PERTチャートにおけるクリティカルパスを求める並列アルゴリズム

右田雅裕[†1] 多田昭雄[†2]
糸川剛[†3,☆] 中村良三[†4]

PERT（Program Evaluation and Review Technique）解析の1つに，PERTチャートにおけるクリティカルパス（critical path）を求める問題があるが，そのクリティカルパスを求める効率良い並列アルゴリズムは見当たらない．本稿では，すでに提案した効率良い並列トポロジカル整列アルゴリズムをもとに，PERTチャートをDAG（Directed Acyclic Graph）の節点に重み付けして表し，CREW-PRAM計算機モデルのもとでPERTチャートにおけるクリティカルパスを求める並列アルゴリズムを提案する．具体的には，まずPERTチャートを入出次数がたかだか1の線形リストに分割し，これを併合しながら各節点の最長所要時間を求める．次にその逆向線形リストを構成し，同様に各節点の最長所要時間を求める．これら2つの最長所要時間を用いて各節点の最早開始時刻（earliest start time），最遅開始時刻（latest start time）および余裕時間（floating time）を求める．最後に節点と経路の判定を定数時間で行い，PERTチャートにおけるすべてのクリティカルパスを求める並列アルゴリズムである．PERTチャートにおいて節点数 n，辺数 m とするとき，提案する並列アルゴリズムの計算量はCREW-PRAMモデルで，プロセッサ数が $O(n+m)$，時間量が $O(\log^2 m)$ である．

Parallel Algorithm for Determining Critical Paths in PERT Chart

Masahiro Migita,[†1] Akio Tada,[†2] Tsuyoshi Itokawa[†3,☆] and Ryozo Nakamura[†4]

In the analyses on PERT (Program Evaluation and Review Technique), the problem to determine the critical paths in a PERT chart is well-known. However, any parallel algorithms to determine the critical paths in a PERT chart have not been presented. In this paper, we propose an efficient parallel algorithm for determining the critical paths in a PERT chart on a CREW-PRAM model, which is based on the efficient parallel topological sorting algorithm in a DAG (Directed Acyclic Graph, n:vertices, m:edges). In the proposed algorithm, at first we divide the given PERT chart into the several linked lists, and calculate the longest time for each vertex on the lists. Next, we calculate the earliest start time, the latest start time and the floating time for each vertex in PERT. Finally, the critical paths in the given PERT chart are determined in $O(1)$ time. On a PERT chart with the number of vertices n and the number of edges m, the proposed parallel algorithm requires $O(n+m)$ processors and $O(\log^2 m)$ times on a CREW-PRAM model.

†1 熊本大学総合情報基盤センター
Center for Multimedia and Information Technologies, Kumamoto University
†2 崇城大学情報学部コンピュータシステムテクノロジー学科
Department of Computer System Technology, Faculty of Computer and Information Sciences, Sojo University
†3 熊本大学大学院自然科学研究科システム情報科学専攻
Department of Systems and Information, Graduate School of Science and Technology, Kumamoto University
†4 熊本大学工学部数理情報システム工学科
Department of Computer Science, Faculty of Engineering, Kumamoto University
☆ 現在，熊本大学大学院自然科学研究科情報電気電子工学専攻
Presently with Computer Science and Electrical Engineering, Graduate School of Science and Technology, Kumamoto University

1. はじめに

PERT（Program Evaluation and Review Technique）は生産工程やプロジェクトの日程または所要時間を管理する技法の1つとして知られている[1)]．PERTではPERTチャート（PERT chart）によってジョブの作業手順をアローダイアグラム（arrow diagram）で表す．PERTチャートの節点や有向辺にジョブやそ

のジョブの所要時間を割り当て，各ジョブの順序関係を図示する．この PERT チャートにおける重要な解析の 1 つにクリティカルパス（critical path）を求める問題がある．クリティカルパスはプロジェクト全体のスケジュールを決定するジョブの順序列であり，この経路上の各ジョブの処理に要する合計時間はプロジェクト全体の完了に要する必要不可欠な時間を表す．そのため，クリティカルパス上のジョブの処理に遅延が発生した場合，プロジェクト全体の完了にも遅延が生じることになる．したがって，プロジェクトの管理においてこのクリティカルパスを求めることは非常に重要である．PERT チャートにおいて節点数 n，辺数 m とするとき，そのクリティカルパスを求める逐次アルゴリズムの時間量は $O(n+m)$ である．すなわちその時間量は節点数および辺数の線形時間となる．その節点数および辺数が非常に大きくなったときに，クリティカルパスを高速に求めるには効率良い並列アルゴリズムを設計する必要がある．しかしながら，PERT チャートにおけるクリティカルパスを求める効率良い並列アルゴリズムは見当たらない．

本稿では，CREW-PRAM 計算機モデルのもとで，PERT チャートにおけるクリティカルパスを求める効率良い並列アルゴリズムを提案する．仮想的な並列計算機モデルである CREW-PRAM モデルは共有メモリの同時読み出し（concurrent read）と排他的書き込み（exclusive write）を行う PRAM の一種である[6]．提案する並列アルゴリズムは，効率良くトポロジカル整列を行う並列アルゴリズム[2] の設計法に準じ，分割統治法と基本的な並列アルゴリズム[3),4)] によって設計される．提案するアルゴリズムでは，PERT チャートをグラフとして簡潔に記述するために DAG（Directed Acyclic Graph，節点数 n，辺数 m）を用いて表し，DAG の各節点には固有のジョブとそのジョブの所要時間を付与する．与えられた PERT チャート（DAG）は，隣接行列や隣接リスト表現ではなく，各有向辺を出節点と入節点の組で表した配列で表示される．はじめにアルゴリズムのステージ 1 として，CREW-PRAM モデルにおける書き込みの競合を回避するため，入力された PERT チャート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．次にステージ 2 では，分割された線形リストの組を再帰的に併合しながら，各節点における最長所要時間を求める．最後のステージ 3 では，まず，ステージ 1 で構成した線形リストに対して，すべての有向辺の向きを逆転させたリストである逆向線形リストを構成し，これに対してステージ 2 と同様に最長所要時間を求める．さらに，これら 2 つの最長所要時間より，各節点における最早開始時刻（earliest start time）と最遅開始時刻（latest start time）を求める．次に，最早開始時刻と最遅開始時刻より各節点における余裕時間（floating time）を求める．そして，各節点におけるこれらの時間を用いて，クリティカルパスを構成する節点と経路の判定を定数時間で行い，PERT チャートにおけるすべてのクリティカルパスを求める並列アルゴリズムである．

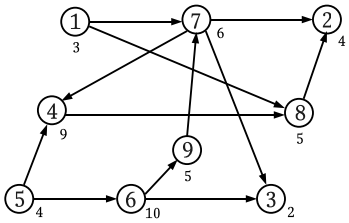
提案するアルゴリズムの計算量は，プロセッサ数 $O(n+m)$ を用いて時間量が $O(\log^2 m)$ となり，プロセッサ数が PERT チャートの節点数と有向辺数によって決まり，グラフの疎密に対応できる効率良い並列アルゴリズムである．並列アルゴリズムの性能は，使用するプロセッサ数と時間計算量との積として定義されるコストによって評価される．したがって，Brent の定理[3] を用いると提案する並列アルゴリズムのコストはクリティカルパスを求める逐次アルゴリズムの時間量と同等となり，効率良い並列アルゴリズムである．また，データ構造を配列により構成しているので，プロセッサ数が少ない場合でもプロセッサ数に応じた分割を行うことが可能で負荷分散ができる．

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計算量を考察する．

2. クリティカルパスを求める並列アルゴリズムの概要

ジョブの作業手順をアローダイアグラムで表した PERT チャートにおいては，1 から通し番号が付けられた節点はジョブを表し，その節点の重みはそのジョブの所要時間を表す．PERT チャートでは一般的に閉路や多重辺を含まない．したがって，PERT チャートを簡潔に表現するため DAG（節点の集合 V，有向辺の集合 E，|V|=n，|E|=m）を用いる．所与の PERT チャートの有向辺は配列の形で表し，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，また，配列 T は各節点の重みすなわちジョブの所要時間を示す．ただし，n≤m で，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．**図 1** (a) に PERT チャートの例を示し，図 1 (b) にその有向辺と節点の重みを表す配列を示す．ただし，図 1 (a) において，節点内の数字は節点番号を表し，また，節点の近傍の数字はその重みを表す．

ここで，アルゴリズムを記述するために必要な用語



(a) PERT チャートの例

	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

	1	2	3	4	5	6	7	8	9
T	3	4	2	9	4	10	6	5	5

(b) PERT チャートの有向辺と節点の重みを表す配列

図 1 所与の PERT チャート
Fig. 1 A given PERT chart.

の定義を行う．各ジョブ（節点）における**最長所要時間**は，そのジョブとそれに先行するすべてのジョブを完了するのに要する最長の所要時間を表す．また，**最早開始時刻**はあるジョブ（節点）に先行する経路上のすべてのジョブを完了してそのジョブを開始できる最も早い時刻を表し，**最遅開始時刻**はあるジョブ（節点）に後続するすべてのジョブを完了してプロジェクト全体の完了時刻に間に合うようにそのジョブを開始できる最も遅い時刻を表す．また，**余裕時間**はプロジェクト全体の完了時刻に間に合うようにあるジョブを開始できる時間的な余裕を表す．すなわち，各ジョブにおける余裕時間は，そのジョブの最遅開始時刻と最早開始時刻の差である．そして，PERT チャートにおけるクリティカルパスでは，PERT における性質からその経路に含まれる各ジョブの余裕時間が 0 となる．

次に，提案するアルゴリズムの概要を示す．提案するクリティカルパスを求める並列アルゴリズムは，次の 3 つのステージから構成される．

ステージ 1　PERT チャートを線形リストに分割する並列アルゴリズム
ステージ 2　各節点（ジョブ）の最長所要時間を求める並列アルゴリズム
ステージ 3　クリティカルパスを求める並列アルゴリズム

まずステージ 1 では，所与の PERT チャートである DAG において節点の入出次数がたかだか 1 となるように節点の分割を行い，有向グラフを線形リストに分割する．次にステージ 2 では，分割された線形リストの組を再帰的に併合しながら，各節点すなわち各ジョブにおいて最長所要時間を求める．このとき，線形リストの組を併合する際には，線形リスト間を連結する併合点リストを構成し，最長所要時間の計算に用いる．ステージ 3 は，次の 2 つの部分で構成される．まず，線形リストにおける各有向辺の向きを逆転させた逆向線形リストを構成し，これに対してステージ 2 を適用して逆向線形リストにおける各節点の最長所要時間を求める．次に，線形リストにおいて求めた最長所要時間より PERT における各ジョブ（節点）の最早開始時刻を求め，逆向線形リストにおける最長所要時間を用いて各ジョブ（節点）の最遅開始時刻を求める．さらに，この 2 つの時刻から各ジョブ（節点）の余裕時間を求める．そして，各節点におけるこれらの時間を用いてクリティカルパスを構成する節点と経路の判定を行い，PERT チャートにおけるクリティカルパスの構成を定数時間で行う．

提案する並列アルゴリズムのステージ 1 と 2 は，基本的にそれぞれ参考文献 2) におけるステージ 1 と 2 に準ずる．

3. 提案するアルゴリズムの詳細化

提案するアルゴリズムのステージ 1，2，3 をそれぞれ 3.1〜3.3 節に分割して詳細化する．

3.1 PERT チャートを線形リストに分割する並列アルゴリズム（ステージ 1）

PERT チャートである DAG において，有向辺を表す配列に基づき，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 PERT チャート（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．詳細を下記に示す．

(1) 各節点の出次数 O[1..n] を求める．
  (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] に格納する．
  (1.2) 配列 B において 1 つ前の境界辺番号との差から出次数 O を求める．
(2) 各節点の入次数 I[1..n] を求める．
  (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) アルゴリズム (1) と同様にして入次数 I を求める．
(3) 各節点の出次数 O と入次数 I の大きい方の値すなわち最大次数である兄弟節点数 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き，その兄弟節点に連続した通し番号を付ける．このとき所与

の PERT チャートである 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，新節点番号に基づいて所与の PERT チャート（DAG）を線形リストに分割する．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 e，旧節点の集合の要素を v，新節点の集合の要素を 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 PERT チャート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，線形リストの末尾の NEXT は仮想節点番号 0 とする）．

ステップ［1.3］ 各線形リスト上で，各節点における最長所要時間の初期値を求める．次に，各線形リストにリスト番号を付けて，各節点に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．詳細を下記に示す．

(1) 所与の DAG における節点の重みすなわち所与の PERT チャートにおけるジョブの所要時間 T[1..n] を，新節点の重みとして配列 NT[1..S[n]] に格納する．次に，各線形リスト上でこの重みすなわちジョブの所要時間 NT の累計和を求めて，新節点における最長所要時間 PT[1..S[n]] の初期値として格納し，また，この初期値を後で参照するため配列 PPT[1..S[n]] にも保存する．

(2) 各線形リストにリスト番号を付ける．仮のリスト番号は，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全節点にブロードキャストする（節点 u のリスト番号は LISTN[u] で表す）．次に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）．

(4) 各節点のリスト番号 LISTN を配列 OLISTN [1..S[n]] に保存する．

```
procedure Stage-1;
{ 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
  { (5) 線形リストの生成 }
  for u := 1 to S[n] in parallel do { NEXT の初期値設定 }
    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
    NT[u] ← T[N[u]];
  Para_Prefixsum(NT[1..S[n]],PT[1..S[n]]); { 各線形リスト上で新節点の PT を計算 }
  for u := 1 to S[n] in parallel do { PT の初期値を保存 }
    PPT[u] ← PT[u];
  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 I[N[u]] = 0 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 }
  for u := 1 to S[n] in parallel do
    OLISTN[u] ← LISTN[u] { リスト番号を保存 }
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**図 2** に示す．既知の，並列累計和を求める Para_Prefixsum，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 については，詳細な記述は省略し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 である節点は前の節点の辺番号で埋めるアルゴリズムであるため，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．なお配列等の初期値はすべて 0 と仮定する．

```
procedure Broadcast-1(var 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;
   if B[n]=0 then B[n] ← B[n-1]
end;
```

図 3 Broadcast-1 のアルゴリズム
Fig. 3 Algorithm for Broadcast-1.

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，出節点配列 OV は節点番号順に整列されているので，その出次数を算定するアルゴリズムの正当性は明らかである．また入次数は入節点配列 IV を昇順に整列し，出次数と同様の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，プロセッサ数 $O(m)$ で，時間量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，並列整列アルゴリズム[5]（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定数時間 $O(1)$ で求まる．したがって，このステップの計算量は，プロセッサ数が $O(m)$ で，時間量は $O(\log m)$ である．

ステップ［1.2］では，兄弟節点数の累計和 S は Para_Prefixsum（並列累計和アルゴリズム）[3] 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がたかだか $(n+m-1)$ 個であるので，各節点に 1 台のプロセッサを割り当てると，プロセッサ数が $O(n+m)$ で，時間量は入節点配列 IV を昇順に整列する時間 $O(\log m)$ となる．

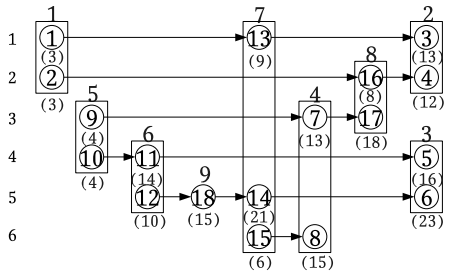
ステップ［1.3］では，まず分割節点の重みを新節点に割り付けて，線形リスト上でこの重みすなわちジョブの所要時間の累計和を求めることで，各節点における最長所要時間 PT の初期値を求めることができる．次に各リストに一連の通し番号を与えるときおよび兄弟節点をリスト番号順に整列するときに並列整列アルゴリズムを用いる．これらは既知の並列アルゴリズムのみを用いて設計できるので，その正当性は明らかである．時間計算量は，線形リストの数および分割節点における兄弟節点数はたかだか $(m-1)$ なので，並列整列アルゴリズムを用いて $O(\log m)$ でできる．また，最長所要時間の初期値算定と各節点へのリ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 $(n-1)$ 個なので並列累計和アルゴリズムとダブリング法[3] を用いてそれぞれ $O(\log n)$ 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，プロセッサ数が $O(n+m)$ で，時間量は $O(\log m)$ となる．

	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) 新節点番号および旧節点番号の対応表

	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 PERT チャートの有向辺を表す配列



(c) ステージ 1 の結果

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

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

[具体例] ステップ［1.2］における図 1 (a) の新および旧節点番号対応表を**図 4** (a) の配列で示す．このとき添字が新節点番号で，配列要素は旧節点番号（分割節点番号）となる．図 1 (b) の有向辺を表す配列は，図 4 (b) のように新節点番号に置き換わり，それらを並列になぞると，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，( ) 内は節点（ジョブ）の最長所要時間 PT を示す．

3.2 各節点（ジョブ）の最長所要時間を求める並列アルゴリズム（ステージ 2）

ステージ 2 では，ステージ 1 で生成された線形リストを再帰的に併合しながら各節点すなわち各ジョブにおける最長所要時間を求める．線形リストの併合は，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 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い，線形リストの各組を再帰的に併合しながら，最後にリストが 1 つの組になるまで併合を繰り返す．この併合のたびに分割節点の最長所要時間を修正しながら，最後に各分割節点の最長所要時

間を求める並列アルゴリズムである．

はじめに，アルゴリズムの記述に必要な用語の定義を行う．併合する 2 つの線形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，線形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，最長所要時間の増分値を線形リスト間に対数時間で伝播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 種類から構成される．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．上向併合点リストは，併合点となる兄弟節点を**上向辺**（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，線形リスト間をできるだけ階段状に 1 段ずつ上るように連結し，併合点における最長所要時間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，**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）によって連結されたリストである．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]] と上向併合点リスト上で最長所要時間 PT の増分値の累計和を計算する配列 UINC[1..[S[n]] を用い，下向併合点リストに関しても同様にポインタ DLINK[1..S[n]] と下向併合点リスト上で増分値の累計和を計算する配列 DINC[1..S[n]] を用いる．

新節点に 1 台のプロセッサを割り当て，ステージ 2 のアルゴリズムを以下に示す．

ステージ 2 のアルゴリズムの詳細化

前処理

(1) 各分割節点において，兄弟節点の最長所要時間の最大値を求め，各兄弟節点の増分値を求める．詳細を下記に示す．

(1.1) 各分割節点（v=1,⋯,n）において，並列平衡 2 分木法[3] を用いてその兄弟節点の最長所要時間 PT の中で最大値 SMAX[1..n] を求める．

(1.2) 各分割節点における最長所要時間の最大値 SMAX と各兄弟節点の最長所要時間 PT 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，増分値を用いて各節点の最長所要時間を修正する．詳細を下記に示す．

(2.1) 各節点 (u) において，最長所要時間 PT と増分値 INC の和を新たな最長所要時間 PT とする（PT[u] ← PT[u] + INC[u]）．

(2.2) 線形リストにおいて，前節点 u における増分値の修正があれば，後続する各節点 v の最長所要時間 PT[v] を，前節点 u の最長所要時間 PT と前節点からその節点 v までのジョブの所要時間（重み）の和となるように修正する（PT[v] ← PT[u] + PPT[v] − PPT[u]）．

前処理 (2) の詳細を**図 5** に示す．

```
procedure Linked_List_Inc(INC[1..S[n]],var PT[1..S[n]],NEXT[1..S[n]]);
begin
  for u := 1 to S[n] in parallel do begin
    PT[u] ← PT[u] + INC[u];
    Q[u] ← NEXT[u]
  end;
  repeat log n times { ブロードキャストに準じた処理 }
    for u := 1 to S[n] in parallel do
      if Q[u] ≠ 0 then begin
        if INC[u] > 0 then begin
          PT[Q[u]] ← PT[u] + PPT[Q[u]] - PPT[u];
          INC[Q[u]] ← INC[u]
        end;
        Q[u] ← Q[Q[u]]
      end
end;
```

図 5 前処理 (2) のアルゴリズム
Fig. 5 Algorithm for Preprocess (2).

(3) 前処理 (1) を再度実行し，配列 MP，ULINK，DLINK，LISTN を初期化する．

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．

線形リストの総数を L として，ステージ 2 のメインループを以下に示す．

メインループ

loop ← 0
repeat log **L times**
loop ← loop+1;
ステップ［2.1］からステップ［2.5］を処理する

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の併合点を求め，線形リスト間を階段状に連結する併合点リストを構成する．ステップ［2.1］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

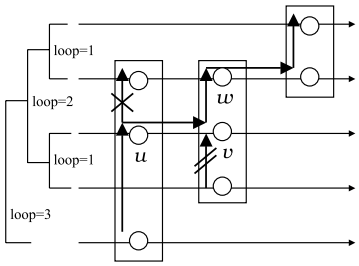


図 6 上向併合点リストの例
Fig. 6 Upward merge point lists.

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，各線形リストの先頭となる併合点は以後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 でそれぞれ連結する．

(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，隣接併合点 v を求める（**図 6** における 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 とは連結しない．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，上記の (3) 上向併合点リストの構成と同様に行う．

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伝播し，各節点の増分値を求める．ステップ［2.2］の詳細を下記に示す．

(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 として，入節点 w の増分値 INC と出節点 v の増分値 INC の差を求める（UINC[w] $\leftarrow$ 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上で増分値 UINC の累計和を求める．

(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 (1) と同様に計算する．

(3) 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 つの値の最大値を用いて，各併合点の増分値 INC を修正する．

```
procedure Merge_List_Inc(var INC[1..S[n]],UNEXT[1..S[n]],DNEXT[1..S[n]]);
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] in parallel do begin
    Q[u] ← UNEXT[u];
    UINC[u] ← 0; DINC[u] ← 0
  end;
  for u := 1 to S[n]-1 in parallel do
    if (MP[u] = 1) and (MP[u+1] = 1) and (ULINK[u+1] = u) then
      UINC[u] ← INC[u] - INC[u+1];
  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(INC[u],UINC[u],DINC[u])
end;
```

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

ステップ［2.2］の詳細を**図 7** に示す．

ステップ［2.3］ 前処理 (2) を実行し，各線形リストにおいて，各節点の増分値 INC を用い各節点の最長所要時間 PT を修正する．

ステップ［2.4］ 前処理 (1) を実行し，各分割節点において，兄弟節点の最長所要時間 PT の最大値 SMAX を求め，各兄弟節点の増分値 INC を求める．

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 全体の概要を**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) では，各分割節点の最長所要時間の最大値は，既知の並列平衡 2 分木法を用いて求めることができる．分割節点における兄弟節点の数はたかだか $(m-1)$ となるので，この処理の計算量はプロセッサ数が $O(m)$ で，時間量が $O(\log m)$ となる．前処理 (2) では，線形リスト上の各節点における最長所要時

```
procedure Stage-2(var PT[1..S[n]],NEXT[1..S[n]]); { 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 PT の最大値 SMAX を求める }
    for i ∈ SIB[v] in parallel do
      SMAX[v] ← Para_Max(PT[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] ← SMAX[N[u]] - PT[u];
  { 前処理 (2) (図 5 参照) }
  Linked_List_Inc(INC[1..S[n]],PT[1..S[n]],NEXT[1..S[n]]);
  { 前処理 (3) } 前処理 (1) と同じアルゴリズムを実行
  for u := 1 to S[n] in parallel do begin { 変数の初期化 }
    MP[u] ← 0; ULINK[u] ← 0; DLINK[u] ← 0; LISTN[u] ← OLISTN[u]
  end;
  { メインループ }
  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) and (I[N[u]] ≠ 0) and (I[N[u+1]] ≠ 0) 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(INC[1..S[n]],ULINK[1..S[n]],DLINK[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.

間の定義に基づき，並列ブロードキャストアルゴリズムに準じた 5 に示されるアルゴリズムによって各節点の最長所要時間が正しく計算される．線形リスト上の節点数はたかだか $(n-1)$ なので，この処理の計算量はプロセッサ数が $O(n)$ で，時間量が $O(\log n)$ である．前処理 (3) は前処理 (1) と同様である．したがって，前処理の計算量はプロセッサ数が $O(n+m)$ で，時間量は $O(\log m)$ となる．

ステージ 2 の繰返し部分では，新節点に 1 台のプロセッサを割り当てる．ステップ［2.1］では，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，併合点を求めるにはリスト番号が奇数（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，その隣接併合点を出節点とする上向辺が存在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．このステップでは，線形リスト上でダブリング法を使って隣接併合点を探せるので，併合点リストを構成する計算量はプロセッサ数が $O(n+m)$ で，時間量が $O(\log n)$ 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，時間量は $O(\log n)$ となる．

ステップ［2.2］では，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ために，各線形リストの増分値の累計和を求める必要がある．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，図 6 および図 7 参照）．

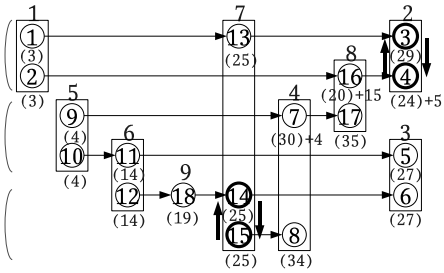
線形リスト間を連結する上向辺 $(v \to w)$ は，出節点 v の真の増分値を入節点 w へ伝達する役割を持つので，入節点 w に伝達される増分値 UINC$[w]$ は UINC$[w] \leftarrow$ INC$[w]-$ INC$[v]$ とする．なぜなら，出節点 v と入節点 w は兄弟節点であり，それらの増分値 INC$[v]$ と INC$[w]$ には，それらの兄弟節点が属する分割節点で最大の最長所要時間に基づく増分値が格納されている．したがって，入節点 w に伝達される増分値 UINC$[w]$ は INC$[w]$ と INC$[v]$ の増分値の差を格納する必要がある．その結果，上向併合点リストは併合点の（上向併合点リストにおける）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上向の場合と同様である．そして，各節点の増分値 INC は 3 つの増分値 UINC，DINC および INC の中の最大値とする．

ステップ［2.2］における計算量は，増分値の累計和を並列累計和アルゴリズムにより求めることができるので，時間量は $O(\log m)$ となる．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，時間量が $O(\log m)$ となる．

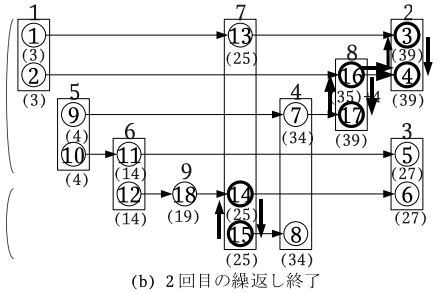
ステップ［2.3］における計算量は前処理 (2) と同様であり，また，ステップ［2.4］における計算量については前処理 (1) と同様となる．最後にステップ［2.5］では，リスト番号更新の計算量はプロセッサ数が $O(n+m)$ で，時間量は $O(1)$ である．

したがって，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，最終的にステージ 2 の計算量は，プロセッサ数が $O(n+m)$ で時間量が $O(\log^2 m)$ となる．

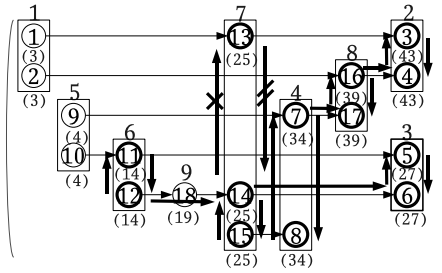
[具体例] 図 4 (c) で示される線形リストに，ステージ 2 の前処理を適用して各節点の増分値を求めた後，ステージ 2 のメインループを実行した結果を**図 9** に示す．図 9 の各節点において，( ) 内の値はその最長所要



(a) 1回目の繰返し終了



(b) 2回目の繰返し終了



(c) 3回目の繰返し終了

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

時間 PT を表し，“+” の後の値はその増分値 INC を表す．また，増分値 INC が 0 の場合は “+” と数値を省略している．そして，太線で示された節点は併合点を表し，太線で示された有向辺は併合点リストを表す．

1 回目の繰返しでは（図 9 (a)），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 が併合点となるが，併合点リストからの増分値の修正はない．

2 回目の繰返しでは（図 9 (b)），分割節点 **8** の兄弟節点 16 と 17 が新たに併合点となり，このとき併合点 16 は隣接併合点 4 に上向併合点リストとして上向辺で連結される．上向併合点リストの増分値 UINC は UINC[16]=+15，UINC[4]=+15，UINC[3]=+10 となり，下向併合点リストの増分値 DINC は DINC[16]=0，DINC[4]=+5，DINC[3]=0 となる（図 9 (a) において INC[16]=+15，INC[4]=+5，INC[3]=0 より）．ステップ［2.2］において，最終的な増分値が INC[16]=+15，INC[4]=+15，INC[3]=+10 となるので，最長所要時間 PT の値は PT[16]=35，PT[4]=39，PT[3]=39 となる（図 9 (b) 参照）．

最終（3 回目）の繰返しでは（図 9 (c)），上向併合点リスト 8→7→17→16→4→3 および 15→14→6→5 が生成され，上向辺 (14→13) が削除される（× 印）．下向併合点リストについては，節点 12 が節点 14 に連結されて下向辺 (13→14) が削除され（//印），11→12→14→15 が生成される．また，増分値 INC が 0 でない節点は 16 だけなので（図 9 (b) において，INC[16]=+4），節点 16 を含むリスト 8→7→17→16→4→3 上でのみ増分値が修正される．よって，このリスト上での増分値 UINC は，UINC[8]=0，UINC[7]=0，UINC[17]=0，UINC[16]=+4，UINC[4]=+4，UINC[3]=+4 となる．その結果，各節点の最長所要時間 PT は図 9 (c) のように計算され，PT[16]=39，PT[4]=43，PT[3]=43 となる．

3.3 クリティカルパスを求める並列アルゴリズム（ステージ 3）

ステージ 3 では，各節点すなわち各ジョブにおける最早開始時刻と最遅開始時刻および余裕時間を求め，これらの時間を用いてクリティカルパスを構成する節点と経路の判定を行い，PERT チャートにおけるクリティカルパスを求める．

まず最早開始時刻に関しては，ステージ 2 で求めた線形リストにおける各節点の最長所要時間を用いて求める．次に最遅開始時刻に関しては，ステージ 1 で生成した線形リストに対して各リストの向きを逆転させた線形リストを構成して求める．この線形リストをステージ 1 で生成した線形リストに対して，**逆向線形リスト**と呼ぶ．この逆向線形リストに対して，線形リストと同様にステージ 2 を適用して逆向線形リストにおける最長所要時間が求められる．ここで，逆向線形リストにおける各節点では，線形リストにおける配列 NEXT および PT と同様に，次節点を示す RNEXT[1..S[n]] と最長所要時間 RPT[1..S[n]] を用いる．また，線形リスト上の各節点において，最早開始時刻 ET[1..S[n]]，最遅開始時刻 LT[1..S[n]]，余裕時間 FT[1..S[n]] の配列を用いる．

ステージ 3 のアルゴリズムを以下に示す．

ステップ［3.1］ ステージ 2 で求めた各節点の最長所要時間より，全体の完了時刻と各節点の最早開始時刻を求める．詳細を下記に示す．

(1) 並列平衡 2 分木法を用いて線形リストにおける最長所要時間 PT の最大値を求め，それをプロジェクト全体の完了時刻 PTMAX とする．

(2) 各節点 (u) において，最早開始時刻 ET として線形

リストにおけるその最長所要時間 PT とそのジョブの所要時間（重み）NT の差を求める（ET[u] ← PT[u]−NT[u]）.

ステップ［3.2］ まず逆向線形リストを生成し，これにステージ 2 を適用して逆向線形リストにおける各節点の最長所要時間を求める．次に，これを用いて最遅開始時刻と余裕時間の計算を行う．詳細を下記に示す.

(1) 逆向線形リストの生成
  (1.1) RNEXT を用いて，線形リストにおける入節点と出節点を逆転させてステップ［1.2］(5) と同様に有向辺で連結する.
  (1.2) ステップ［1.3］(1) と同様に，逆向線形リスト上で各節点の重みすなわちジョブの所要時間 NT の累計和を計算して，逆向線形リストにおける各節点の最長所要時間 RPT の初期値として格納し，また，この初期値を後で参照するため配列 PPT にも保存する.

(2) 逆向線形リストにステージ 2 を適用し，逆向線形リストにおける各節点の最長所要時間 RPT を求める.

(3) 各節点（u）において，最遅開始時刻 LT としてプロジェクト全体の完了時刻 PTMAX と逆向線形リストにおけるその最長所要時間 RPT の差を求め（LT[u] ← PTMAX−RPT[u]），また，余裕時間 FT としてその最遅開始時刻 LT とその最早開始時刻 ET の差を計算する（FT[u] ← LT[u]−ET[u]）.

ステップ［3.3］ 線形リスト上の各節点すなわち各ジョブにおけるその余裕時間と最早開始時刻を用いて，クリティカルパスを構成する節点と有向辺の判定を行い，PERT チャートにおけるクリティカルパスを求める．詳細を下記に示す.

(1) クリティカルパスに含まれる節点の判定を行う．各有向辺の出節点と入節点においてその余裕時間 FT がともに 0 であり，かつ出節点における最早開始時刻 ET とそのジョブの所要時間（重み）NT の和が入節点の最早開始時刻 ET と等しい場合，配列 MARK[1..S[n]] を用いてその出節点にクリティカルパスとしての印を付ける．また，入節点が末尾でありかつその最長所要時間 PT が最大すなわち PTMAX と等しい場合，その入節点にも印を付ける.

(2) クリティカルパスに含まれる有向辺の判定を行う．各有向辺において，その出節点が MARK により印付けされている場合，配列 CP[1..m] を用いてその有向辺にクリティカルパスとしての印を付ける.

```
procedure Stage-3;
{ ステージ 3 の並列アルゴリズム }
begin
  { ステップ [3.1] }
  PTMAX ← Para_Max(PT[1..S[n]]); { 並列平衡 2 分木法 }
  for u := 1 to S[n] in parallel do
    ET[u] ← PT[u] - NT[u];
  { ステップ [3.2] }
  { (1) 逆向線形リストの生成 }
  for e := 1 to m in parallel do begin
    RNEXT[NIV[e]] ← NOV[e];
  end;
  Para_Prefixsum(NT[1..S[n]],RPT[1..S[n]]);
  for u := 1 to S[n] in parallel do { RPT の初期値を保存 }
    PPT[u] ← RPT[u];
  { (2) 逆向線形リストにステージ 2 を適用 }
  Stage-2(RPT[1..S[n]],RNEXT[1..S[n]]);
  { (3) LT,FT の計算 }
  for u := 1 to S[n] in parallel do begin
    LT[u] ← PTMAX - RPT[u];
    FT[u] ← LT[u] - ET[u]
  end;
  { ステップ [3.3] }
  for u := 1 to S[n] in parallel do
    if NEXT[u] ≠ 0 then
      if (FT[u] = 0) and (FT[NEXT[u]] = 0) and
      (ET[u] + NT[u] = ET[NEXT[u]]) then begin
        MARK[u] ← 1;
        if (NEXT[NEXT[u]] = 0) and (PT[NEXT[u]] = PTMAX) then
          MARK[NEXT[u]] ← 1
      end;
  for e := 1 to m in parallel do
    if MARK[NOV[e]] = 1 then CP[e] ← 1
end;
```

図 10 ステージ 3 のアルゴリズム
Fig. 10 Parallel algorithm for Stage 3.

ステージ 3 のアルゴリズムの概要を疑似コードで**図 10** に示す.

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

ステップ［3.1］では，節点の最長所要時間 PT の最大値が所与の PERT チャートにおけるプロジェクト全体の完了時刻 PTMAX となり，各節点の最早開始時刻 ET がステージ 2 で求めた線形リストにおけるその最長所要時間 PT とそのジョブの所要時間である重み NT の差として求まるのは明らかである．これらを求める計算量は，プロセッサ数が $O(m)$ で，時間量が $O(\log m)$ となる.

ステップ［3.2］では，逆向線形リストにおける節点の最長所要時間 RPT を用いて各節点の最遅開始時刻 LT と余裕時間 FT を求めることができる．逆向線形リストにステージ 2 を適用すると逆向線形リストにおける最長所要時間 RPT が求まる．各節点の最長所要時間 RPT は逆向線形リスト上における先頭からその節点までの最長所要時間を表すので，これは線形リスト上におけるその節点から末尾となる節点までの最長所要時間に等しい．したがって，節点 u における最遅開始時刻 LT[u] は，プロジェクト全体の完了時刻 PTMAX と逆向線形リストにおける最長所要時間

RPT[u] の差として求まる．このステップでは前述のアルゴリズムを用いるので，これらを求める計算量はプロセッサ数が $O(n+m)$ で，時間量が $O(\log^2 m)$ となる．

ステップ［3.3］では，PERT における性質からクリティカルパスに含まれる各節点はその余裕時間 FT が 0 となるので，クリティカルパス上の節点が容易に判定でき，その正当性は明らかである．クリティカルパスに含まれる線形リスト上の有向辺 $(u \to v)$ については，出節点 u における最早開始時刻 ET[u] とそのジョブの所要時間（重み）NT[u] の和が入節点 v における最早開始時刻 ET[v] と等しくなる有向辺のみを選定することで正しく判定できる．その結果，所与の PERT チャートにおけるクリティカルパスがすべて求まる．このステップにおける計算量は，プロセッサ数が $O(n+m)$ で，時間量が $O(1)$ となる．

したがって，ステージ 3 における計算量はプロセッサ数が $O(n+m)$ で，時間量が $O(\log^2 m)$ となる．

［具体例］ ステップ［3.1］において，図 1 (a) で示された PERT チャートにおけるプロジェクト全体の完了時刻 PTMAX は，図 9 (c) より（分割節点 **2** の）節点 3 および 4 における 43 となる．

次にステップ［3.2］において逆向線形リストにステージ 2 を適用した結果を**図 11** に示す．図 11 における各節点の ( ) 内の値は，逆向線形リスト上の先頭（線形リスト上の末尾）からその節点までの最長所要時間 RPT を表す．ステップ［3.2］の結果，各節点における最早開始時刻，最遅開始時刻および余裕時間は**図 12** のように求まる．図 12 において，各節点における ( ) 内の値は左から順にそれぞれ最早開始時刻 ET，最遅開始時刻 LT，余裕時間 FT を示す．

図 12 はステージ 3 終了後の結果を線形リスト上で表したものである．図 12 において，余裕時間 FT が 0 となる太線で示された節点 4，7，10，12，15，16，18 がステップ［3.3］で印付けされたクリティカルパスに含まれる節点となる．ここで，クリティカルパスに含まれない有向辺について見てみると，余裕時間 FT が 0 でない入節点または出節点を持つ有向辺は明らかに含まれず，このような有向辺は図 12 において//印で示される．次に入節点および出節点における余裕時間 FT がともに 0 となる有向辺では，出節点における最早開始時刻 ET とそのジョブの所要時間 NT の和が入節点における最早開始時刻 ET と等しくない場合にクリティカルパスに含まれない．そのような有向辺を図 12 において × 印で示している．したがって，図 12 において//および × 印の付いていない太線で示された有向辺 (10→11)，(12→18)，(18→14)，(15→8)，(7→17)，(16→4) がクリティカルパスに含まれる有向辺として選定される．このとき，各節点は新節点番号で表されるが，これは旧節点番号（分割節点番号）で容易に表すことができる．よって，これを所与のジョブ番号を示す分割節点番号で表すと，所与の PERT チャートにおけるクリティカルパスが，**図 13** において太線で示される $5 \to 6 \to 9 \to 7 \to 4 \to 8 \to 2$ と求まる．

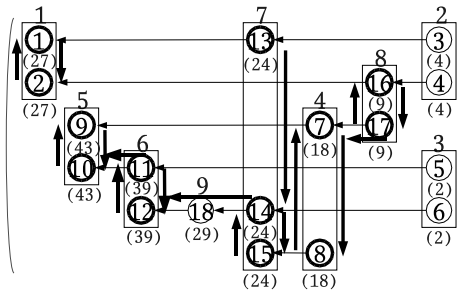


図 11 ステップ［3.2］終了時における逆向線形リストの結果
Fig. 11 Result of the backward linked list after Step [3.2].

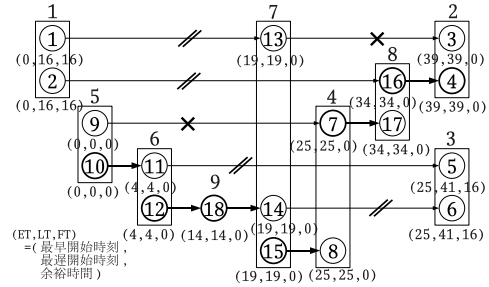


図 12 ステージ 3 終了後の結果
Fig. 12 Result after Stage 3.

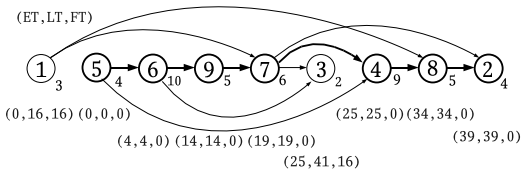


図 13 所与の PERT チャートにおけるクリティカルパス
Fig. 13 Critical path in the given PERT chart.

4. ま と め

PERT チャートのクリティカルパスを求める効率の良い並列アルゴリズムを提案した．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に基づき，基本的な並列アルゴリズムのみを用いたアルゴリズムである．この並列アルゴリズムの計算量は

CREW-PRAM 計算機モデルのもとで，プロセッサ数が $O(n+m)$，時間計算量が $O(\log^2 m)$ である．

提案する並列アルゴリズムのプロセッサ数は $O(n+m)$ となるが，PERT チャートが密な場合でもプロセッサ数はたかだか $O(n^2)$ であり，また，疎な場合には $O(n)$ であるため，PERT チャートの疎密に対応してプロセッサ数を変動でき，クリティカルパスを効率良く求めることができる．

提案した並列アルゴリズムは，PERT チャートのクリティカルパスの経路をすべて求めることができるとともに，各ジョブにおける最早開始時刻，最遅開始時刻や余裕時間をも求めることができる．

今後の課題として，より現実的な並列計算機モデルである EREW-PRAM モデルを前提とした並列アルゴリズムへの詳細化や，提案した並列アルゴリズムの設計法を用いて強連結成分を求める並列アルゴリズムの設計が考えられる．

参 考 文 献

1) 関根智明：PERT の手法，PERT·CPM，pp.35–46, 日科技連出版社 (1977).
2) 多田昭雄，右田雅裕，中村良三：並列トポロジカル整列アルゴリズム，情報処理学会論文誌，Vol.45, No.4, pp.1102–1111 (2004).
3) Gibbons, A. and Rytter, W.: *Efficient Parallel Algorithms*, pp.6–18, Cambridge University Press (1988).
4) Xavier, C. and Iyegar, S.S.: *Introduction to Parallel Algorithms*, pp.108–140, Wiley-interscience (1998).
5) Cole, R.: Parallel Merge Sort, *SIAM J. Comput.*, Vol.17, No.4, pp.770–785 (1988).
6) 宮野 悟：並列ランダムアクセス機械，並列アルゴリズム，pp.39–44, 近代科学社 (1993).

(平成 17 年 8 月 15 日受付)
(平成 18 年 4 月 4 日採録)

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 年熊本大学大学院工学研究科電気情報工学専攻修士課程修了．2000 年熊本大学大学院自然科学研究科後期博士課程システム科学専攻単位取得退学．現在，熊本大学総合情報基盤センター助手．並列アルゴリズムの設計と解析に関心を持っている．本会の平成 16 年度論文賞を受賞．

多田 昭雄（正会員）

1966 年京都大学工学部数理工学科卒業．2004 年熊本大学大学院自然科学研究科後期博士課程修了．博士（工学）．1966 年東京芝浦電気（現東芝）入社．1994 年熊本工業大学（現崇城大学）講師．現在，同大学情報学部コンピュータシステムテクノロジー学科教授．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．本会の平成 16 年度論文賞を受賞．

糸川 剛（正会員）

1997 年熊本大学大学院自然科学研究科修士課程修了，2001 年熊本大学大学院後期博士課程修了．博士（工学）．同年，熊本大学工学部数理情報システム工学科助手を経て現在，熊本大学大学院自然科学研究科情報電気電子工学専攻助手．研究の中心はアルゴリズムとデータ構造の設計と解析．

中村 良三（正会員）

1964 年防衛大学校応用物理専攻卒業．1968 年熊本大学大学院電気工学専攻修士課程修了．中部電力株式会社を経て，1975 年熊本大学工学部勤務．現在，熊本大学工学部数理情報システム工学科教授．工学博士．アルゴリズムとデータ構造の設計と解析に興味を持っている．本会の平成 16 年度論文賞を受賞．