec)
1 CPU	5.75	0.05	1.56
2 CPU	2.93	0.08	0.78
4 CPU	1.53	0.07	0.39
8 CPU	0.79	0.06	0.20

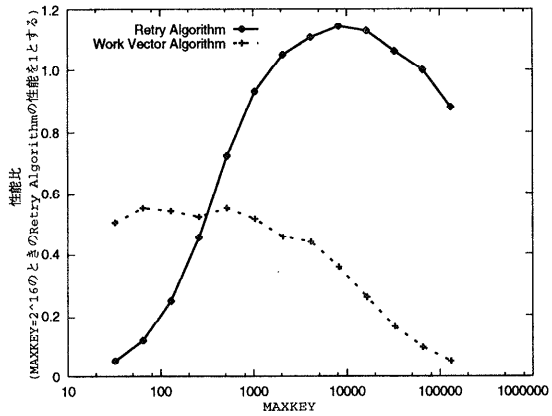


図14 キー値の衝突の頻度と性能の関係
Fig. 14 The relation between performance and frequency of key value conflicts.

100 回としている），キー数 $N = 2^{16}$ として，キー値の最大値 $MAXKEY$ を変化させたときの 1 CPU での実行時間を測定した．また，Work Vector アルゴリズムの拡張次元のサイズは 64 とした．

図14に，その結果を示す．グラフから分かるように，Retry アルゴリズムは，$MAXKEY$ がおよそ 10000 のときに最大の性能を示している．$MAXKEY$ が 10000 を上回ると，累計の計算に要する時間（$MAXKEY$ に比例する）が支配的になるため性能が低下している．$MAXKEY$ が 10000 を下回るとキー値の衝突の頻度が大きくなるため，4 章に述べたように性能が低下している．また，$MAXKEY$ の値がおよそ 300 を下回り，キー値の衝突の頻度が非常に大きい場合には，Work Vector アルゴリズムの方が優れた性能を示すが，その他の場合には Retry アルゴリズムの性能は Work Vector アルゴリズムを大きく上回っている．

7. ま と め

本稿では，共有メモリ型ベクトル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．

バケツソートを高速化するためには，ヒストグラムの生成と累計の計算の高速化がポイントとなる．

我々の開発したRetryアルゴリズムは，粒子シミュレーションの分野においてきわめて頻繁に現れるParticle Pusherを非常に高速にベクトル実行できる．このアルゴリズムを用いて，Particle Pusherの一種であるヒストグラム生成の高速化を行った．

また，累計の計算に対しては，配列データ（ヒストグラム）に対する並列処理の方向を変更することにより大きな高速化を達成した．

以上の手法を，NPB ISに実際に適用して，NEC SX-4上でその実行時間を測定した．その結果，Retryアルゴリズムは従来のWork Vectorアルゴリズムに比べて4倍近い性能を示した．また，メモリサイズについても，Work Vectorアルゴリズムに比べて優れていた．並列実行においては，8CPUで約6倍の台数効果が得られた．

なお，本稿で開発した手法をNPB ISに適用して得られた結果は，1996年11月にNASA Ames Research Centerに登録されている[9]．

謝辞 本研究にあたり，有益なご助言をいただいたNECコンピュータ事業部松本寛主席技師長，同C&Cメディア研究所中田登志之研究部長，また測定を手伝っていただいた同スーパーコンピュータ販売推進本部藤森猛夫氏に感謝いたします．

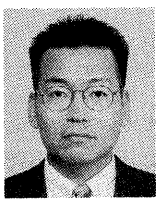
参考文献

1) Knuth, D.E.: *The Art of Computer Programming*, Vol.3, Addison-Wesley (1973).
2) Blelloch, G.E., Dagum, L., Smith, S.J., Thearling, K. and Zagha, M.: An Evaluation of Sorting as a Supercomputer Benchmark, RNR Technical Report, RNR-93-002 (1993).
3) Bailey, D., Barszcz, E., Barton, J., Browning, 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).
4) 津田孝夫：数値処理プログラミング，岩波書店 (1988).
5) 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).
6) Hockney, R.W. and Eastwood, J.W.: Computer Simulation Using Particles, *Institute of Physics* (1988).
7) Nishiguchi, A., Orii, S. and Yabe, T.: Vector Calculation of Particle Code, *Journal of Computational Physics*, Vol.61, p.519 (1985).
8) 登家正夫，古勝紀誠，渡辺貞ほか：スーパーコンピュータSX-4シリーズ特集，NEC技報，Vol.48, No.11, pp.1–74 (1995).
9) Saini, S. and Bailey, D.H.: NAS Parallel Benchmark Results 11-96, *Report NAS-96-17* (1996).

(平成9年11月5日受付)
(平成10年4月3日採録)

村井 均（正会員）

1971年生．1994年京都大学工学部情報工学科卒業．1996年同大学院情報工学専攻修士課程修了．同年NEC入社．現在，C&Cメディア研究所に勤務．並列化コンパイラの研究開発に従事．



末広 謙二（正会員）

1965年生．1988年京都大学工学部電気工学科卒業．1990年同大学院情報工学専攻修士課程修了．1993年同博士課程単位取得退学．同年NEC入社．現在，C&Cメディア研究所主任．並列計算機の基本ソフトウェアの研究開発に従事．

妹尾 義樹（正会員）

1961年生．1984年京都大学工学部情報工学科卒業．1986年同大学院修士課程修了．同年NEC入社．以来C&C研究所にてスーパコンピュータの研究開発に従事．特に並列処理アーキテクチャ，分散メモリマシンのための並列化言語環境，並列アルゴリズムに興味を持つ．現在C&Cメディア研究所研究専門課長．工学博士．1988年本会論文賞受賞．