プログラミング-言語・基礎・実践- 16－18
(1994. 3. 10)

共有メモリシステム上でスナップショット問題を解く分散アルゴリズム

井上美智子　陳慰　増澤利光　都倉信樹

大阪大学基礎工学部

〒560 豊中市待兼山町 1-1

あらまし 複数の非同期プロセスが共有メモリを介して通信を行なうシステム上でスナップショットオブジェクトを実現する分散アルゴリズムを提案する．このオブジェクトは，プロセス数 n のセグメントに分かれる共有データ構造で，各プロセスはそれぞれ1セグメントを所有している．各プロセスは，自分のセグメントの値を更新する操作（$UPDATE$）と全セグメントの値を瞬時に読み込む操作（$SCAN$）を行なうことができる．アルゴリズムの評価は，スナップショットオブジェクトの各操作の実現に必要な共有メモリへの操作数で行なう．本稿では，オブジェクトの各操作を，共有メモリへの $O(n)$ 回の操作で実現する．また，順序保存数値付け換え問題に関しても，これを解く効率の良いアルゴリズムを提案する．

和文キーワード 共有メモリシステム，無待機アルゴリズム，線形化可能性，スナップショット問題，順序保存数値付け換え問題

Efficient Implementation of Atomic Snapshot Object on Shared Memory Distributed System

Michiko Inoue　Wei Chen
Toshimitsu Masuzawa　Nobuki Tokura

Faculty of Engineering Science, Osaka University

Toyonaka-shi, 560 Japan

Abstract We present two efficient wait-free algorithms on the shared memory system. One implements the atomic snapshot object, and the other solves the order-preserving problem. A snapshot object shared by n processes is a data structure partitioned into n segments such that each process owns one segment. Each process can *update* its own segment, and instantaneously *scan* all segments. In our implementation, each operation on the object requires $O(n)$ operations on atomic registers. We also present an order-preserving alrorithm, in which each process requires $O(n)$ operations on atomic registers.

key words shared memory system, wait-free algorithm, linearizability, atomic snapshot, order-preservimg problem

1 まえがき

複数個の非同期プロセスと，それらに共有される複数のレジスタからなる共有メモリシステム上で，協調問題を解く分散アルゴリズムを考える．このうち，各プロセスが他のプロセスの実行速度に関わらず，レジスタへの有限回の操作で解を得るものは，無待機（wait-free）アルゴリズムと呼ばれる．無待機アルゴリズムでは，実行速度の遅いプロセスや故障して停止したプロセスにシステム全体の効率が影響されることがない．近年，共有メモリシステム上の無待機アルゴリズムに関する研究が盛んに行なわれている．

共有メモリシステムでは，各プロセスが非同期に動作し，プロセス間の通信はレジスタを介してのみ行なわれるので，無待機アルゴリズムの設計や検証は複雑である．同期を取るためのプリミティブは，アルゴリズムの設計や検証を簡単にする．さらに，効率の良いプリミティブを与えれば，それを利用するアルゴリズムの効率も良くすることができる．スナップショットオブジェクトは，このようなプリミティブとして注目されている．スナップショットオブジェクトは，コンセンサス問題[1]，共有オブジェクトの実現問題[2]などを解くアルゴリズムで利用されている．

プロセス数を n と表す．スナップショットオブジェクトは，n 個のセグメントからなる共有データ構造で，各プロセスは，それぞれ1つのセグメントを所有している．各プロセスは，自分のセグメントの値を更新する操作（$UPDATE$），および全てのセグメントの値を瞬時に読み込む操作（$SCAN$）を行なうことができる．

これまで，スナップショットオブジェクトを実現する多くの無待機アルゴリズムが提案されている．Attiya, Rachman は，（スナップショットオブジェクトの）各操作を $O(n \log n)$ 回のレジスタへの操作で実現するアルゴリズム[3]を提案している．また，Attiya, Herlihy, Rachman は，各操作を $O(n)$ 回の $Test\&Set$ オブジェクトまたはレジスタへの操作で実現するアルゴリズムを提案している[4]．$Test\&Set$ オブジェクトはレジスタより抽象度の高い共有オブジェクトである．Kirousis, Spirakis, Tsigas は，同時に $SCAN$を行なうプロセス数が高々1 であるときに，各操作を $O(n)$ 回のレジスタへの操作で実現するアルゴリズムを提案している[5]．Israeli, Shaham, Shirazi は，各操作を $O(f(n))$ 回のレジスタへの操作で実現するアルゴリズムを，任意の一方の操作は $O(n)$ 回のレジスタへの操作で実現できるアルゴリズムに変換する手法を提案している[6]．Hoepman, Tromp は，各セグメントのサイズが1ビットであるスナップショットオブジェクトを，各操作に対して $O(n)$ 回のレジスタへの操作で，任意のスナップショットオブジェクトに変換する手法を提案している[7]．また，文献 [4] では，$UPDATE$操作の実現には，$\Omega(n)$ 回のレジスタへの操作が必要であることが証明されたと報告されている．

文献 [4] では，束合意（lattice agreement）問題を解くアルゴリズムを，スナップショットオブジェクトを実現するアルゴリズムに変換する手法が提案されている．本稿では，各プロセスが $O(n)$ 回のレジスタへの操作で束合意問題を解くアルゴリズムを提案し，結果的に，スナップショットオブジェクトの各操作を $O(n)$ 回のレジスタへの操作で実現する無待機アルゴリズムを提案する．

本稿で扱うレジスタは，1つのレジスタに対して読み書きするプロセス数に制限のない multi-writer multi-reader register である．これに対し，文献 [3], [5], [6], [7] では，書き込みを行なうプロセス数が1に制限されている single-writer multi-reader register が用いられている．しかし，single-writer という制限がない場合でも，これらの結果を改善するアルゴリズムは提案されていなかった．すなわち，本稿では，multi-writer multi-reader register からなる共有メモリシステム上でスナップショットオブジェクトを実現する無待機アルゴリズムに関して，既知の結果を改善する．

また，本稿では，束合意問題を解くアルゴリズムのアイディアを利用して，順序保存数値付け変え問題を効率良く解く無待機アルゴリズムを提案する．順序保存数値付け変え問題とは，各プロセスの入力値を，入力値間の順序関係を保存して，より小さいサイズの出力値に書き換える問題である．これまで，各プロセスが $O(n^3)$ 回のレジスタへの操作でこの問題を解く無待機アルゴリズムが知られていた．本稿では，各プロセスが $O(n)$ 回のレジスタへの操作でこの問題を解く無待機アルゴリズムを提案する．

2 諸定義

2.1 状態機械

共有メモリシステムの定義をするために，まず，状態機械の定義を行なう．**状態機械** A は以下の要素からなる．

- $States(A)$：状態の集合で初期状態集合を含む．
- $Out(A)$：出力イベントの集合．
- $In(A)$：入力イベントの集合．
- $Steps(A)$：3項組 (s, e, s') で与えられる状態遷移関数の集合．ただし，s, s'は状態，eはイベントである．

3項組 $(s, e, s') \in Steps(A)$ であることを，イベント e は状態 s で発生可能であるという．入力イベントは任意の状態で発生可能であるイベントである．

状態機械 A に対し，有限系列 $s_0, e_1, s_1, \cdots, e_m, s_m$ または無限系列 $s_0, e_1, s_1, \cdots$ を**状態遷移系列**と呼ぶ．ただし，s_0 は A の初期状態，各 $(s_i, e_{i+1}, s_{i+1}) \in Steps(A)$ である．状態遷移系列からイベントだけを取り出した部分系列を**履歴**と呼ぶ．

出力イベントが相異なる複数の状態機械 $A_1, A_2, \cdots, A_n$ から新たに状態機械 Bを構成することを考える．各要素は以下のように構成する．

- $States(B)=\{(s_1, s_2, \cdots, s_n) | s_i \in State(A_i), i = 1, \cdots, n\}$.
  Bの初期状態は状態を表す n 項組の要素がそれぞれ構成要素の状態機械の初期状態であるものである．
- $Out(B) = \bigcup_i Out(A_i)$.
  構成要素の状態機械の全出力イベントの集合である．
- $In(B) = \bigcup_i In(A_i) - \bigcup_i Out(A_i)$.
  構成要素の状態機械の全イベントのうち出力イベントでないものの集合である．
- $Steps(B)$：任意の i に対し，以下の条件のどちらかが成り立つ3項組 $((s_1, s_2, \cdots, s_n), e, (s'_1, s'_2, \cdots, s'_n))$.
  (1) $(s_i, e, s'_i) \in Steps(A_i)$.
  (2) $e \notin Out(A_i) \cup In(A_i)$, かつ $s_i = s'_i$.
  イベント e を持つすべての状態機械に対し，状態 $(s_1, s_2, \cdots, s_n)$ で e が発生可能であるとき，e が発

生して状態 $(s'_1, s'_2, \cdots, s'_n)$ に遷移することを表す．

状態機械 Bが A を含む複数の状態機械から構成されているとする．Bの履歴 Hに対し，A に関するイベントだけを取り出したHの部分系列を部分履歴といい，$H|A$ と表す．

2.2 プロセス，レジスタ，システム

共有メモリシステムは，複数個の非同期プロセスと複数個のレジスタからなるシステムである．プロセス間の直接通信は行なわれず，情報の交換は共有するレジスタの読み書きによって行なう．各プロセスは，レジスタに対する読み込み，書き込み等の操作を逐次的に行なう．レジスタは複数のプロセスからの操作を同時に処理することができる．このような，共有メモリシステムを状態機械を用いて定義する（図1）．

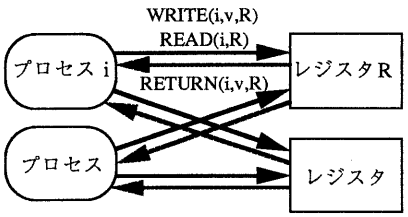

図1．共有メモリシステム

プロセス i は出力イベントとして $WRITE(i,v,R)$, $READ(i,R)$ を，入力イベントとして $RETURN(i,v,R)$ を持つ状態機械である（vは値，Rはレジスタである）．また，**レジスタ**は出力イベントとして $RETURN(i,v,R)$ を，入力イベントとして $WRITE(i,v,R)$, $READ(i,R)$ を持つ状態機械である（iはプロセス，vは値である）．共有メモリシステムはプロセス $1, 2, \cdots, n$ と複数個のレジスタから構成される状態機械である．以降，プロセスの個数を n で表す．

$WRITE$, $READ$をコール，$RETURN$をレスポンスと呼ぶ．コールとレスポンス中に現れるプロセス名とレジスタ名がそれぞれ等しいとき，2つのイベントはマッチするという．履歴 Hにおいて，コールとそれ以降で初めてマッチするレスポンスの対を操作と呼ぶ．以降では，まぎらわしくなければ，コールが $WRITE$（$READ$）である操作のことも $WRITE$（$READ$）と呼ぶ．履歴 Hにおいて，あるコール eに対し，e 以降に e とマッチするレスポンスがないとき，eはペンディングしているという．以下の条件を満たす履歴 Hを**逐次的**であるという．

(1)Hの最初のイベントはコールである．

(2)Hではコールとレスポンスが交互に現れ，コールとその直後のレスポンスはマッチする．

逐次的でない履歴は**並行する**という．

各プロセスは逐次的に操作を行なう．すなわち，システムの履歴 Hに対し各プロセス i に関する部分履歴 $H|i$ は逐次的である．プロセスとは異なり，レジスタの履歴は並行する．レジスタの振舞いを定義するために，まず，履歴 H上での操作間の半順序関係 $\xrightarrow{H}$ を定義する．

定義 1 履歴 H中の2つの操作 OP_1, OP_2に対し，OP_1のレスポンスが OP_2のコールより前にあるとき，$OP_1 \xrightarrow{H} OP_2$ と定義する． ■

逐次的な履歴 Hでは $\xrightarrow{H}$ は全順序関係になる．逐次的な履歴 Hにおいて，各操作は $\xrightarrow{H}$ の順に1つずつ作用する．各 $WRITE(v,R)$ は Rの値を vに更新し，各 $READ(v,R)$ は，それが行なわれるときの Rの値を返す．各レジスタには，初期状態に依存する初期値が設定されている．

並行する履歴の正当性条件として線形化可能性(linearizability)[8]を採用する．

定義 2 システムの各履歴 Hに対し，履歴 H'と逐次的な履歴 Sが存在し，以下の3条件が成り立つとき，システムは線形化可能であるという．

(1)H'は H中のペンディングしているコールにマッチするレスポンスを Hの後に付け加えた履歴である．

(2) すべてのプロセス i に対し，$H'|i = S|i$.

(3)$\xrightarrow{H'} \subseteq \xrightarrow{S}$. ■

定義より，線形化可能な履歴では，各操作はその操作が行なわれた時間内のある時刻で作用するとみることができる．

2.3 分散問題，分散アルゴリズム

共有メモリ上システムで考えられる分散問題には，決定問題と永続的問題がある．決定問題は各プロセスが入力値に対する出力値を1回求めて終了する．永続的問題は永続的オブジェクトというものを実現する問題で，各プロセスは何度でも永続的オブジェクトに対する操作を行なう．各操作では入力値に対して出力値を求める．すなわち，どちらの問題も入力値集合，出力値集合，およびその間の条件によって与えられる．与えられた問題に対して，各プロセスが任意の入力値から，正しい出力値を得るかどうかは，個々のプロセスがレジスタにどのような操作を行なうかに依存する．すなわち，各プロセスの出力イベントに関する状態遷移関数によって決まる．ある問題の任意の入力値に対して，正しい出力値を得ることができる状態遷移関数の集合を，その問題を解く分散アルゴリズムと呼ぶ．

本稿では，分散アルゴリズムを各プロセスに対するPASCAL風のプログラムで記述する．すなわち，プログラム中で，レジスタへの操作とそれに続く内部変数の書き換えが，1つの状態遷移関数を表す．プログラム中では，いくつかの手続きをサブルーチンとして用いる．手続きの実行では，レジスタへの複数個の操作が逐次的に行なわれる．履歴 Hにおける2つの手続きの実行 pr_0, pr_1を考える．pr_0中で最後に行なわれたレジスタへの操作を op_0，pr_1中で最初に行なわれたレジスタへの操作を op_1とする．このとき，$op_0 \xrightarrow{H} op_1$であることを $pr_0 \overset{H}{\Longrightarrow} pr_1$と表す．また，$pr_0 \overset{H}{\Longrightarrow} pr_1$が成り立たないことを，$pr_0 \overset{H}{\not\Longrightarrow} pr_1$と表す．

関係 $\overset{H}{\Longrightarrow}$ は半順序関係であり，以下の性質がある．

性質 1 任意の履歴 H中の手続きの実行 a, b, c, d に対して，$a \overset{H}{\Longrightarrow} b$ かつ $c \overset{H}{\Longrightarrow} d$ であるとき，$a \overset{H}{\Longrightarrow} d$ または $c \overset{H}{\Longrightarrow} b$ が成り立つ．

本稿では分散アルゴリズムを，無待機（wait-free）であるものに限定する．無待機アルゴリズムとは，各プロセス

が出力値を得るまでの状態遷移の回数，すなわちレジスタへの操作の回数に，他のプロセスの実行速度に関わらず上限が存在するアルゴリズムである．無待機アルゴリズムでは，遅いプロセスに待たされて多くのプロセスが実行できなくなるといったことがない．また，あるプロセスが故障停止した場合でも，他のプロセスは正しく解を得るという故障耐性を持つ．

3 スナップショット問題

3.1 スナップショットオブジェクト

スナップショット問題は，スナップショットオブジェクトを実現する永続的問題である．スナップショットオブジェクトは，n 個のセグメント $seg_1, seg_2, \cdots, seg_n$ からなり，プロセス i だけが seg_i の値を更新する．各プロセス i は2つの操作 $UPDATE(v)$, $SCAN$ を用いてスナップショットオブジェクトにアクセスすることができる．$UPDATE(v)$ は seg_i の値を v に更新し，$SCAN$ は，ある意味で，ある瞬間のすべてのセグメントの値を返す．$SCAN$ では，n 値ベクトル $view$ が返され，$view_i$ は seg_i の値である．

スナップショットオブジェクトの振舞いに関する定義を行なう．各プロセス i に対して，$UPDATE(v)$ は，コール $UPDATE(i,v)$ とレスポンス $RETURN(i,v)$ からなり，$SCAN$ はコール $SCAN(i,view)$ とレスポンス $RETURN(i,view)$ からなる．操作間の半順序 $\xrightarrow{H}$ をレジスタと同様に定義する．逐次的な履歴 H では $\xrightarrow{H}$ は全順序関係になる．逐次的な履歴 H において，各操作は $\xrightarrow{H}$ の順に1つずつ作用する．i による $UPDATE$ は seg_i を更新し，$SCAN$ は，それが行なわれるときの全セグメントの値を返す．各セグメントには初期状態に依存する初期値が設定されている．

正当性の条件として線形化可能性を採用する．すなわち，スナップショットオブジェクトの各操作はその操作が行なわれた時間内のある時刻で作用するとみることができ，各 $SCAN$ の各 $view_j$ はそれが作用した瞬間の seg_j の値である．

3.2 束合意問題

文献 [4] では，束合意 (Lattice Agreement) 問題と呼ばれる決定問題を解くアルゴリズムを利用したスナップショット問題を解くアルゴリズムが提案されている．

命題 1 束合意問題を解くアルゴリズム A が与えられたとき，これを利用したスナップショット問題を解くアルゴリズム B が存在し，B において，スナップショットオブジェクトの各操作は，A の $O(1)$ 回の実行と $O(n)$ 回のレジスタへの操作で実現できる．■

本稿では，各プロセスのレジスタへの操作数が $O(n)$ である束合意問題を解く分散アルゴリズムを提案し，結果的に，スナップショット問題を解く分散アルゴリズムに対し，レジスタへの操作数が $O(n)$ であるアルゴリズムを提案する．まず，束合意問題の説明を行なう．

半順序関係 $\leq$ を持つ束 S を考える．S の部分集合 $T \subseteq S$ に対する上限，下限の定義は既知であるとする．T の上限を $join(T)$ と表す．

決定問題である束合意問題を定義する．束合意問題とは，束 S の要素を入力値とする問題であり，各プロセスの出力値は，自分の入力値以上，全プロセスの入力値の上限以下であり，任意の2プロセスの出力値は比較可能でなければならない．すなわち，各プロセスの入力値を x_i, 出力値を y_i とすると，以下の3条件が成り立つ．

(L-1) 任意の i に対して，$x_i \leq y_i$.
(L-2) 任意の i に対して，$y_i \leq join(x_1, \cdots, x_n)$.
(L-3) 任意の i, j に対して，$y_i \leq y_j$ または $y_j \leq y_i$.

3.3 アルゴリズム LA

束合意問題を解くアルゴリズム LA を示す．LA では，以下のレジスタが用いられる．

- レジスタの配列 $X[1 \cdots n]$
- レジスタの配列 $\{A_{k,p_k}[1 \cdots 2^{k-1}] \mid (k, p_k$ は，$1 \leq k \leq \lceil \log n \rceil$, $1 \leq p_k \leq 2^{\lceil \log n \rceil - k+1}$ である整数$)\}$

アルゴリズムを図3に示す．V は入力値のドメインを表す．アルゴリズムの概略と，その性質をトップダウンに説明する．LA では手続き $Participant$ をサブルーチンとして利用する．$Participant$ は，この手続きの実行を始めたことのあるプロセス（実行中のプロセスと実行を終了したプロセス）名を集める手続きであり，各プロセスが集めた集合間には包含関係が成り立つ．各プロセス i は，$Participant$ で集めたプロセス名に対応する入力値をレジスタの配列 X から $READ$ して，その上限を LA の出力値とする．

手続き $Participant$ を説明する．概念的には，$Participant$ の実行において各プロセスは，高さ $\lceil \log n \rceil$ の完全2分木を葉から根に向かって移動するととらえることができる（図2参照）．各プロセスは，それぞれ異なる葉から実行し始める．内部節点 v に対して，v の子孫から実行を始めるプロセスを，v の子孫プロセスと呼ぶ．各内部接点 v は，2つのレジスタの配列を持つ．v の各子孫プロセスは，v において，v の2つの配列に対して手続き $Union$ を実行する．

各節点をレベルとポジションという数値で識別する．レベルはその節点と最も近い葉との距離を表す．ポジションは，根から深さ優先探索を行なって，各節点に付けた値であり，同一レベルの節点中で何番目に探索されたかを表す．

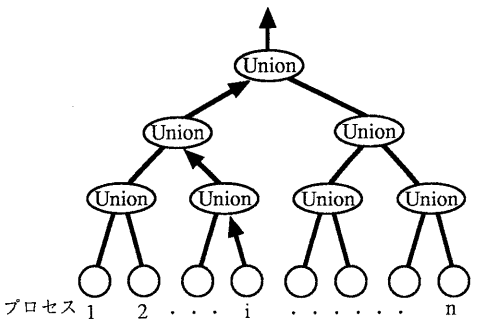

図2．手続き Participant

```
Algorithm LA (code for prosess i)
type SetSize = (set,size);
LA(x_i; return y_i)
begin
  WRITE(i, x_i, X[i]); T := ∅;
  for j ∈ Participant() doT := T ∪ READ(i, X[j]);
  return(y_i = join(T))
end;
Participant(return S)
begin
  S := {i};
  for k := 1 to ⌈log n⌉ do
    g := ⌈i/2^k⌉;
    if odd(g)then S := Union(A^k_g, A^k_{g+1}, S, 2^{k-1})
            else S := Union(A^k_g, A^k_{g-1}, S, 2^{k-1})
  end;
  return(S)
end
Union (A_1, A_2, S, A_size; return U)
var prev_1, prev_2. current_1, current_2: SetSize;
begin
  WriteSet(A_1,S);
  prev_1 := ReadSet(A_1, (S, |S|), A_size);
  prev_2 := ReadSet(A_2, (∅, 0), A_size);
  repeat
    current_1 := ReadSet(A_1, prev_1, A_size);
    current_2 := ReadSet(A_2, prev_2, A_size);
    if prev_1 == current_1 and prev_2 == current_2
      then return (U = prev_1.set ∪ prev_2.set);
    prev_1 := current_1; prev_2 := current_2;
  until true
end
WriteSet(A, S)
begin
  for j := 1 to |S| do WRITE(i, S, A[j])
end
ReadSet(A, max:SetSize, A_size; return max)
begin
  p := max.size;
  while p ≤ max.size do
    for p := p to max.size do
      WRITE(i, max.set, A[p]);
    if p == A_size return(max)
    repeat
      p := p + 1; temp.set := READ(i, A[p]);
      temp.size := |temp.set|;
      if temp.size > max.size then max := temp;
    until max.size ≤ p and temp.set ≠ ∅ and
          p < A_size
  end;
  return(max)
end
```

図3. アルゴリズム LA

レベル k, ポジション p の節点を $v_{k,p}$ と表す. $v_{k,p}$ で実行される $Union$ を $Union_{k,p}$ と表し, レベル k, ポジション p の $Union$ と呼ぶ. $Union_{k,p}$ は節点 $v_{k,p}$ の子孫プロセスによって実行される. アルゴリズムの記述では, $Union_{k,p}$ は陽には現れないが, 引数として $A_{k,2p-1}$ と $A_{k,2p}$ を持つ $Union$ の実行が, $Union_{k,p}$ に相当する. 各プロセス i は, 葉 $v_{0,i}$ の親でのレベル 1 の $Union$ から, 根でのレベル $\lceil \log n \rceil$ の $Union$ までの実行を行なう.

$Union$ は, 入力値, 出力値がともにプロセス名の集合である. プロセス i に対して, レベル 1 の $Union$ の入力値は $\{i\}$, レベル k ($k \geq 2$) の $Union$ の入力値はレベル $k-1$ の $Union$ の出力値である. $Union_{k,p}$ の実行では, $Union_{k,p}$ に対するいくつかの入力値の和集合を求める. このとき, 同一節点に対する $Union$ の実行で求められた集合間には包含関係が成り立つ.

$Union_{k,p}$ では, 2つのレジスタの配列 A_1, A_2 を用いる. ここで, A_1, A_2 は $Union_{k,p}$ の引数で指定される配列で, それぞれ, 配列 $A_{k,2p-1}$, $A_{k,2p}$ のどちらかである (左の子の子孫プロセスは $A_{k,2p-1}$ を, 右の子の子孫プロセスは $A_{k,2p}$ を A_1 とする). $A_{k,2p-1}$, $A_{k,2p}$ のサイズは 2^{k-1}, すなわち, 左, および右の子の子孫プロセス数である. A_1, A_2 の各レジスタの初期値は $\emptyset$ である. 配列に対する操作は, 集合の値を書き込む手続き $WriteSet$ と集合の値を読み込む手続き $ReadSet$ で行なう. このとき, 同一の配列に対する2つの $ReadSet$ の実行では, 後から実行された方がより大きい集合を返す. 各プロセスは, まず, A_1 に入力値を書き込んで, A_1, A_2 を交互に読み込むことを繰り返す. 各配列に対して, 同じ値を連続して読み込んだとき, 2つの配列から読み込んだ値の和を返す.

手続き $WriteSet$, $ReadSet$ を説明する. 配列 A に集合 S を書き込む $WriteSet$ では, レジスタ $A[1]$ から $A[|S|]$ までに順に S を $WRITE$ する. 配列 A から読み込みを行なう $ReadSet$ では, レジスタ $A[1]$ から順に $\emptyset$ が返されるまで, または A の要素をすべて読み込むまで $READ$ を行ない, それまでに読み込んだ最大の集合を返す. ただし, 各 $ReadSet$ の実行では, A に対してそれ以前に実行した $WriteSet$, または $ReadSet$ で $\emptyset$ でないと分かっているレジスタへの $READ$ は行なわない. また, $A[j]$ から大きさが j より大きな集合 S を $READ$ したときは, $A[j+1]$ から $A[|S|]$ までに順に S を $WRITE$ して, $A[|S|+1]$ から $READ$ を続ける.

3.4 LA の正当性

アルゴリズム LA の正当性を証明する. 証明はボトムアップに行なう. まず, $ReadSet$ に対する性質を示す.

補題 1 任意の履歴 H に対して, 任意の配列 A への $ReadSet$ の任意の2つの実行 rs_0, rs_1 が, それぞれ S_0, S_1 を返すとする. このとき, $WriteSet$ によって A に書き込まれた任意の2つの集合 S, S' に対して, $S \subseteq S'$ または $S' \subseteq S$ が成り立つとき, $rs_0 \stackrel{H}{\Longrightarrow} rs_1$ ならば $S_0 \subseteq S_1$ が成り立つ.

(証明) 逆を仮定する. アルゴリズムより, A から読み込まれる値は, A に書き込まれる値または初期値 $\emptyset$ なので, $S_1 \subset S_0$ が成り立ち, $|S_1| < |S_0|$ である.

rs_1 中でのレジスタへの最後の操作を op_1 とする. op_1 は $A[|S_1|+1]$ から $\emptyset$ を $READ$ する. rs_0 を行うプロセスを i とする. アルゴリズムより, i は rs_0 を終える前に, 各 $A[p]$ ($1 \leq p \leq |S_0|$) が $\emptyset$ でないことを知っている. すなわち, $A[|S_1|+1]$ に対しても, $\emptyset$ でない値の $WRITE$, または $READ$ である操作 op_0 を行なっている.

ここで, $rs_0 \stackrel{H}{\Longrightarrow} rs_1$ より, $op_0 \stackrel{H}{\longrightarrow} op_1$ が成り立つ. ところが, アルゴリズム中では, $\emptyset$ を $WRITE$ することはないので, op_1 が $\emptyset$ を $READ$ することに矛盾する. よって, 補題が成り立つ. ■

次に, 任意の $Union$ の実行に対して成り立つ性質を示す. 要素が集合である集合 S に対し, S の任意の2つの要

素 s, s'に対して $s \subseteq s'$ または $s' \subseteq s$ が成り立つことを $Inclusion(S)$ と表す.

補題 2 任意の履歴 Hに対して, 任意の$Union_{k,p}$の全ての実行に対する $v_{k,p}$の左の子, 右の子の子孫プロセスの入力値の集合を, それぞれ, L, R, 出力値の集合を Out とする. このとき, $Inclusion(L)$, $Inclusion(R)$ がともに成り立てば, 以下の (1), (2) が成り立つ.
(1)$Union_{k,p}$の任意の実行の入力値 S_{in}と出力値 S_{out}に対して $S_{in} \subseteq S_{out}$ が成り立つ.
(2)$Inclusion(Out)$ が成り立つ.

((1) の証明) $Union_{k,p}$の実行では, 各プロセスは, まず, 配列 A_1に入力値 S_{in}を書き込む. A_1, A_2から, それぞれ最後に読み込んだ集合を S^1, S^2とすると, $S_{out} = S^1 \cup S^2$. また, S^1は A_1に S_{in}が書き込まれた後に読み込まれるので, $|S_{in}| \leq |S_1|$ である. $Inclusion(L)$, $Inclusion(R)$ より, $S_{in} \subseteq S^1$が成り立ち, $S_{in} \subseteq S_{out}$ である.
((2) の証明) $Union_{k,p}$の任意の 2 つの実行を U_i, U_j, それらの出力値を, それぞれ S_i, S_jとする. このとき, $S_i \subseteq S_j$ または $S_j \subseteq S_i$が成り立てば (2) が証明される.

アルゴリズムより, S_iは, 配列 $A_{k,2p-1}$, $A_{k,2p}$から $ReadSet$ で読み込んだ値 (それぞれ, S_i^1, S_i^2とする) の和であり, S_i=$S_i^1 \cup S_i^2$である. 同様に, S_j^1, S_j^2が定義でき, S_j=$S_j^1 \cup S_j^2$ である. 配列 $A_{k,2p-1}$, $A_{k,2p}$は, それぞれ, $v_{k,p}$の左の子, 右の子の子孫プロセスが入力値を書き込む配列なので, S_i^1, $S_j^1 \in L$, S_i^2, $S_j^2 \in R$である.

$Inclusion(L)$ より, 一般性を失うことなく, $S_i^1 \subseteq S_j^1$ とする. S_i^1が S_j^1に真に含まれるかどうかで場合分けする.
(a)$S_i^1 = S_j^1$の場合
$Inclusion(R)$ より, $S_i^2 \subseteq S_j^2$または $S_j^2 \subseteq S_i^2$である. よって, $S_i \subseteq S_j$または $S_j \subseteq S_i$が成り立つ.
(b)$S_i^1 \subset S_j^1$の場合
U_i, U_jそれぞれに対して, レジスタへ対する最後の 4 つの $ReadSet$ の実行を考える. 4 つの実行では, $A_{k,2p-1}$, $A_{k,2p}$を交互に読んで, それぞれの配列に対して, 連続する 2 つの $ReadSet$ が同じ値を返す. これらの $ReadSet$ の実行のうち, 以下のものを証明に用いる.
$prev_i^2$: U_i中での $A_{k,2p}$への最後から 2 番目の実行
$last_i^1$: U_i中での $A_{k,2p-1}$への最後の実行
$prev_j^1$: U_j中での $A_{k,2p-1}$への最後から 2 番目の実行
$last_j^2$: U_j中での $A_{k,2p}$への最後の実行

各プロセスの部分履歴は逐次的なので, $prev_i^2 \overset{H}{\Longrightarrow} last_i^1$, かつ $prev_j^1 \overset{H}{\Longrightarrow} last_j^2$である. また, $S_i^1 \subset S_j^1$と補題 1 より, $prev_j^1 \overset{H}{\not\Longrightarrow} last_i^1$ よって, 性質 1 より, $prev_i^2 \overset{H}{\Longrightarrow} last_j^2$が成り立ち, 補題 1 より, $S_i^2 \subseteq S_j^2$となり, $S_i \subseteq S_j$が成り立つ. ■

$Participant$ は各プロセス iの入力値 $\{i\}$ に対し, プロセス名の集合 S_iを返す手続きである. 任意の履歴 Hにおいて, 以下の条件を満たすことを証明する.
(P-1) すべての i に対して, $i \in S_i \subseteq \{1, \cdots, n\}$.
(P-2) 任意の i, jに対して, $S_i \subseteq S_j$または $S_j \subseteq S_i$.
(P-3) 任意の i, jの $Participant$ の実行を pp_i, pp_jとするとき, $pp_i \overset{H}{\Longrightarrow} pp_j$ならば, $j \notin S_i$.

補題 3 任意の履歴 Hに対して, 各 iの $Participant$ の出力値を S_iとすると, 上記の (P-1), (P-2), (P-3) が成り立つ.

各条件に対して証明を行なう.
(P-1): $Participant$ はプロセス名の集合を求めるので, $S_i \subseteq \{1, \cdots, n\}$. また, 各プロセス iに対して, レベル 1 の $Union$ の入力値は $\{i\}$ である. 補題 2 を繰り返し適用すると, $\{i\} \subseteq S_i$が成り立ち, (P-1) が成り立つ.
(P-2): 任意の $Union_{k,p}$の出力値間に包含関係が成り立つことを, レベルに関する帰納法で証明する. $Union_{k,p}$に対して, 左の子, 右の子の子孫プロセスの入力値集合を, それぞれ $L_{k,p}$, $R_{k,p}$, 出力値集合を $Out_{k,p}$と表す. 任意の $Union_{1,p}$に対して, 左の子, 右の子の子孫プロセスはそれぞれ高々 1 個なので, $Inclusion(L_{1,p})$, $Inclusion(R_{1,p})$ が成り立ち, 補題 2 より, $Inclusion(Out_{1,p})$ が成り立つ. 任意の p'に対し, $Inclusion(Out_{k-1,p'})$ が成り立つとする. 任意の$Union_{k,p}$に対して, $L_{k,p}$, $R_{k,p}$はレベル $k-1$ の $Union$ の出力値集合なので, $Inclusion(L_{k,p})$, $Inclusion(R_{k,p})$ が成り立ち, 補題 2 より, $Inclusion(Out_{k,p})$ が成り立つ. よって, $Inclusion(Out_{\lceil \log n \rceil,1})$ が成り立ち, (P-2) が成り立つ.
(P-3): レベル kの節点を構成するレジスタ配列をレベル kの配列と呼ぶ. また, $\lceil \log n \rceil = N$とおく.

逆, すなわち, $j \in S_i$を仮定する. よって, iはレベル Nの配列から jが属する集合を読み込んでいる. このとき, 手続きの実行 $ws_1, rs_1, ws_2, rs_2, \cdots, ws_N, rs_N$が存在して, 各 ws_k, rs_kは, それぞれレベル kの配列に対する jが属する集合の $WriteSet$, $ReadSet$ の実行であり, $rs_k \overset{H}{\not\Longrightarrow} ws_k$, かつ $rs_k \overset{H}{\Longrightarrow} ws_{k+1}$が成り立つ. $rs_N \overset{H}{\not\Longrightarrow} ws_N$, $rs_{N-1} \overset{H}{\Longrightarrow} ws_N$, $rs_{N-1} \overset{H}{\not\Longrightarrow} ws_{N-1}$と性質 1 の対偶より, $rs_N \overset{H}{\not\Longrightarrow} ws_{N-1}$が成り立つ. すなわち, $rs_N \overset{H}{\not\Longrightarrow} ws_N$ならば $rs_N \overset{H}{\not\Longrightarrow} ws_{N-1}$が成り立ち, よって,, $rs_N \overset{H}{\not\Longrightarrow} ws_1$が成り立つ. レベル 1 の配列に jを要素とする集合を書き込むのは jだけなので, ws_1は pp_j 中の実行である. ところが, rs_Nは iが実行するので, $pp_i \overset{H}{\Longrightarrow} pp_j$より, $rs_N \overset{H}{\Longrightarrow} ws_1$となり矛盾が生じ, (P-3) が成り立つ. ■

補題 4 アルゴリズム LA は束合意問題を正しく解く.

(証明) 任意の履歴 Hに対して, 任意の 2 プロセス i, jの $Participant$ の実行を pp_i, pp_jとする. (P-3) の対偶より, プロセス iの $Participant$ の出力値に jが属していれば, iが pp_iを終える前に, jは pp_jを始めている. よって, iが $X[j]$ を $READ$する前に, jは $X[j]$ に入力値を $WRITE$ している. よって, iは $Participant$ で求めたプロセス名に対する入力値を集めることができ, LA の出力値は, 集めた入力値集合の上限である. (P-1) より (LA-1), (LA-2) が成り立ち, (P-2) より (LA-3) が成り立つので, LA は束同意問題を正しく解く. ■

補題 5 アルゴリズム LA において, 各プロセスはレジスタに対して $O(n)$ 回の操作を行なう.

(証明) 任意のプロセス iが $Union$ の実行中で行なうレジスタへの操作数を考える. レベル kの $Union$ では, 大きさ 2^{k-1}の 2 つのレジスタの配列 A_1, A_2に対する操作を行なう. iは A_1, A_2の要素であるレジスタ Rから$\emptyset$を $READ$したときに限り, 再び Rに対して操作を行なう. 2 回目の操作も $READ$であり, 2 つの $READ$は異なる $ReadSet$ の実行中で行なわれる. iが A_1の要素に対して 3 回目以降

の $READ$を行なうのべ回数を考える．このとき，i は A_1 に対する $ReadSet$ の連続する 3 つの実行 rs_1, rs_2, rs_3を行ない，rs_1, rs_2は同じ Rから$\emptyset$ を読み込んで同じ集合を返す．アルゴリズムより，iは，A_1，A_2に交互に $ReadSet$ を実行し，各配列から，それぞれ同じ値を連続して読み込むと，それ以上 A_1と A_2 に対する $ReadSet$は行なわない．よって，rs_1とrs_2の間，rs_2とrs_3の間，それぞれで実行される A_2に対する $ReadSet$ は同じ集合を返さず，補題 1 より，集合の大きさは真に大きくなる．A_2から読み込まれる集合の大きさは高々2^{k-1}なので，A_1の要素に対して 3 回目以降の $READ$を行なうのべ回数は高々2^{k-1}であり，A_1の要素に対する操作数は高々$3\cdot 2^{k-1}$である．A_2に対しても同様であるので，レベル kの $Union$では，レジスタに対して高々$6\cdot 2^{k-1}$の操作を行なう．よって，すべての $Union$の実行で行なうレジスタへの操作数は，

$$\sum_{k=1}^{\lceil \log n \rceil} 6\cdot 2^{k-1} = 6(2^{\lceil \log n \rceil}-1) < 6(2n-1) = O(n)$$

となる．

$Union$ の実行以外でのレジスタへの操作は，$X[i]$ に対する 1 回の $WRITE$と，Xの各要素に対する高々 1 回の $READ$である．よって，補題が成り立つ． ■

補題 4，補題 5より以下の定理が成り立つ．

定理 1 アルゴリズム LA は束合意問題を正しく解き，各プロセスはレジスタに対して $O(n)$ 回の操作を行なう

命題 1，定理 1 より，以下の定理が成り立つ．

定理 2 スナップショット問題を解くアルゴリズム Bが存在し，Bにおいて，スナップショットオブジェクトに対する各操作は，$O(n)$ 回のレジスタへの操作で実現できる．

4 順序保存数値付け換え問題

4.1 問題の定義

順序保存数値付け換え問題（OPP）は，各プロセスの入力値を，入力値間の順序関係を保存して，より小さいサイズの出力値に付け換える問題である．各プロセスの入力値は互い異なると仮定する[1]．入力値のドメインを Vとする．V上には全順序関係$<_V$ が定義されている．プロセス iの入力値を x^i, 出力値を y^i と表すと，以下の条件が成り立つ．

(OP-1) 関係$<_{OP}$が定義されている集合 $OPNS$が存在し，$OPNS$は出力値のドメインである．

(OP-2) 任意の i, jに対し，$x^i <_V x^j$ならば，$y^i <_{OP} y^j$が成り立つ．

上記の条件から，出力値の集合 $\{y^1, y^2, \cdots, y^n\}$ 上では関係$<_{OP}$は全順序関係である．OPP を解くアルゴリズムは，以下の 2 つの尺度で評価する．

- 各プロセスのレジスタに対する操作数．
- 出力値のドメイン $OPNS$の大きさ $|OPNS|$.

操作数に関しては，起こり得るすべての履歴に対する最悪値評価を行なう．

[1] 入力値が互いに異なると仮定できない場合は，入力値とプロセス名を新たに入力値と考えれば仮定は成り立つ．

4.2 アルゴリズム RN

文献 [9] では，メッセージパッシングシステム上で OPP を解くアルゴリズムが提案されている．文献 [10] では，文献 [9] での出力値の構成法を用いて，共有メモリシステム上で OPP を解くアルゴリズムが提案されている．本稿で提案するアルゴリズム RNでは，出力値のドメイン $OPNS$は [9]，[10] と同じであるが，出力値の構成法，および関係$<_{OP}$の定義が少し異なる．詳細は文献 [11] を参照されたい．

アルゴリズム RNを説明する．RNでは，手続き $Participant$ のコピーを 2 つ用意する．コピーとは，手続き中で用いるレジスタを別に用意することを意味する．RNでは，LA と同様の手法で，まず，1 つ目のコピーを使って入力値の集合を求める．ここで，求められた入力値集合を参加入力集合と呼ぶ．次に，2 つ目のコピーを使って，参加入力集合の集合 Tを求める．最後に，手続き encode を用いてTから出力値 yを計算する．$encode$ では，レジスタへの操作は行なわず，プロセスの内部計算によって出力値を計算する．

アルゴリズムのアイディアを説明する．$Participant$ の性質から，参加入力集合間には必ず包含関係がある．よって，任意の履歴を 1 つ固定すると，各参加入力集合は，その大きさで識別できる．また，各プロセスが集めた参加入力集合の集合間にも包含関係がある．よって，任意の 2 つのプロセスは，同じ参加入力集合 Sを得ることができ，S 中で自分の入力値が何番目に小さいか（ランクと呼ぶ）の情報から，入力値間の大小関係を求めることができる．出力値は，集めた各参加入力集合 Sに対して，Sの大きさと入力値のランクをコード化したものである．

手続き $encode$ での，出力値の計算方法を説明する．参加入力集合 Sに対する入力値 x のランクを $rank(x,S)$ と表す．$x \notin S$のときは，$rank(x,S)$ は集合 $S\cup\{x\}$ に対して x が何番目に小さいかを表す．

入力値を x とする．出力値への変換は参加入力集合の集合 Tから z, w, yへと段階的に行なう．まず，以下で示す zに変換する．T中の参加入力集合で最大のものの大きさを Tおよび z, w, yの最大入力集合サイズと呼び，max とおく．zは長さ max の系列 $z=(z_1, z_2, \cdots, z_{max})$ で，各 $S\in T$に対して $z_{|S|} = rank(x,S)$ であり，それ以外の要素は 0 である．以降では，z, w, y，それぞれに対して，要素間で添字が小さい（大きい）方を左（右）の要素と呼ぶ．

zのドメイン $OPNS_z$上の関係$<_z$を定義する．

定義 3 $OPNS_z$の要素 z^i, z^jに対して，z^i, z^j の最大入力集合サイズを，それぞれ，max^i, max^jとし，小さい方を m とする（$m = min(max^i, max^j)$）．以下の (1),(2) のいずれかを満たすことを $z^i <_z z^j$と表す．

(1)$z^i_m < z^j_m$.

(2)$z^i_m = z^j_m$，かつ $m < max^i$. ■

(1) は大きさ m の参加入力集合 Sに対して x^iのランクが x^jのランクより小さい場合で，(2) は Sに対するランクが等しく，$x^i \notin S$，かつ $x^j \in S$の場合である．証明は省略するが，上記の関係$<_z$は入力値間の順序関係を保存する．

補題 6 任意の履歴に対して，任意のプロセス i,jが得た参加入力集合の集合が，それぞれ z^i, z^jに変換されたとする．i,jの入力値をそれぞれ，x^i, x^jすると，$x^i <_V x^j$ならば $z^i <_z z^j$が成り立つ． ■

出力値のドメインの大きさを小さくするために，zを要素が 0, 1, 2 だけからなる系列に変換する．$z = (z_1, z_2, \cdots, z_{max})$ に対して，wは長さ max の系列 $w = (w_1, w_2, \cdots, w_{max})$ で，0 でない各 z_iに対して，$w_1 + w_2 + \cdots + w_i = z_i$を満たす．各 w_i の値は，なるべく 0, 1 だけを使い，条件を満たす中で，より大きい数値が（0 よりも 1, 1 よりも 2 が）より左になるように決定する．例えば，$z = (0,3,0,4,0,5,5)$ は，$w = (2,1,1,0,1,0,0)$ に変換される．

上記の wのドメインを $OPNS_w$とすると，$OPNS_w$の要素に関して，定義 3の (1),(2) の評価が行なえるので，$OPNS_w$上にも入力値間の順序関係を保存する関係$<_w$が存在する．

wを出力値 yに変換する．yのドメイン $OPNS$の大きさは $OPNS_w$の約半分になる．$w = (w_1, w_2, \cdots, w_{max})$ に対して，yは長さ max の系列で，$w_{max} = 0$ のとき $y = w$，$w_{max} = 1$ のとき，z 中で 0 でない最右の要素を z_i（ないときは $i = 1$）とすると，$y = (w_1, w_2, \cdots, w_{i-1}, w_i + 1, w_{i+1}, \cdots, w_{max-1}, 0)$ である．

証明は省略するが，$OPNS$の要素に対しても，関係$<_w$は入力値間の順序関係を保存する．$<_w$を$<_{OP}$とする．

補題 7 任意の履歴に対して，任意のプロセス i,jの入力値をそれぞれ，x^i, x^j，出力値を，それぞれ y^i, y^jとする．このとき，$x^i <_V x^j$ならば $z^i <_{OP} z^j$が成り立つ． ■

$OPNS$の構成要素は文献 [9] と同じであるので，その大きさは $2^n - 1$ である．また，この値は下界に等しい[9]．

定理 3 アルゴリズム RNは順序保存数値付け換え問題を解き，RNにおいて，各プロセスのレジスタに対する操作数は $O(n)$，かつ出力値のドメインの大きさは $2^n - 1$ である．

（証明）補題 7より，(OP-1),(OP-2) が成り立ち，RNは順 序保存数値付け換え問題を正しく解く．また，RNにおいて，各プロセスのレ ジスタに対する操作は，2 回の *Participant* の実行中とその前後の $O(n)$ 回 だけであるので，その操作数は $O(n)$ である．さらに，文献 [9] より，出力値のドメインの大きさは $2^n - 1$ である． ■

5 まとめ

共有メモリシステム上で，スナップショットオブジェクトの各操作を $O(n)$ 回のレジスタへの操作で実現する無待機アルゴリズム（n はプロセス数），および，各プロセスが $O(n)$ 回のレジスタへの操作で順序保存数値付け換え問題を解く無待機アルゴリズムを提案し，既知の結果を改良した．

スナップショット問題を解くアルゴリズムに関しては，これまで，各操作を実現するためのレジスタへの操作数に $O(n \log n)$ と $\Omega(n)$ のギャップが存在していた．本稿の結果により．このギャップは埋められたことになる．本稿では，multi-writer multi-reader register を用いたアルゴリズムを提案している．また，文献 [4] で提案された束合意問題を解くアルゴリズムからの変換法を利用しているので，スナップショットオブジェクトを実現するのに必要なレジスタ数は有限でない．今後の課題として，有限個の single-writer multi-reader register を用いて，スナップショットオブジェクトの各操作を $O(n)$ 回のレジスタへの操作で実現するアルゴリズムを開発することが残されている．

謝辞 本研究の一部は，文部省科学研究費補助金による研究成果である．

参考文献

[1] J. Aspnes: "Time- and space-efficient randomized consensus", Proc. of the 9th ACM Symposium on Principles of Distributed Computing, pp. 325–331 (1990).

[2] M. Herlihy: "Wait-free synchronization", ACM TOPLAS, **11**, 1, pp. 124–149 (1991).

[3] H. Attiya and O. Rachman: "Atomic snapshots in O(n log n) operations", Proc. of the 12th ACM Symposium on Principles of Distributed Computing, pp. 29–40 (1993).

[4] H. Attiya, M. Herlihy and O. Rachman: "Efficient atomic snapshots using lattice agreement", Proc. of the 6th Int. Workshop on Distributed Algorithms (LNCS 647), pp. 35–53 (1992).

[5] L. Kirousis, P. Spirakis and P. Tsigas: "Simple atomic snapshots: A linear complexity solution with unbounded time-stamps", Proc. of International Conference on Computing and Information(LNCS 497), pp. 582–587 (1991).

[6] A. Israeli, A. Shaham and A. Shirazi: "Linear-time snapshot protocols for unbalanced systems", Proc. of the 7th Int. Workshop on Distributed Algorithms, pp. 26–38 (1993).

[7] J.-H. Hoepman and J. Tromp: "Binary snapshots ", Proc. of the 7th Int. Workshop on Distributed Algorithms, pp. 18–25 (1993).

[8] M. Herlihy and J. Wing: "Linearizability: A correctness condition for concurrent objects", ACM TOPLAS, **12**, 3, pp. 463–492 (1990).

[9] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg and R. Reischuk: "Achievable cases in an asynchronous environment", Proc. of the 28th IEEE Symp. on Foundations of Computer Science, pp. 337–346 (1987).

[10] A. Bar-Noy and D. Dolev: "Shared-memory vs. message-passing in an asynchronous distributed environment", Proc. of the 8th ACM Symposium on Principles of Distributed Computing, pp. 307–318 (1989).

[11] 井上, 陳, 増澤, 都倉："共有メモリシステム上で順序保存数値付け換え問題を解く分散アルゴリズム", Technical Report 94-ICS-2, 大阪大学基礎工学部情報工学科 (1994).