並列トポロジカル整列アルゴリズム

多田昭雄[†] 右田雅裕[††] 中村良三[††]

無閉路有向グラフ DAG（Directed Acyclic Graph）において節点数 n，辺数 m とするとき，節点のトポロジカル整列を求める従来の並列アルゴリズムは，CRCW-PRAM 計算機モデルで $O\left(n^3\right)$ 個のプロセッサを用いて，$O(\log n)$ 時間で実行するアルゴリズムがある．このアルゴリズムは推移的閉包行列の計算に依存しているので，$O\left(n^3\right)$ 個のプロセッサを減少させることは難しい．本稿では，CREW-PRAM 計算機モデルのもとで，DAG 上における効率良い並列トポロジカル整列アルゴリズムを提案する．提案する並列アルゴリズムは分割統治法によるアルゴリズムの設計法をもとに，基本的な並列アルゴリズムのみを用いて設計される．具体的には，はじめに DAG を入出次数がたかだか 1 の節点からなる線形リストに分割し，次に再帰的に線形リストを併合しながら各節点の最大ランク値を求める．最後に，最大ランク値をもとに節点のトポロジカル整列を行う並列アルゴリズムである．この提案するアルゴリズムの計算量は CREW-PRAM 計算機モデルでプロセッサ数が $O(n+m)$，計算時間が $O(\log^2 m)$ である．

Parallel Topological Sorting Algorithm

Akio Tada,[†] Masahiro Migita[††] and Ryozo Nakamura[††]

The traditional parallel topological sorting algorithm in a DAG (Directed Acyclic Graph) with the number of vertices n and the number of edges m is the one that takes $O(\log n)$ time using $O\left(n^3\right)$ processors on a CRCW-PRAM model. However this algorithm depends on the transitive closure matrix calculation, therefore it is difficult to reduce the number of processors furthermore. In this paper, we propose an efficient parallel topological sorting algorithm in a given DAG, based on a CREW-PRAM model. Namely, based on the divide-conquer method, the proposed algorithm at first divides a given DAG into several linked lists, in which each vertex has at most 1 both input and output edge degree. Next in the merging step, it merges recursively two combined linked lists to get a maximum rank of each vertex. Finally the maximum rank of each vertex is found, then all vertices of a given DAG are ordered so that all directed edges go from left to right, namely, a topological sorting is given. The proposed algorithm requires $O(n+m)$ processors and $O(\log^2 m)$ time on a CREW-PRAM model.

1. はじめに

無閉路有向グラフ DAG（Directed Acyclic Graph, n = 節点数，m = 辺数）に対する 1 つの基本操作として，節点間の半順序（partial order）を示す有向辺をもとに，グラフの節点を線形順序（linear order）に並べ替えるトポロジカル整列がある．すなわちトポロジカル整列はすべての辺の向きが左から右に向かうように節点を線形順序に並べることである[1)]．トポロジカル整列は生産工程のグラフ表示や PERT などに利用されている．トポロジカル整列の並列アルゴリズムとしては CRCW-PRAM 計算機モデル上で実現されたいくつかのアルゴリズム[2)~4)] がある．文献 2) はすべてのプロセッサが完全に同期をとるという仮定のもとに議論されている．その計算量はプロセッサ数が $O(m)$ で計算時間が $O(n)$ を要する．文献 3) は隣接行列から推移的閉包行列を求めてトポロジカル整列を行うアルゴリズムである．その計算量はプロセッサ数が $O\left(n^3\right)$ で計算時間は $O(\log n)$ である．文献 4) は文献 3) に対して行列計算を改良して，プロセッサ数を $O\left(n^3\right)$ から $O\left(n^{2.376}\right)$ に減少させている．しかし，このアルゴリズムは推移的閉包行列の計算に依存しているので，これ以上プロセッサ数を減少させることは難しい．上記の文献以外には新たな研究成果は見当たらない．

本稿では，CREW-PRAM 計算機モデルのもとで，従来の推移的閉包行列の計算方法を用いず，分割統治法によるアルゴリズムの設計法をもとに，基本的な並

† 崇城大学
Sojo University
†† 熊本大学
Kumamoto University

列アルゴリズム[5),6)] のみを用いてプロセッサ数を減少させる効率良い並列トポロジカル整列アルゴリズムを提案する．所与の DAG は各辺を出節点と入節点の組で表した配列として入力される．はじめに，所与の DAG を入出次数がたかだか 1 の節点からなる線形リストに分割する．次に，分割された線形リストの組を再帰的に併合しながら，各節点の最大のランク値を求める．最後に，最大ランク値をもとに節点のトポロジカル整列を行う並列アルゴリズムである．このアルゴリズムの計算量は，プロセッサ数が $O(n+m)$，計算時間が $O(\log^2 m)$ である．従来のアルゴリズムは DAG の疎密に柔軟に対応できないが，提案するアルゴリズムでは，DAG が密で辺の数が $m = O(n^2)$ でもプロセッサ数はたかだか $O\left(n^2\right)$ であり，また疎で辺の数が $m = O(n)$ のときにはプロセッサ数は $O(n)$ となり効率良い並列アルゴリズムである．すなわち，アルゴリズムの性能は時間計算量と領域量によって評価されるが，前者の時間計算量は実際の計算機処理における実行時間の長さと直接に結び付き特に重要である．上記の時間計算量に対応して，並列アルゴリズムの性能は，使用するプロセッサ数と時間計算量との積として定義されるコストによって評価される．したがって，時間計算量が同等であってもプロセッサ数を減少させることはコストの減少になり，より効率の良い並列アルゴリズムとなる．

以下，2 章では，提案する並列アルゴリズムの概要を述べ，3 章では，そのアルゴリズムの詳細化を行い，アルゴリズムの正当性と時間計算量を考察する．

2. 並列トポロジカル整列アルゴリズムの概要

対象とするグラフは，閉路や多重辺を含まない無閉路有向グラフ DAG（節点の集合 V，有向辺の集合 E，|V| = n，|E| = m）とし，節点は 1 から通し番号を付けて表す．所与の DAG は有向辺を配列の形で表し，配列 OV が出節点，配列 IV が入節点を示し，各有向辺は OV[i] から IV[i](1 ≤ i ≤ m) への向きで表される．ただし，$n \leq m$ で，また配列 OV の出節点は節点番号の順に整列されていると仮定する．

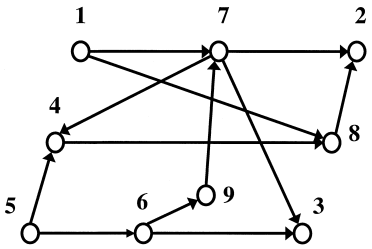
具体例として，**図 1** (a) に DAG の例を示し，図 1 (b) にその有向辺を表す配列の入力例を示す．

並列トポロジカル整列アルゴリズムは大きく分けて，次の 3 つのステージから構成される．

ステージ 1　DAG を入出次数がたかだか 1 の線形リストに分割する並列アルゴリズム

ステージ 2　線形リストの組を再帰的に併合しながら各節点の最大のランク値を求める並列アルゴリズム

ステージ 3　各節点の最大ランク値をもとにトポロジカル整列を行う並列アルゴリズム



(a) DAGの例
(a) Instance of DAG.

	1	2	3	4	5	6	7	8	9	10	11	12
OV	1	1	4	5	5	6	6	7	7	7	8	9
IV	7	8	8	4	6	3	9	2	3	4	2	7

(b) DAGの有向辺を表す配列
(b) Directed edge array of DAG.

図 1　所与の DAG
Fig. 1　A given DAG.

3. アルゴリズムの詳細化

ステージ 1　DAG を入出次数がたかだか 1 の線形リストに分割する並列アルゴリズム

与えられた有向辺を表す配列に基づいて，各節点の入出次数がたかだか 1 となるように節点を分割し，所与の DAG を線形リストに分割する．このとき，所与の節点を**分割節点**と呼び，1 つの節点が分割されて新たに生成された節点を**兄弟節点**と呼ぶ．

ステップ［1.1］　各節点の出次数および入次数を求め，大きい方の値をその節点の最大次数とし，各節点における最大次数をその節点の兄弟節点数とする．ステップ［1.1］のアルゴリズムの詳細を下記に示す．

(1) 各節点の出次数 O[1..n] を求めるアルゴリズム
  (1.1) 出節点の配列 OV[1..m] において，その要素の節点番号が異なる境界の辺番号を配列 B[1..n] の要素として格納する．
  (1.2) 配列 B において，1 つ前の境界の辺番号との差から出次数を求める．
(2) 各節点の入次数 I[1..n] を求めるアルゴリズム
  (2.1) 入節点の配列 IV[1..m] の節点番号を昇順に整列する．
  (2.2) 出次数を求めるアルゴリズム (1) と同様にして入次数を求める．
(3) 各節点の出次数と入次数の大きい方の値すなわち

最大次数（兄弟節点数 C[1..n]）を求める．

ステップ［1.2］ 各節点の兄弟節点数に基づいて，その兄弟節点には連続した通し番号を付ける．このとき所与の DAG の節点番号の大きさの順序を保つように連続した新しい節点番号を付け，所与の DAG を新節点番号に基づいて線形リストに分割する．ステップ［1.2］のアルゴリズムの詳細を下記に示す．

(1) 各節点までの兄弟節点数の累計和を配列 S[1..n] に求める（このとき S[n] には新節点の総数が格納される）．

(2) 旧節点番号（分割節点番号）と新節点番号の対応表 N[1..S[n]] を作成する（以下では，有向辺の集合の要素を記号 e，旧節点の集合の要素を記号 v，および新節点の集合の要素を記号 u で表す）．

(3) ステップ［1.1］で求めた境界の辺番号を表す配列 B と節点数の累計和を表す配列 S を用いて，出節点の節点番号 OV を新節点番号 NOV[1..m] に置き換える．

(4) はじめに，配列 IV の入節点の節点番号を昇順に整列し，次に，出節点に対するアルゴリズム (3) と同様にして，入節点 IV を新節点番号 NIV[1..m] に置き換える．

(5) 新節点番号によって表された有向辺，すなわち出節点の配列 NOV と入節点の配列 NIV に基づき，配列 NEXT[1..S[n]] を用いて新節点を連結する．この結果，所与の DAG は新節点に基づいていくつかの線形リストに分割される（なお線形リストの最後尾の節点の NEXT の要素は仮想の節点番号 0 とする）．

ステップ［1.3］ 各線形リストにリスト番号を付け，各節点にランク値とリスト番号を与える．さらに各分割節点において，それらの兄弟節点をリスト番号順に整列する．ステップ［1.3］のアルゴリズムの詳細を下記に示す．

(1) 各線形リストにおいて，節点にランク値 RANK[1..S[n]] を付ける．ランク値はリストの先頭の節点のランク値を 1 として一連の通し番号を付ける．

(2) 各線形リストにリスト番号を付ける．リスト番号は，まず各リストの先頭の節点番号とするが，次にそのリスト番号を昇順に整列し，1 から一連の通し番号を付ける．

(3) 各線形リストのリスト番号をそのリスト上のすべての節点にブロードキャストする（各節点のリスト番号は LISTN[u] で表す）．

(4) 各分割節点において，兄弟節点のリスト番号が昇順になるように整列する（このとき配列 SIB[v] によって分割節点 v の兄弟節点の集合を表す）．

```
procedure Stage-1(OV[1..n],IV[1..n]);
{ ステージ 1 の並列アルゴリズム }
begin
  { ステップ [1.1] }
  { (1) 出次数を求めるアルゴリズム }
  for v := 1 to n in parallel do { 境界の辺番号 B を求める }
    B[v] ← 0;
  for e := 1 to m-1 in parallel do
    if OV[e] ≠ OV[e+1] then B[OV[e]] ← e;
  B[OV[m]] ← m;
  Broadcast-1(B[1..n]); { 出次数が 0 である節点は前の節点の辺番号で埋める }
  O[1] ← B[1];
  for v := 2 to n in parallel do
    O[v] ← B[v] - B[v-1]; { 出次数 }
  { (2) 入次数を求めるアルゴリズム }
  入節点 IV[1..m] を整列し，出次数と同様にして入次数 I[1..n] を求める
  for v := 1 to n in parallel do
    C[v] ← Max(O[v],I[v]); { 兄弟節点数 }
  { ステップ [1.2] }
  Para_Prefixsum(C[1..n],S[1..n]); { 兄弟節点数 C をもとにその累計和 S を求める }
  N[1] ← 1;
  for v := 2 to n in parallel do begin { 新節点 u と旧節点 v の対応表 N を作る }
    u ← S[v-1]+1; N[u] ← v
  end;
  Broadcast-2(N[1..S[n]]); { N[i]=0 ならば旧節点番号で埋める }
  { (3) 新しい出節点の配列 NOV を求める }
  B[0] ← 0; S[0] ← 0;
  for e := 1 to m in parallel do { 出節点を新節点番号で置換し，新出節点 NOV を求める }
    NOV[e] ← S[OV[e]-1] + (e-B[OV[e]-1]);
  { (4) 新しい入節点の配列 NIV を求める }
  入節点 IV[1..m] を整列し，NOV と同様にして新入節点 NIV[1..m] を求める
  for u := 1 to S[n] in parallel do { NEXT 初期値設定 }
    NEXT[u] ← 0;
  for e := 1 to m in parallel do { 線形リストの生成 }
    NEXT[NOV[e]] ← NIV[e];
  { ステップ [1.3] }
  for u := 1 to S[n] in parallel do begin { 初期値設定 }
    LISTN[u] ← 0; RANK[u] ← 1
  end;
  List_Ranking(RANK[1..S[n]]); { 各線形リスト上の新節点のランク値を計算 }
  for u := 1 to S[n] in parallel do { リストの先頭の節点番号を仮リスト番号とする }
    if RANK[u] = 1 then LISTN[u] ← u;
  Para_Sort(LISTN[1..S[n]]); { 仮リスト番号を整列して 1 からの通し番号を付ける }
  Broadcast(LISTN[1..S[n]]); { 新リスト番号をすべての節点にブロードキャストする }
  for v := 1 to n in parallel do
    Para_Sort(SIB[v]) { 各分割節点で兄弟節点をリスト番号順になるように整列 }
end;
```

図 2 ステージ 1 のアルゴリズム
Fig. 2 Parallel algorithm for Stage-1.

ステージ 1 のアルゴリズムの概要を擬似コードによって**図 2** に示す．図 2 において用いる既知の並列アルゴリズム，並列プレフィックス和を求める Para_Prefixsum，並列リストランキング List_Ranking，並列ブロードキャスト Broadcast および並列整列アルゴリズム Para_Sort は，その詳細な記述は省略し，その手続き名だけを記した．また Broadcast-1 アルゴリズムは既知のアルゴリズム Broadcast と若干異なり，配列 B の要素の値が 0 すなわち出次数が 0 である節点は前の

```
procedure Broadcast-1(B[1..n]);
begin
   for v := 1 to n-1 in parallel do
      Q[v] ← v+1;
   Q[n] ← n;
   repeat log n times
      for v := 1 to n in parallel do
         if Q[v] < n then begin
            if B[Q[v]] = 0 then
               B[Q[v]] ← B[v];
            Q[v] ← Q[Q[v]]
         end
end;
```

図 3 Broadcast-1 アルゴリズム
Fig. 3 Broadcast-1 algorithm.

節点の辺番号で埋めるアルゴリズムであるから，その詳細を**図 3** に示す．Broadcast-2 は Broadcast-1 とほとんど同様であるので省略する．なお擬似コードの中で定義した配列などの初期値はすべて 0 と仮定する．

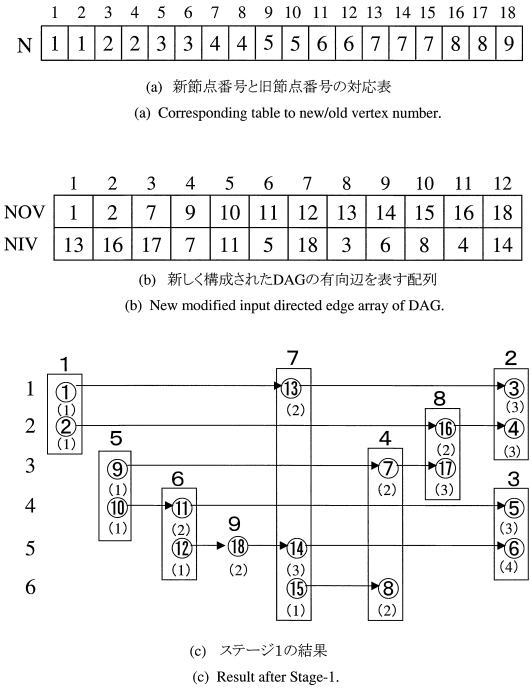
ステージ 1 のアルゴリズムの正当性と計算量に関する考察

ステップ［1.1］では，配列 OV の出節点は節点番号順に整列されているので，その出次数を並列に算定するアルゴリズムの正当性は明らかである．また入次数は配列 IV の入節点の節点番号を昇順に整列し，出次数を求めたと同じアルゴリズムで処理することができる．この出次数を求める計算量は，各辺に 1 台のプロセッサを割り付けると，プロセッサ数 $O(m)$ で，計算時間はブロードキャスト（Broadcast-1）に要する時間 $O(\log n)$ である．入次数は配列 IV を昇順に整列する必要があるので，並列整列アルゴリズム[7]（Para_Sort）を用いれば $O(\log m)$ 時間でできる．また，各分割節点の兄弟節点数は出次数と入次数の大きい方を選べばよいので定数時間 $O(1)$ でできる．したがって，このステップの計算量は，プロセッサ数は $O(m)$ で，計算時間は $O(\log m)$ である．

ステップ［1.2］では，兄弟節点数の累計和 S[1..n] は並列プレフィックス和アルゴリズム[5]（Para_Prefixsum）を用いて求めることができる．そのほかにも Broadcast-2 アルゴリズムや並列整列アルゴリズムを用いてアルゴリズムを設計しているので，その正当性は明らかである．計算量は，分割した線形リストの新節点数はたかだか $(n+m-1)$ 個であるので，各節点に 1 台のプロセッサを割り付けると，プロセッサ数は $O(n+m)$ で，最大の計算時間は入節点の配列 IV を昇順に整列する時間 $O(\log m)$ となる．

ステップ［1.3］では，まず各線形リストにおける節点のランク値は先頭の節点のランク値を 1 として，ダブリング法[5]で算定できる．次に各リストに一連の通し番号を与えるとき，および兄弟節点をリスト番号順に整列するときに並列整列アルゴリズムを用いる．したがって既知の並列アルゴリズムのみを用いてアルゴリズムを設計できるので，その正当性は明らかである．計算量は，線形リストの数および分割節点における兄弟節点の数はたかだか $(m-1)$ 個なので，並列整列アルゴリズムを用いて $O(\log m)$ 時間でできる．また各節点のランク値の算定と各節点へのリスト番号のブロードキャストは線形リスト上の節点数がたかだか $(n-1)$ 個なのでダブリング法を用いて $O(\log n)$ 時間でできる．プロセッサ数はいずれも $O(n+m)$ なので，このステップの計算量は，プロセッサ数は $O(n+m)$ で，計算時間は $O(\log m)$ となる．

以上の考察から，ステージ 1 の並列アルゴリズムはプロセッサ数は $O(n+m)$ で，計算時間は $O(\log m)$ である．



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
N	1	1	2	2	3	3	4	4	5	5	6	6	7	7	7	8	8	9

(a) 新節点番号と旧節点番号の対応表
(a) Corresponding table to new/old vertex number.

	1	2	3	4	5	6	7	8	9	10	11	12
NOV	1	2	7	9	10	11	12	13	14	15	16	18
NIV	13	16	17	7	11	5	18	3	6	8	4	14

(b) 新しく構成されたDAGの有向辺を表す配列
(b) New modified input directed edge array of DAG.

(c) ステージ1の結果
(c) Result after Stage-1.

図 4 ステージ 1 の処理
Fig. 4 Process of Stage-1.

［具体例］

図 1 (a) の具体例において，ステップ［1.2］における新節点番号と旧節点番号の対応表を**図 4** (a) の配列で示す．このとき添字（インデックス番号）が新節点番号で，配列の要素は旧節点番号（分割節点番号）である．そして図 1 (b) の入力された有向辺を表す配列は図 4 (b) のように新しい節点番号に置き換わり，それらを並列になぞれば，図 4 (c) のような 6 つの線形リストに分割される．図 4 (c) において，左端列の数

字はリスト番号，○内の数字は新節点番号，矩形の上部の数字は分割節点番号，および ( ) 内はランク値を示す．

ステージ 2　線形リストの組を再帰的に併合しながら各分割節点の最大のランク値を求める並列アルゴリズム

ステージ 2 では，ステージ 1 で生成された線形リストに対して，まず 2 つのリストを一対として，各対においてリストを併合して 1 つの組にする．次に 2 つの組を一対としてリストの併合を行う．そして再帰的にリストを併合しながら，最後にリストが 1 つの組になるまで併合を繰り返す．このとき併合のたびに分割節点のランク値を修正しながら，最後に各分割節点の最大のランク値を求める並列アルゴリズムである．

はじめに，アルゴリズムを記述するために必要な用語の定義を行う．併合する 2 つのリストの組にまたがる分割節点の中の兄弟節点を**併合点**と定義する．またリスト間を連結する併合点のみからなるリストを**併合点リスト**と呼ぶ．併合点リストはランクの増分値を線形リスト間に対数時間でブロードキャストするために必要なリストである．併合点リストは上向 1 方向連結リストと下向 1 方向連結リストの 2 種類からなる．以後，上向 1 方向連結リストを**上向併合点リスト**および下向 1 方向連結リストを**下向併合点リスト**と呼ぶ．上向併合点リストとは，併合点となる兄弟節点を**上向辺**（併合点となる兄弟節点をリスト番号の大きい方から小さい方へ結んだ有向辺）で連結し，線形リストを介して併合点を上向辺のみによって 1 方向に連結したリストである．すなわち上向併合点リストはリスト番号の大きい方から小さい方への上向辺と線形リストを介して，できるだけ線形リスト間を 1 段ずつ階段を上るようにつなぎ，併合点におけるランクの増分値の累計和を伝播するリストである．下向併合点リストは，上向併合点リストと同様に構成するが，**下向辺**（併合点となる兄弟節点をリスト番号の小さい方から大きい方へ結んだ有向辺）によって連結したものである．併合点となった兄弟節点は配列 MP[1..S[n]] によって示す．また各節点には上向併合点リストを構成するポインタ ULINK[1..S[n]] と増分値の累計和を計算する配列 UINC[1..[S[n]] および下向併合点リストを構成するポインタ DLINK[1..S[n]] と増分値の累計和を計算する配列 DINC[1..S[n]] を用いる．

ステージ 2 のアルゴリズムは新節点に 1 台のプロセッサを割り付けると仮定して，以下に示す．

ステージ 2　線形リストの組を再帰的に併合しながら各分割節点の最大のランク値を求めるアルゴリズム

```
procedure Linked_List_Rank(INC[1..S[n]],RANK[1..S[n]]);
begin
   for u := 1 to S[n] in parallel do begin
      RANK[u] ← RANK[u] + INC[u];
      Q[u] ← NEXT[u]
   end;
   for i := 1 to log n do
      for u := 1 to S[n] in parallel do
         if Q[u] ≠ 0 then begin
            if RANK[u]+2^i > RANK[Q[u]]
               then RANK[Q[u]] ← RANK[u] + 2^i;
            Q[u] ← Q[Q[u]]
         end
end;
```

図 5　前処理 (2) のアルゴリズム
Fig. 5　Preprocessing (2) algorithm.

前処理

(1) 各分割節点において，兄弟節点のランクの最大値を求め，各兄弟節点の増分値を求めるアルゴリズム
  (1.1) 各分割節点 (V=1,⋯,n) において，兄弟節点のランク値の中で最大のランク値 RMAX[1..n] を並列平衡 2 分木法[5] を用いて求める．
  (1.2) ランクの最大値 RMAX と各兄弟節点のランク値との差すなわち各兄弟節点の増分値 INC[1..S[n]] を求める．

(2) 各線形リストにおいて，各節点の増分値を用いて各節点のランク値を修正するアルゴリズム
  (2.1) 各節点 u において，ランク値 RANK と増分値 INC との和を新たなランク値 RANK[u] とする（RANK[u] ← RANK[u] + INC[u]）．
  (2.2) 各節点のランク値は前節点のランク値より少なくとも 1 大きくなるようにランク値 RANK を修正する．

  この前処理 (2) のアルゴリズムの詳細を擬似コードによって**図 5** に示す．

(3) 再度 (1) を実行する．

以下のアルゴリズムでは，ステージ 1 で生成された線形リストの構造には変更はないが，線形リストを併合するとき仮のリスト番号を付けて処理するので，リスト番号はいくつかの線形リストの組すなわちリストの集合をも表す．

loop ← 0

以下のステップを log（線形リストの総数 $= L$）回繰り返す

loop ← loop+1

ステップ［2.1］　まずリスト番号が奇数と偶数となるリストの組を作り，各リストの組において併合点を求める．次に各併合点から線形リスト上をなぞり，次

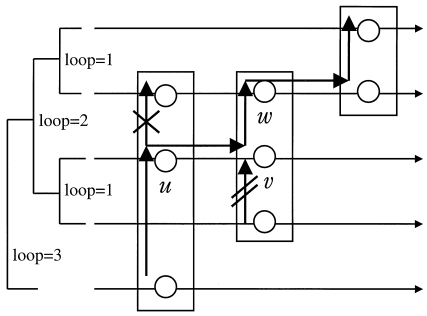


図 6 上向併合点リストの例
Fig. 6 Upward_mergepoints list.

の併合点を求めて，併合点リストを構成する．ステップ［2.1］のアルゴリズムの詳細を下記に示す．

(1) リスト番号の奇数と偶数すなわち $(i, i+1), (i = 2j-1, j = 1, 2, \cdots L/2^{loop})$ の組を作り，各リストの組において併合点を求める．併合点は隣接する兄弟節点のリスト番号が奇数 (i)，偶数 $(i+1)$ の連続した番号を持つ兄弟節点である（ただし最大のランク値が 1 の併合点は以後ランク値が変化しないので対象としない）．

(2) 新しく併合点となる兄弟節点は配列 MP で印を付け，上向辺のポインタ ULINK と下向辺のポインタ DLINK でそれぞれにリンクする．

(3) 上向併合点リストの構成

(3.1) 各上向辺の入節点 u から線形リスト上をなぞり，次の併合点（隣接併合点 v）を探す（**図 6** において loop=3 の併合時を参照）．

(3.2) 隣接併合点 v が上向辺の出節点であれば，併合点 u と隣接併合点 v を上向併合点リストとして連結する．このとき併合点 u が上向辺を持っていれば，上向辺は自動的に削除される（図 6 の×印）．また隣接併合点 v が上向辺の入節点になっている場合には，出節点のみにする．すなわち隣接併合点 v への兄弟節点からの上向辺を削除する（図 6 の//印）．

隣接併合点 v が上向辺の出節点でないときは，併合点 u と隣接併合点 v とはリンクしない．ただし併合点 u が既存の隣接併合点とのリンクを持っていれば，それを削除する．

(4) 下向併合点リストの構成についても，上向併合点リストの構成アルゴリズム (3) と同様に行う．

ステップ［2.2］ 併合点リストを用いて各線形リスト間に併合点の増分値の累計和を受け渡し，各節点の増分値を求める．ステップ［2.2］のアルゴリズムの詳細を下記に示す．

```
procedure Merge_List_Inc(UINC[1..S[n]],DINC[1..S[n]],
   INC[1..S[n]],Q[1..S[n]]);
{ Q は併合点リスト上の次併合点を示す }
begin
   { (1) 上向併合点リストにおける増分値 UINC の計算 }
   for u := 1 to S[n]-1 in parallel do begin
      UINC[u] ← 0; DINC[u] ← 0;
      if (MP[u] = 1) and (MP[u+1] = 1) and (ULINK[u+1] = u) then
         UINC[u] ← INC[u] - INC[u+1]
   end;
   repeat log m times
      for u := 1 to S[n] in parallel do
         if Q[u] ≠ 0 then begin
            UINC[Q[u]] ← UINC[Q[u]] + UINC[u];
            Q[u] ← Q[Q[u]]
         end;
   { (2) 下向併合点リストにおける増分値 DINC の計算 }
   下向併合点リストの増分値の計算に関しては上向併合点リストと同様に実行
   { (3) 併合点の増分値の修正 }
   for u := 1 to S[n] in parallel do
      INC[u] ← Max(UINC[u],DINC[u],INC[u])
end;
```

図 7 ステップ［2.2］のアルゴリズム
Fig. 7 Step [2.2] algorithm.

(1) 上向併合点リストにおける増分値 UINC の計算（以下は図 6 を参照）．

(1.1) 上向辺を構成する兄弟節点 $(v \rightarrow w)$ 間では，出節点 v の真の増分値を入節点 w に伝播するために，入節点 w の増分値 UINC には，入節点 w の増分値 INC の値から出節点 v の増分値 INC の値を減算した値を代入する（UINC[w] ← INC[w] − INC[v]）．

(1.2) 増分値 UINC を線形リスト間に伝播するために，各上向併合点リストを用いて増分値 UINC の累計和をブロードキャストする．

(2) 下向併合点リストにおける増分値 DINC の計算も，上向併合点リストに対する上記のアルゴリズム (1) と同様に増分値 DINC を計算する．

(3) 各併合点の増分値 INC は上向併合点リストの増分値 UINC と下向併合点リストの増分値 DINC および元の増分値 INC の 3 つの値の中で最大の値を用いて修正する．

ステップ［2.2］のアルゴリズムの詳細を擬似コードによって**図 7** に示す．

ステップ［2.3］ 各線形リストにおいて，各節点の増分値 INC を用いて各節点のランク値 RANK を修正する．すなわち前処理 (2) と同じアルゴリズムを実行する．

ステップ［2.4］ 各分割節点において，兄弟節点のランクの最大値 RMAX を求め，各兄弟節点の増分値 INC を求める．すなわち前処理 (1) と同じアルゴリズムを実行する．

```
procedure Stage-2(linked_list); { ステージ 2 の並列アルゴリズム }
begin
  { 前処理 (1) }
  for v := 1 to n in parallel do { 兄弟節点のランク値 RANK の最大値 RMAX を求める }
    for i ∈ SIB[v] in parallel do
      RMAX[v] ← Max(RANK[i]); { 並列平衡 2 分木法 }
  for u := 1 to S[n] in parallel do { 増分値 INC を求める }
    INC[u] ← RMAX[N[u]] - RANK[u];
  { 前処理 (2) (図 5 参照) }
  Linked_List_Rank(INC[1..S[n]],RANK[1..S[n]]);
  { 前処理 (3) } 前処理 (1) と同じアルゴリズムを実行
  { merging }
  loop ← 0;
  repeat log L times { L:初期線形リストの総数 }
    loop ← loop + 1;
    { ステップ [2.1] }
    { (1) リスト番号の奇数と偶数の組を作り上向辺と下向辺の構成 }
    for u := 1 to S[n]-1 in parallel do { 併合点を探す }
      if (N[u] = N[u+1]) and (LISTN[u]+1 = LISTN[u+1]) and
      (LISTN[u] mod 2 = 1) then begin
        MP[u] ← 1; MP[u+1] ← 1; { 併合点 }
        ULINK[u+1] ← u; DLINK[u] ← u+1 { (2) 上向辺および下向辺の構成 }
      end;
    { (3) 上向併合点リストの構成 }
    for u := 1 to S[n] in parallel do begin
      P[u] ← NEXT[u]; { 補助的な配列 P を用いる }
      while (P[u] ≠ 0) and (MP[P[u]] ≠ 1) do
        P[u] ← P[P[u]]; { 線形リスト上で隣接併合点を探す }
      if (u < S[n]) and (P[u] ≠ 0) and (MP[u] = 1) and (MP[P[u]] = 1) then
        if (ULINK[u+1] = u) and (ULINK[P[u]] = P[u]-1) then
          if ULINK[u] ≠ P[u] then begin
            ULINK[u] ← P[u];
            if N[P[u]+1] = N[ULINK[P[u]+1]] then
              ULINK[P[u]+1] ← 0 { 兄弟節点のリンクを削除 }
          end
        else if ULINK[u] ≠ u-1 then { 既存の隣接併合点へのリンクがあれば削除 }
          ULINK[u] ← 0
    end;
    { (4) 下向併合点リストの構成 }
    下向併合点リストも上向併合点リストと同様に構成
    { ステップ [2.2] 併合点リスト上での増分値の処理 (図 7 参照) }
    Merge_List_Inc(UINC[1..S[n]],DINC[1..S[n]],INC[1..S[n]],ULINK[1..S[n]]);
    { ステップ [2.3] } 前処理 (2) と同じアルゴリズムを実行
    { ステップ [2.4] } 前処理 (1) と同じアルゴリズムを実行
    { ステップ [2.5] リスト番号の更新 }
    for u := 1 to S[n] in parallel do
      if (LISTN[u] mod 2) = 1
        then LISTN[u] ← (LISTN[u]+1)/2
        else  LISTN[u] ← LISTN[u]/2
end;
```

図 8 ステージ 2 のアルゴリズム
Fig. 8 Parallel algorithm for Stage-2.

ステップ［2.5］ 各線形リストの組のリスト番号 $(i, i+1), (i = 2j-1, j = 1, 2, \cdots L/2^{loop})$ を $\lceil (i+1)/2 \rceil$ に更新し，ステップ［2.1］に帰る．

ステージ 2 の全体のアルゴリズムの概要を擬似コードによって**図 8** に示す．

ステージ 2 のアルゴリズムの正当性と計算量に関する考察

前処理 (1) における各分割節点のランクの最大値は新節点に 1 台のプロセッサを割り付けて，既知の並列平衡 2 分木法を用いて求めることができる．任意の分割節点における兄弟節点数はたかだか $(m-1)$ 個なので，計算時間は $O(\log m)$ となる．兄弟節点の増分値の計算は同時読み込みが可能（CR）であるから定数時間 $O(1)$ でできる．前処理 (2) は，図 5 に示すアルゴリズムによって各線形リスト上のランク値の修正が正しくできる．この前処理 (2) は線形リスト上の節点数がたかだか $(n-1)$ 個なのでダブリング法によって $O(\log n)$ 時間でできる．前処理 (3) は前処理 (1) の再計算であるから，計算量は前処理 (1) と同様である．このときプロセッサ数はいずれの場合も $O(n+m)$ である．したがって前処理の計算量はプロセッサ数が $O(n+m)$ で計算時間は $O(\log m)$ となる．

ステージ 2 の繰返し部分においても，新節点に 1 台のプロセッサを割り付ける．ステップ［2.1］においては，併合点を求めるには，各分割節点における兄弟節点はリスト番号順に整列されているので，リスト番号が奇数 (i)，偶数 $(i+1)$ と連続した兄弟節点を併合点とすればよい．次に併合点リストを構成するアルゴリズムでは，線形リスト上をなぞって隣接併合点を探し，併合点リストとして条件に合うかどうかを簡単に検証するだけであるから，その正当性は明らかである．このステップの計算量については，まず併合点を求める処理および上向辺と下向辺をリンクする処理は定数時間 $O(1)$ でできる．また併合点から隣接併合点を探すには，線形リスト上をダブリング法でなぞって求めることができるので，計算時間は $O(\log n)$ となる．併合点リストを構成するための処理は隣接併合点の兄弟節点間の関係を見れば分かるので定数時間 $O(1)$ でできる．プロセッサ数は新節点に 1 台のプロセッサを割り付けるので，いずれの場合も $O(n+m)$ である．したがって，このステップの計算量はプロセッサ数が $O(n+m)$ で計算時間は $O(\log n)$ となる．

ステップ［2.2］においては，各併合点の増分値を併合点リストを用いて各線形リスト間に伝播するには，各線形リストの増分値の累計和を受け渡す必要がある．下記に上向併合点リストを用いて，各線形リストの増分値を伝播するアルゴリズムを具体的に考察する（以下は図 6 および図 7 を参照）．

線形リスト間をまたぐ上向辺 $(v \to w)$ は，出節点 v の真の増分値を入節点 w へ伝達する役割を持つもので，入節点 w に伝達される増分値 UINC[w] は UINC[w] ← INC[w] − INC[v] とする．なぜなら出節点 v と入節点 w は兄弟節点であり，それらの増分値 INC[v] と INC[w] には，それらの兄弟節点が属する分割節点の最大ランク値に基づく増分値が格納されている．したがって入節点 w に伝達される増分値 UINC[w] は

INC[w] と INC[v] の増分値の差を格納する必要がある．その結果，上向併合点リストは併合点の増分値 UINC の累計和として真の増分値を線形リスト間に伝播することができる．下向併合点リストについても同様なアルゴリズムで真の増分値を線形リスト間に伝播することができる．そして各節点の増分値 INC は 3 つの増分値 UINC，DINC および INC の値の中で最大の値とする．

ステップ［2.2］の計算量は，増分値の累計和は並列プレフィックス和アルゴリズムより求めることができるので，$O(\log m)$ 時間であり，上向辺（下向辺）を構成する併合点の増分値の調整と各節点の 3 つの増分値の最大値を求める処理は定数時間 $O(1)$ である．プロセッサ数は新節点に 1 台のプロセッサを割り付けるので，いずれの場合も $O(n+m)$ である．したがって，このステップの計算量はプロセッサ数 $O(n+m)$ で計算時間は $O(\log m)$ となる．

ステップ［2.3］の線形リスト上のランク値の修正は前処理 (2) のアルゴリズムと同様であり，計算量はプロセッサ数 $O(n+m)$ で $O(\log n)$ 時間である．

ステップ［2.4］の各分割節点における兄弟節点の増分値の算定は前処理 (1) のアルゴリズムと同様であり，計算量はプロセッサ数 $O(n+m)$ で $O(\log m)$ 時間である．

ステップ［2.5］のリスト番号の更新はプロセッサ数は $O(n+m)$ で定数時間 $O(1)$ である．

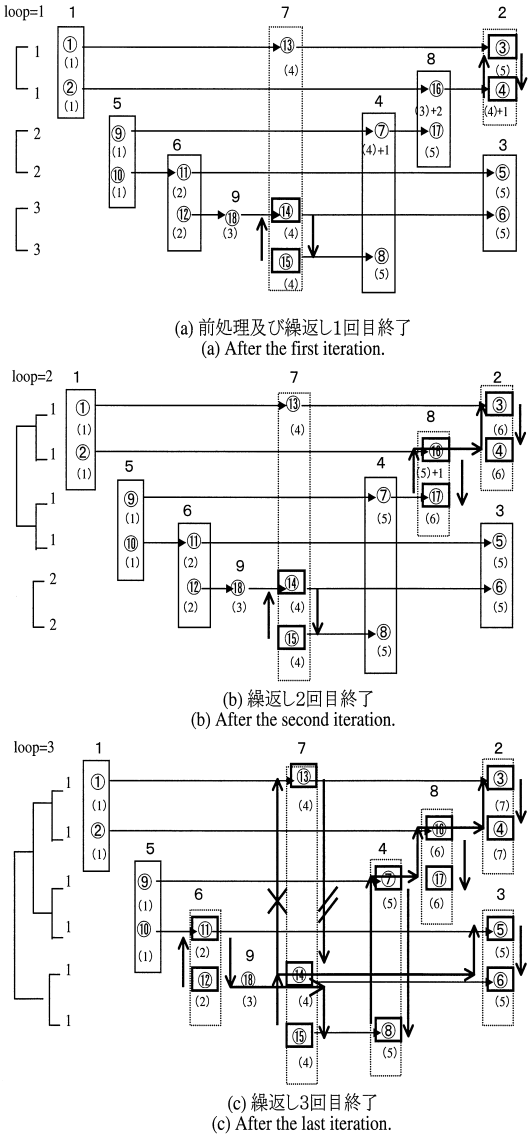
最終的には，ステージ 2 の繰返し回数はたかだか $O(\log m)$ 回であるので，ステージ 2 の処理に要する計算量はプロセッサ数が $O(n+m)$ で計算時間は $O(\log^2 m)$ となる．

［具体例］

具体例におけるステージ 2 の処理を**図 9** に示す．図 9 において，( ) のランク値に続く符号付数値は増分値 INC を，太枠で囲った節点は併合点を，太線の矢印は併合点リストを表す．

繰返しの 1 回目図 9 (a) では，分割節点 **2** の兄弟節点③と④および分割節点 **7** の兄弟節点⑭と⑮が併合点となり，上向辺と下向辺でリンクされる．1 回目の処理では増分値の修正に対して併合点リストからの影響はない．

繰返しの 2 回目図 9 (b) では，新たに分割節点 **8** の兄弟節点⑯と⑰が併合点となり，上向辺と下向辺でリンクされる．このとき節点⑯の併合点から隣接併合点を探すために線形リスト上をなぞると，併合点④は上向辺の出節点であるから上向併合点リストとして併合点⑯と④をリンクする．併合点リストの増分値 UINC



(a) 前処理及び繰返し1回目終了
(a) After the first iteration.

(b) 繰返し2回目終了
(b) After the second iteration.

(c) 繰返し3回目終了
(c) After the last iteration.

図 9 ステージ 2 の処理
Fig. 9 Process of Stage-2.

の値は図 9 (a) において INC[16]=+2，INC[4]=+1 および INC[3]=0 なので，UINC[16]=+2，UINC[4]=0 および UINC[3]=0−1=−1 となる．そして，その上向併合点リスト上で増分値 UINC の累計和をブロードキャストすると，増分値 UINC の値は UINC[16]=+2，UINC[4]=+2 および UINC[3]=+1 となり，節点⑯のランク値は (5)，節点④と③のランク値はそれぞれ (6) となる（図 9 (b)）．

最後に，繰返しの 3 回目図 9 (c) では，分割節点 **6** の兄弟節点⑪と⑫，分割節点 **7** の兄弟節点⑬と⑭，分割節点 **4** の兄弟節点⑦と⑧および分割節点 **3** の兄弟

節点⑤と⑥が併合点となり，上向辺と下向辺でリンクされる．上向辺および下向辺の入節点から線形リスト上をなぞって隣接併合点を探し，存在すればその隣接併合点とリンクする．その結果，上向併合点リストとして，まず併合点⑦は隣接併合点⑰と新たにリンクされ，⑧→⑦→⑰→⑯→④→③の上向併合点リストが生成される．さらに併合点⑭は上向併合点リストとして隣接併合点⑥とリンクされるので，上向辺（⑭→⑬）は削除され（×印），上向併合点リスト⑮→⑭→⑥→⑤が生成される．一方下向併合点リストとしては，下向辺（⑪→⑫）の併合点⑫は下向併合点リストとして隣接併合点⑭とリンクされるので，下向辺（⑬→⑭）は削除され（//印），下向併合点リスト⑪→⑫→⑭→⑮が生成される．このとき，節点の増分値は図 9 (b) に示すように，節点⑯のみに INC[16]=+1 が存在するので，この節点が含まれる上向併合点リスト⑧→⑦→⑰→⑯→④→③によってのみ増分値が修正される．すなわち，この上向併合点リストの増分値 UINC の値は併合点⑯のみが UINC[16]=+1 で，併合点⑯以外の節点の UINC は 0 となる．この上向併合点リスト上で増分値 UINC の累計和をブロードキャストすると，増分値 UINC の値は併合点⑯までの UINC の値は 0 で，UINC[16]=+1，UINC[4]=+1 および UINC[3]=+1 となる．その結果，節点⑯のランク値は (6)，節点④と③のランク値はそれぞれ (7) となり，各分割節点の最大ランク値が求まる（図 9 (c)）．

ステージ 3　各分割節点の最大ランク値をもとにトポロジカル整列を行う並列アルゴリズム

ステージ 3 では各分割節点の最大ランク値をもとに節点のトポロジカル整列を行う．具体的には，ステージ 2 で求めた分割節点の最大ランク値 RMAX[1..n] に基づいて，節点の最大ランク値を昇順に整列し，所与の有向辺で連結すれば，すべての有向辺が左から右へ向かう節点のトポロジカル整列が求められる．

ステージ 3 のアルゴリズムの正当性と計算量に関する考察

節点のトポロジカル整列の正当性は自明であり，並列整列アルゴリズムを用いてプロセッサ数 $O(n)$，計算時間が $O(\log n)$ で求めることができる．

[具体例]

具体例からステージ 3 のトポロジカル整列の結果を表示すれば**図 10** のようになる．

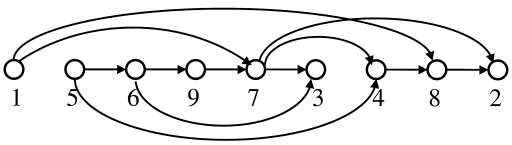


図 10　トポロジカル整列の結果
Fig. 10　Result of topological sorting.

4. ま と め

無閉路有向グラフ DAG において，基本的な並列アルゴリズムのみを用いてトポロジカル整列を行う並列アルゴリズムを示した．このアルゴリズムは CREW-PRAM 計算機モデルで，プロセッサ数が $O(n+m)$，計算時間が $O(\log^2 m)$ で実行できる．

はじめに，従来の並列アルゴリズムと提案した並列アルゴリズムとを同じ CREW-PRAM 計算機モデルのもとで比較検討する．まず計算時間に関しては，従来のアルゴリズムは CRCW-PRAM 計算機モデルのもとで $O(\log n)$ であるので，CREW-PRAM 計算機モデルにおいては $O(\log^2 n)$ であり，提案するアルゴリズムでは $O(\log^2 m)$ であるから，計算時間は漸近的に同等である．一方，使用するプロセッサについては，従来のアルゴリズムでは隣接行列から推移的閉包行列を用いて計算しているので，$O\left(n^3\right)$ または $O\left(n^{2.376}\right)$ 個のプロセッサ数が必要である．しかし提案した並列アルゴリズムでは，DAG が密になってもプロセッサ数はたかだか $O\left(n^2\right)$ であり，疎のとき $(O(m) = O(n))$ にはプロセッサ数は $O(n)$ である．したがって，提案した並列アルゴリズムはプロセッサ数を減少できるので，従来の並列アルゴリズムよりコストが削減でき，より効率の良い並列アルゴリズムである．

次に，トポロジカル整列の逐次（直列）アルゴリズムの時間計算量は $\Theta(n+m)$ であるが，提案した並列トポロジカル整列のコストは $O((n+m)\log^2 m)$ であるから，まだ最適アルゴリズムではない．しかし Brent の定理[5] を適用して最適化することは容易である．すなわちデータ構造として配列を用いているので，$\log^2 m$ 個の配列の要素を 1 台のプロセッサに受け持たせることによって可能である．

以上の考察から，提案した並列アルゴリズムは DAG の疎密に対応してプロセッサ数を減少できるので，DAG から効率良くトポロジカル整列を求めることができる．提案した並列トポロジカル整列アルゴリズムは生産工程や PERT などの半順序関係を DAG で表現できれば，そのトポロジカル整列を並列に効率良く実行することができる．

謝辞　有益なコメントをいただきました査読者の方に感謝いたします．

参 考 文 献

1) Knuth, D.E.: *Fundamental Algorithms*, *The Art of Computer Programming*, 3rd ed., Vol.1, pp.261–268, Addison-Wesley (1997).
2) Er, M.C.: A Parallel Computation Approach to Topological Sorting, *The Computer J.*, Vol.26, No4, pp.293–295 (1983).
3) Chandhui, P. and Ghosh, R.K.: Parallel Algorithms for Analyzing Activity Networks, *BIT 26*, pp.418–429 (1986).
4) Coppersmith, D. and Winograd, S.: Matrix Multiplication via Arithmetic Progressions, *Proc. 19th ACM Symp. on Theory of Computing*, pp.1–6 (1987).
5) Gibbons, A. and Rytter, W.: *Efficient Parallel Algorithms*, pp.6–18, Cambridge University Press (1988).
6) Xavier, C. and Iyegar, S.S.: *Introduction to Parallel Algorithms*, pp.108–140, Wiley-Inter Science (1998).
7) Cole, R.: Parallel Merge Sort, *SIAM J. Comput.*, Vol.17, No.4, pp.770–785 (1988).

(平成 15 年 6 月 6 日受付)
(平成 16 年 2 月 2 日採録)

多田 昭雄（正会員）

1966 年京都大学工学部数理工学科卒業．同年東京芝浦電気（株）（現（株）東芝）入社．1994 年熊本工業大学（現崇城大学）講師．現在同大学工学部応用電気情報工学科助教授．熊本大学大学院自然科学研究科後期博士課程在学中．アルゴリズムとデータ構造の設計と解析等に関心を持つ．電子情報通信学会会員．

右田 雅裕（正会員）

1994 年熊本大学工学部電気情報工学科卒業．1996 年同大学院工学研究科電気情報工学専攻修士課程修了．2000 年同大学院自然科学研究科後期博士課程単位取得退学．現在熊本大学総合情報基盤センター助手．アルゴリズムとデータ構造の設計と解析等に関心を持つ．

中村 良三（正会員）

1968 年熊本大学大学院修士課程修了．工学博士．1968 年〜1974 年中部電力（株）．1975 年熊本大学工学部．現在同大学工学部数理情報システム工学科教授．アルゴリズムとデータ構造の設計と解析等に関心を持つ．電子情報通信学会会員．