計算機アーキテクチャ 58－2
（１９８５・６・２１）

データフローコンピュータにおけるプロシジャコールとループ処理の実現方法

Implementation of Procedure Call and Loop Processing on Dataflow Computer

會和　将容，　菊池　道夫
（群馬大学　工学部）

１．まえがき

現在，大量のデータ解析や知能処理などの研究が進むにつれて，並列処理可能コンピュータ実現への要求が高まっている。並列処理コンピュータは，直列処理を専用とするノイマンコンピュータとは根本的に異なるので，ノイマンコンピュータで使われていた多くの概念をそのまま並列処理コンピュータに適用すると不都合を生じる場合も多い。たとえば，引数を多くもつプロシジャ呼出しでは，引数が並列的に作られ，それらがまた並列的に入力されるので，ノイマンコンピュータのように，すべての引数がそろったとき，プロシジャの実行を始めるという方法では，実行効率が低下してしまう。また，１つのプロシジャが同時に多くのプログラムから呼び出されることがあるので，お互いのデータが干渉をおこし間違った結果をかえすということが起りうる。ループ処理でもループによって戻されるデータが複数個になるのが普通であるし，また，あるサイクルのデータと次のサイクルのデータが混じり合うので，プロシジャコールの場合と同様なことがおこりうる。これらの問題を解決するために，プロシジャコールではプログラムを呼出しごとにコピーする方法［１］と，データに色づけする方法［２］が考えられている。ループ処理では，データにカウント値をつけ，ループ本体に再入するごとにカウント値を１増やす方法やテールリカージョンによってループ処理を代用する方法などが考えられている。また，プロシジャとループなどを統一的に取り扱う方法として，プログラムをブロックと呼ばれる単位にわけ，ブロックやプロシジャ，ループ処理それぞれ用のタグをデータにつけ，上記の問題を解決しようとする試みもなされている［３］。ループカウントやタグをつける方法は，結果的にはデータの色づけと原理は同じであるので，結局，上記の問題は，展開による方法か色づけによる方法によって解決されていることになる。

しかしながら，展開による方法は，呼出しごとにプログラムのコピーを作るということから，時間も遅く，メモリ量も多くなる。色づけによる方法は，多くの色を必要とし，また色管理が集中して，並列処理ではそこがボトルネックになる可能性がある。テールリカージョンによるループ処理の実現は，プロシジャコールのオーバヘッドの大きさのため，ループ処理より効率が悪くなるのが普通である。

一般に，これらのプロシジャコールやループ処理は並列に行われるので，資源のアクセス競合が起り，そのため排他制御が必要となり，直列処理では考えられないほどその構造や制御が複雑になることが多かった。

本論文は，すでに開発されたデータフローコンピュータをモデルに，色づけを基本としたプロシジャコールとループの実現［４］に関するものである。ここでは，プロシジャコールに関しては，プロシジャ内で色を割りつけるという方法により，色の数を減らし，色管理の分散を図った方法（プロシジャユニーク色づけと呼ぶ）を提案する。また，ループ処理に関しては，テールリカージョンとループ処理が，並列処理ではほぼ等価になるこに注目してこれらを統一的に取り扱い，使用済みの色を再使用するという方法で，色の減少と高速性を同時に満たす方法を提案する。

２．データフローコンピュータ

図１に本論文で対象とするデータフローコンピュータの構造を示す。ＴＭはトークンメモリと呼ばれるメモリで，トークンと呼ばれるデータを格納する。ＦＵはプロセッサで，Ｍはプログラムやその他の情報を格

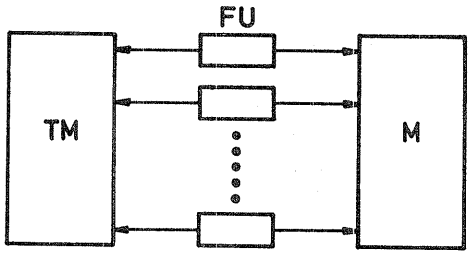

図１　　データフローコンピュータの構成

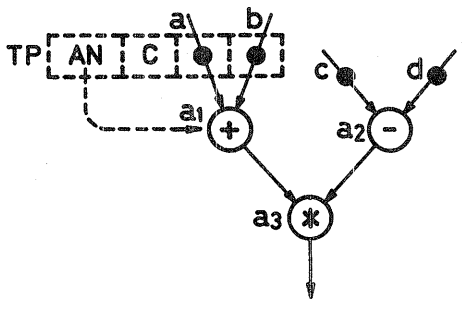

図２　（ａ＋ｂ）（ｃ－ｄ）を計算するデータフロープログラム

納するメモリである。図2はデータフロープログラムであり，（a＋b）（c－d）を計算している。データフロープログラムの命令は，アクタと呼ばれており，a_1 ～a_3 はそれぞれのアクタにつけられた名前（アクタネーム）である。黒丸はトークンである。各アクタは処理に必要なトークンが入力アークに到着したとき発火され，結果のデータを出力アークに出力して鎮火する。このコンピュータでは，同図破線で示されているように，同一のアクタに入力されるトークンは，トークンパケット（ＴＰ）として1まとめにして取り扱われる。トークンパケットには，トークンの他に，そのトークンが入力されるべきポインタ（アクタネーム（ＡＮ））とトークンの色を表わすカラー（Ｃ）が接続される。コンピュータシステム全体に対して同じ色がただ1つしか存在しない色づけ方法を，ここでは，システムユニーク色づけと呼ぶ。トークンパケットのうち，1つのアクタを発火させるのに必要なすべてのトークンがそろっているトークンパケットのことを完全トークンパケット（Ｃｏｍｐｌｅｔｅ　ｔｏｋｅｎ　ｐａｃｋｅｔ（ＣＴＰ））と呼び，処理結果であるトークンが，ただ1つだけ入ったトークンパケットを結果トークンパケット（ｒｅｓｕｌｔａｎｔ　ｔｏｋｅｎ　ｐａｃｋｅｔ（ＲＴＰ））と呼ぶ。

図3は色づけによるプロシジャコールを表わす図である。図では，プロシジャの入出力引数が2個であり，プロシジャAが，プロシジャコールアクタa_1 とa_2 から呼ばれていると仮定されている。プロシジャコールアクタa_1，a_2 に到着しているトークンは白色であり，それがプロシジャに入力されるときには，青色と赤色にそれぞれ色が変えられ，プロシジャからトークンが出力されるときには，この逆の色づけが行なわれる。本論文で提案するプロシジャユニーク色づけは，たとえば，プロシジャ内の色に赤のかわりに白を使うといったように，呼出しプログラム内での色と呼び出されたプロシジャ内のトークンの色が同一でもよい。プロシジャユニーク色づけでは，トークンはプロシジャ名とカラーによって区別される。データフローコンピュータでは，トークンに，アクタネームを付加し，パケットとして用いるのが一般的である。したがって，プロシジャ名のかわりに，このアクタネームを用い，アクタネームと色とにより，トークンを区別することにすれば，プロシジャネーム用のハードウェアを追加することなくプロシジャユニーク色づけが実現できる。また，この色づけはプロシジャ単位で行なうことが出来るので，この方法は，ハードウェア量を増やすことなく，カラー管理を集中することなく多量のカラーを準備する効果的な方法である。プロシジャは多くの引数入力と引数出力を持つのが普通である。したがって，他のアクタのようにすべてのトークンが入力アークに到着したとき，プロシジャコールアクタを発火させ，プロシジャを活性化するのは，効率が悪い。したがって，ここでは，1つでもトークンが到着すると，プロシジャコールアクタが発火し，そのトークンがプロシジャにわたされ，また，プロシジャからのトークン出力は，1つでもプロシジャからプロシジャアクタの方に戻されるものとする。この場合，あるプロシジャコールアクタからプロシジャに渡される引数のうち，一番最初にプロシジャに到着するトークンをファースト入力トークン，それ以降に到着するトークンをアフタ入力トークン，最後に到着するトークンをラスト入力トークンと便宜上呼ぶことにする。ファースト入力トークンとアフタ入力トークン，ラスト入力トークンは同じ色に着色されなければならない。このようにすると，プロシジャ内では色によりトークンを区別することができるので，データが混ざりあって処理結果が誤った値になることはない。もちろん，プロシジャ内で，色の違ったそれぞれのトークンは，お互いに何の関係もなく，各アクタも，色の同じ必要なトークンが到着したときのみ発火する。

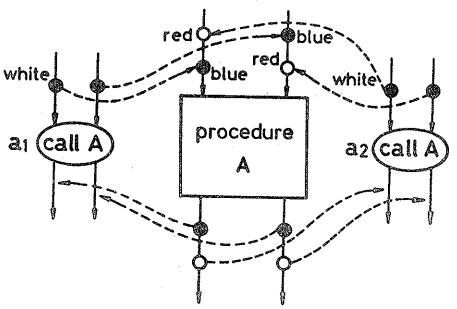

図3　　プロシジャコール

コンピュータは次のように動作する。1）ＦＵは，ＴＭ内で，アクタを発火するに十分なトークンが入ったトークンパケット（完全トークンパケット　Ｃｏｍｐｌｅｔｅ　ｔｏｋｅｎ　ｐａｃｋｅｔ　（ＣＴＰ））を取り出す。2）ＣＴＰのアクタネームをもとに，アクタ（命令）をメモリから取り出す。3）アクタを実行する。4）結果のデータにアクタネームとカラーをつけ，結果トークンパケットとしてＴＭに格納する。5）1）に戻る。ＴＭは連想メモリ機能を持っており，この機能により，もし同じアクタ名とカラーを持った結果トークンパケットがＴＭ内にあれば，それらはＣＴＰを作るために，ＴＭ内の同一ワード内に集められる。

アクタとトークンの構成を表わすために，ＢＮＦ記号を変形して用いることにする。アクタの構成は，次のように書くことができる。

<actor>::=<actor name><instruction>
[<output arc>]
<output arc>::=<destination actor name>
<destination actor's input arc No.>
<procedure indicator>

ここで，ブラケット [] は内部に書かれたものの有限回のくりかえしを表わし，下線部分はわかりやすくするためにつけられたものである。<destination actor name>は，結果のトークンが送られるべきアクタを示し，<destination actor's input arc No.> は，出力アークが，送り先アクタの何番目の入力アークであるかを表わす。<procedure indicator> は，<output arc>の指すアクタがプロシジャコールアクタと，後に述べるループリカージョンアクタに関するものであることを表わす。

トークンパケットは次のように表わされる。

<complete token packet>::=
<destination actor name>
<color>[<actor's input
arc No.><data>]
<resultant token packet>::=
<destination actor name>
<color><actor's input
arc No.><data>
token::=<color><data>
<color>::=<basic color>
<sub_color><sub_color
overflow indicator>

<color> は一般には，<basic color> と<sub_color> とを分けずに使用され，ループリカージョンの場合のみ，これらを分けて取り扱う。<sub_color overflow indicator>は<sub_color> がオーバフローしているかどうかを表わす。

本コンピュータの動作をこれらの記号を用いて表わすと図4のようになる。図の（）内は主に，オペレーションのためのデータベースを表わし，たとえば，（3）のexecute <actor$_1$>(<actor$_1$>.<instruction>,<complete token packet>.[<data>]) は，（2）でフェッチしたアクタ<actor$_1$>を，その命令と（1）でフェッチしたＣＴＰのデータを用いて実行することを表わす。（5）TM← make <resultant token packet>(<resultant data>,<actor$_1$>.<output arc>)は，結果のデータに出力アークを接続して結果トークンパケットを作り，その結果トークンパケットをトークンメモリに入れることを表わしている。（4）～（6）は出力アークの数だけ繰り返される。（7）以降は，プロシジャコールとループリカージョンに関する処理である。すなわち，（7）でフェッチしたアクタの命令の種類によって，それぞれの動作アルゴリズムに入る。（8）以降のアクタにつけられた番号は，（7）の<actor$_2$>を，たとえば，説明の便宜上<actor$_3$>と読み替えることを表わす。

3．プロシジャコール

データフロー処理のプロシジャコールでは，引数が複数個あり，その引数も時間的にランダムにプロシジャコールアクタに到着するのが普通である。また，1つのプロシジャは，複数個のプロシジャコールアクタによって，同時に呼び出されることもある。以上のよ

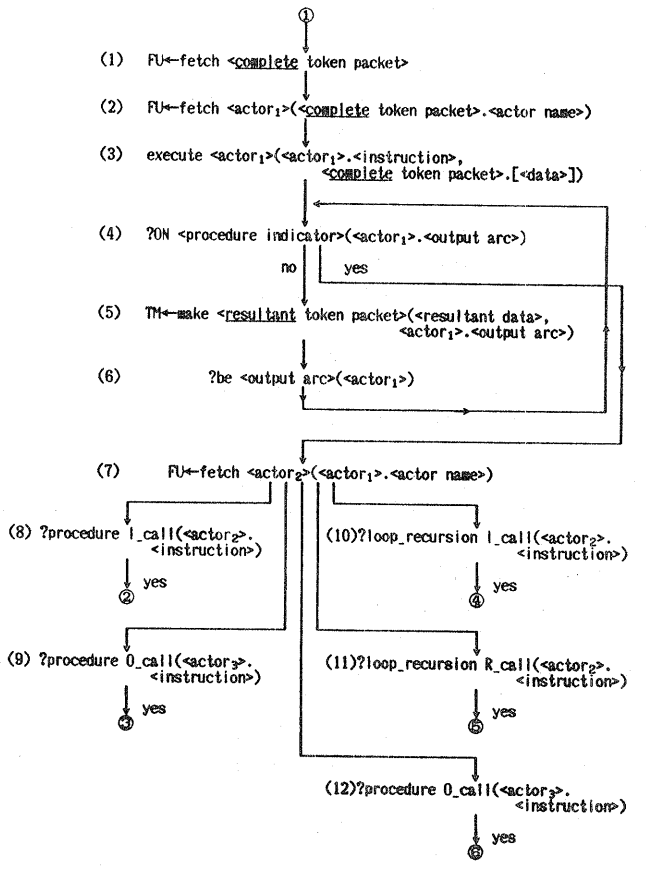

図4　　アクタの発火

うなことを考慮にいれ，プログラムの効率的実行をはかるため，プロシジャコールアクタとプロシジャの実行開始と終了を次のように定める。

実行開始；プロシジャコールアクタは，ファースト入力トークンが到着したときｆｉｒｅし，その結果，それによって呼ばれたプロシジャがアクティブになる。

実行終了；アクティブになったプロシジャは，そのプロシジャ上にトークンが存在しなくなったときノンアクティブになる。プロシジャコールアクタは，プロシジャがノンアクティブになった後，鎮火する。

プロシジャは

```
<procedure>::=<procedure name><attribute>
              <procedure body>[<procedure
              0_actor>]
<procedure 0_actor>::=<actor name>
              <procedure 0-actor instruction>
              <procedure name>
```

の構成にて表わされる。<procedure 0_actor> は，プロシジャからトークンを出力するためのアクタである。<attribute> にはこのプロシジャの属性が入れられる。プロシジャコールアクタは

```
<procedure 1_call actor>::=<actor name>
              <procedure 1-call instruction>
              <procedure name>[<input arc>]
              [<output arc>]
<input arc>::=<output arc>::=<called actor
              name><called actor's input arc
              No.><procedure indicator>
```

<procedure 1_call actor>は，プロシジャを実行開始し，トークン（引数）をプロシジャに入力するためのアクタである。

色づけを使った並列処理のプロシジャコールでは，１つのプロシジャに複数個の引数（トークン）が時間的にランダムに到着するが，それらは，すべて同じ色に着色されなければならないし，また，出力時には元の色に戻されなくてはならない。したがって，あるプロシジャコールアクタの発火によってプロシジャがアクティブになったとき，そのコールのプロシジャ内のトークンに割り当てられた色（ｉｎ－ｃｏｌｏｒと呼ぶ）は，アフタ入力トークンの着色のために記憶されなければならないし，また，トークン出力のとき，そのプロシジャに入るまえの色（ｏｕｔ－ｃｏｌｏｒと呼ぶ）に戻すために，元の色を記憶しなければならない。このような情報を記憶する場所として，コールバケットと呼ばれる記憶場所を用いる。コールバケットの構成は

```
<call packet>::=<in-color><procedure 1-call
                actor name><out-color>
```

である。

プロシジャコールは，図５に示すように行なわれる。すなわち，ファースト入力トークンの到着により<procedure 1_call actor>が発火し，プロシジャをアクティブにする。プロシジャの実行終了は，そのプロシジャを呼んでいるすべてのプロシジャコールアクタの鎮火，プロシジャのノンアクティブ化の順に行なわれる。ここでは便利のため最初にプロシジャがアクティブになることをプロシジャのオープン，最後にプロシジャがノンアクティブになることをプロシジャのクローズと呼ぶ。

コールバケットはメモリ内のコールバケット域に格納され，コールバケット域は，プロシジャがオープンされたとき，それぞれのプロシジャに対して与えられる。コールバケット域へのポインタはコールバケット域バケット

```
<call packet area pointer packet>::=<procedure
          name><nil><call packet area pointer>
```

として，コールバケトへのポインタはコールバケットポインタバケットとして

```
<call packet pointer packet>::=
        <procedure 1-call actor name><color>
        <call packet pointer indicator>
        <call packet pointer>
```

としてＴＭに格納される。

<call packet pointer packet>は，アフタ入力トー

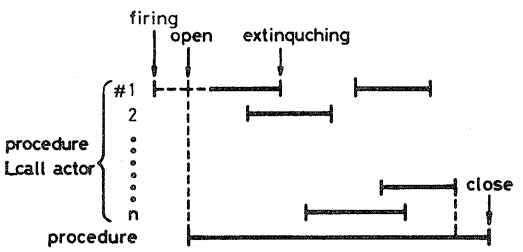

図５　プロシジャコールアクタの発火とプロシジャのオープン，クローズ

クンを色づけするためのコールパケットを捜すために用いられ，<call packet area pointer packet> は，プロシジャから結果トークンを出力するとき，その出力に対するコールパケットをさがすために用いられる。

図６に，プロシジャコールの開始アルゴリズムを示す。このアルゴリズムは到着する引数の種類によって，３つの異なったルートを通る。ａ）プロシジャがオープンされておらず，到着した引数がファースト入力トークンの場合は，このアルゴリズムを上から順にすべて実行し，ｂ）プロシジャがすでにオープンされていて，到着した引数がアフタ入力トークンの場合は，最初の分岐を通り，また，ｃ）プロシジャがすでにオープンされているが，到着した引数がファースト入力トークンの場合は，２番目の分岐を通る。

ａ）の場合は，（１）で，トークンがファースト入力トークンかアフタ入力トークンかを区別し，（２）でプロシジャがオープンされているかどうかをチェックする。そして，（３）でコールパケット領域を割り当て，（４）でポインタを作り，（５）でプロシジャ内での新しい色を割りあてる。次には（６）でコールパケットを作り，（７）でそのポインタをＴＭに入れ，最後に（８）で，引数であるトークンをプロシジャ内に送る。破線で接続された部分はクリティカルセクションで，排他制御されなければならない部分である。これらは，ハードウェアに用意されているＴｅｓｔ ａｎｄ Ｓｅｔ機構により実現される。

プロシジャからのトークン出力の場合も，そのプロシジャコールに対するラスト入力トークンであるのかないのか，そのプロシジャが他のプロシジャコールアクタによって同時に呼ばれているのかどうかによってアルゴリズムが異なる。出力トークンがラスト入力トークンでない場合は，ただトークンの色を元にもどして出力するだけでよいし，ラスト入力トークンの場合には，トークンを出力した後，コールパケットを消去しなければならない。そして，次に，そのプロシジャが他のプロシジャコールアクタにより呼ばれているかどうかを調べ，呼ばれていない場合はプロシジャをクローズしなければならない。トークン出力時の詳細なアルゴリズムは図７のようになる。

すなわち，（１）でコールパケットを見つけるためのポインタを取り出し，（２）でコールパケットを取り出す。（３）でコールパケットの情報をもとに，<procedure I_call actor>を取り出し，トークン出力先アークを知り，（４）では結果トークンパケットを作り出力する。（５），（６）はプロシジャからの出力トークンがラスト入力トークンかどうか見ており，（７），（８）ではプロシジャのクローズを行なう。（５），（７）のプロシジャ内にトークンが存在するかどうかの判定は，トークンメモリの連想機能を使って行なわれる。

４．テールリカージョンとループ

図８は２引数のテールリカージョン処理の原理を表わす図である。i_1，i_2，・・・，i_n は，呼び出されたプロシジャを図式化したもので，ここではこれらをインスタンスと呼ぶ。テールリカージョンでは，プロシジャの出口直前で自分自身を呼ぶので，図のように引数の送受が行なわれる。それぞれの呼出しごとにトークンの色づけが行なわれ，また，それぞれの戻りごとに色が元に戻される。図９はループ処理を同様にインスタンスで表わしたものである。ループ処理でも，引数が複数個あるのが普通であるので，ｉ回目のトークンとｉ＋１回のトークンがまじりあう。したがって，それをさけるためにプロシジャの場合と同様に，各インスタンスに入るときにトークンの色づけを行い，最終インスタンス i_n を出るときには，トークンの色を i_1 に入るまえの色に戻す。ループでは，同じ内容を持ったインスタンスが多数回繰りかえされるので，

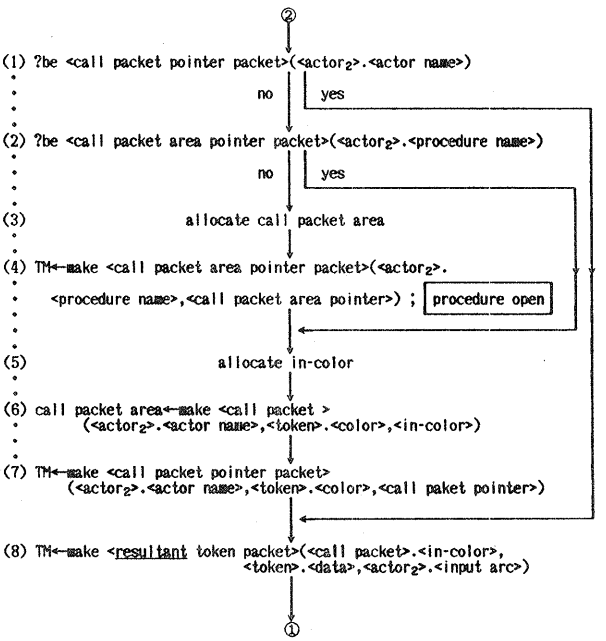

図６　プロシジャコールの開始アルゴリズム

トークンの色づけに順序性を持たす方法が取り入れられる場合が多い。同じ内容を持ったインスタンスが多数回繰りかえされることはテールリカージョンの場合も同じであるので，この場合も色に順序性を持たすことができる。したがって，テールリカージョンとループ処理は並列処理においては機能的にはほとんど同一であると見なすことがでる。

テールリカージョンでは各インスタンスのトークンに割りあてられたカラーは，最終インスタンスが終るまで解放されない。しかも，テールリカージョンはループのかわりに用いられることが多いので，そのインスタンスの数は膨大になることが多い。このことは多量の色を準備する必要性があることを示しており，それだけハードウェアが複雑になることを意味している。また，ループでは，ループカウントを用い，プログラムの最小単位であるブロックとループそれぞれに色づけを行い，システムユニーク色づけによってなされることが多い［3］。そのためトークン色づけのための膨大な色を準備しなければならなくなり，ハードウェアが複雑になってしまう。

ここでは図8の破線で表わされたようにトークンを戻すテールリカージョンで，各インスタンス内でのトークンの色づけに順序性をもたす方法をループリカージョン（loop_recursion）と呼ぶことにし，この方法でループ処理とテールリカージョンを統一して取り扱うことにする。リカージョンでは，インスタンスの数が膨大になるのが一般的であるので，たとえば，インスタンスｉからインスタンスｉ－１への戻り手続を省略することは，大幅なスピードアップをもたらすと考えられる。また，ループリカージョンは呼出しという形で行われるので，ループ処理のようにサイクリックな処理がプログラム中に組みこまれることがなく，プログ

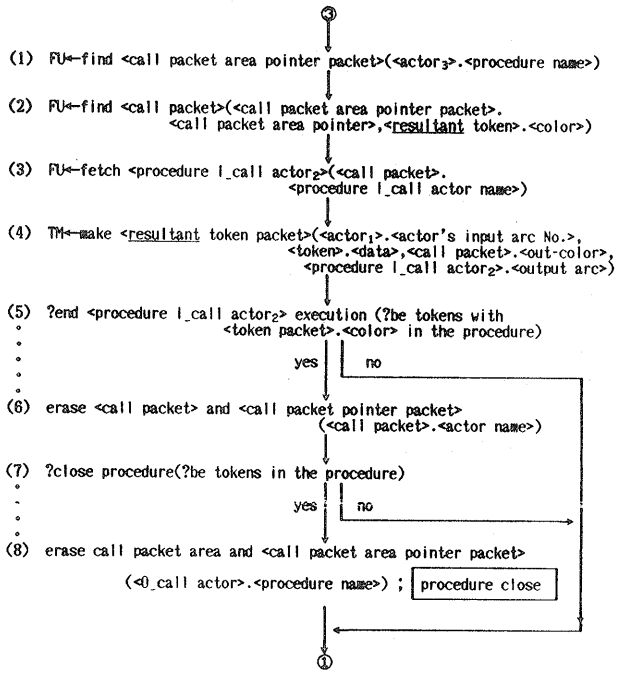

図7　　トークン出力アルゴリズム

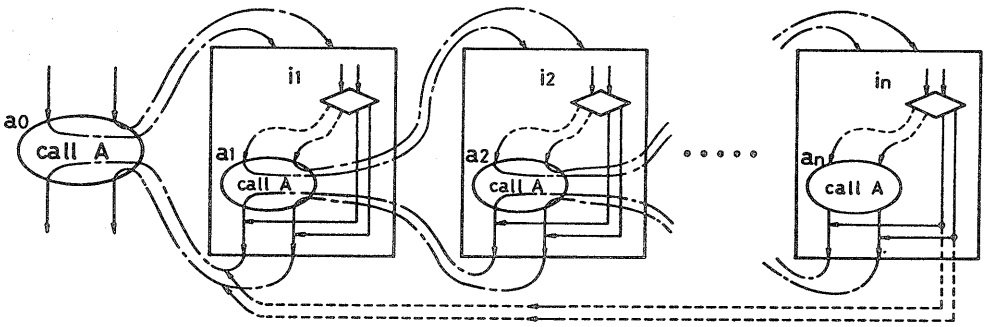

図8　　テールリカージョンによる処理

ラムの良解性がよくなる。

また実際のリカージョンでは，インスタンスはi_1からi_2，i_3とアクティブになっていき，インスタンスi_nがアクティブになるまえに，インスタンスがi_1からi_2・・・と実質的にノンアクティブになっていくことが普通である。このことはループの場合も同様である。そのため，たとえ膨大な繰り返し処理によって多量のインスタンスが作られるとしても，常時実質的にアクティブなインスタンスはその一部である。したがって，ノンアクティブになったインスタンスのカラーを再利用すれば，色の準備数を大幅に削減できる。ループリカージョンではこの性質がそのまま受けつがれており，しかも，図8のアクタa_0のように，最初は呼出し命令により呼出されるので，ループリカージョン内で，他のプロシジャとは方式の異なった色づけが可能である。

5．ループリカージョン

ループリカージョンは次のような構成である。

<loop_recursion>::=<loop_recursion name>
<attribute><loop_recursion body>
<loop_recursion R_call actor>
<loop_recursion 0_actor>
<loop_recursion R_call actor>::=
<actor name><loop-recursion R-call instruction>[<output arc>]
<loop_recursion 0_actor>::=<actor name><loop-recursion 0-call instruction><loop_recursion name>

<loop_recursion R_call actor> は図8のアクタa_1，a_2，・・・，a_nである。また，<loop_recursion 0_actor>は，同図インスタンスi_nのトークン出力を司るアクタである。<loop_recursion>は<loop_recursion l_call actor> （図8のa_0）により呼ばれる。<loop_recursion l_call actor> と<loop_recursion 0_actor>は，それぞれ<procedure l_call actor>と<procedure 0_actor> と同じである。

ループリカージョンでは，<loop_recursion l_call actor> 発火のとき，<basic color> に対して色づけがなされ，<loop_recursion R_call actor> のとき<sub_color> に対して色づけがなされる。<sub_color> の色づけは，順序性を持たして行われるので，<sub_color> をカウンタとしてカウントをインクリメントするような形で行なわれる。トークンの戻り場所は1ヶ所でよいので，コールパケットは，<loop_recursion l_call actor> の発火のときのみ作られる。<loop_recursion l_call actor> の発火アルゴリズムは，トークンの<basic color> のみに色づけを行い，<loop_recursion 0_actor>の発火では，トークンの<basic col

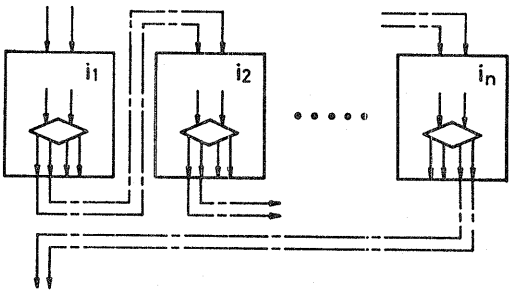

図9　ループ処理のインスタンス

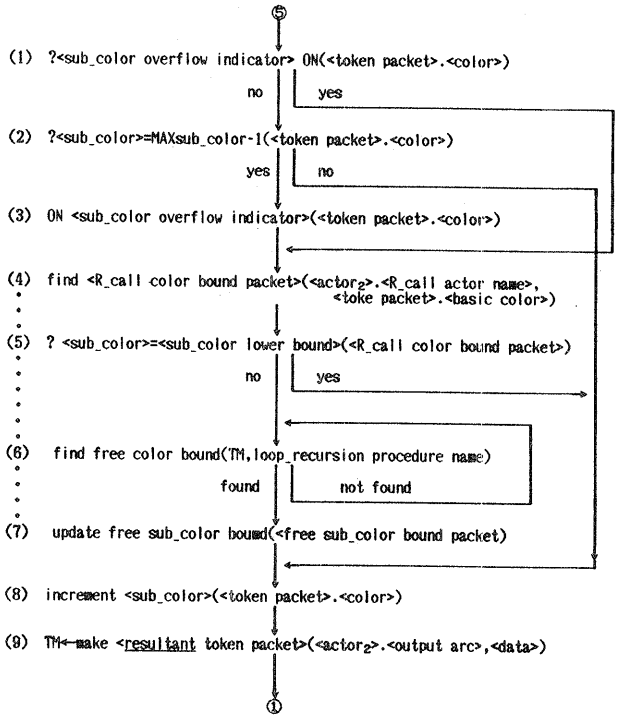

図10　<loop_recursion R_call actor>の実行アルゴリズム

or> をもとにコールパケットが捜されることを除けば，プロシジャコールのときと全く同じである。

<loop_recursion R_call actor> を発火したとき，使用中の色を管理するために<R_call color bound packet>が作られる。

```
<R_call color bound packet>::=
    <loop-recursion R-call actor name>
    <basic color><nil><sub_color lower bound>
    <sub_color uper bound><call packet area
    pointer>
```

アクティブなインスタンスにより使われている<sub_color> は<sub_color lower bound><sub_color uper bound> によって表わされる間の色である。<R_call color bound packet> は，<color> の<sub_color overflow indicator>がＯＮのときのみ参照され，ＯＦＦのときは，参照されない。ＯＦＦのときは，ただ，<sub_color> がインクリメントされるだけである。したがって，準備された<sub_color> の数だけで間に合う繰り返しの場合には，オーバヘッドが少なく高速実行が可能である。

<loop_recursion R_call actor>の実行アルゴリズムは図１０のようになる。

このアルゴリズムでは，<sub_color> の使用状況を調べ（（１），（２）），もし，オーバフローしないのならば，トークンパケットの<sub_color> をインクリメントして出力する（（８），（９））。もし，オーバフローしているならば<R_call color bound packet> から色をみつけ（（４），（５）），もし，<R_call color bound packet> 内でみつからなかった場合は，この<R_call color bound packet> が作られたとき以降解放された色がないかどうかをみて（（５），（６）），あれば，<R_call color bound packet> の領域を更新する（（７））。

６．むすび

以上，プロシジャコールとループリカージョンの実現方法を提案した。並列処理では，すべての処理が並列処理可能であることが望ましいが，システムユニーク色づけでは色づけ処理が集中化し，そこの並列度が減ってボトルネックになる可能性がある。本論文のプロシジャユニーク色づけでは，色づけが分散化されるので，このボトルネックを解消できる。また，本色づけはプロシジャユニーク故，異なったプロシジャ間では同じ色を用いることができ，しかも，プロシジャを区別するためのプロシジャネームのかわりに，データフロー処理のようなパケット通信形処理に必要なアクタネームをそのかわりに用いることができるので，色づけのためのビット数が減り，それだけハードウェア量が減少する。

ループリカージョンでは，ループとリカージョンを統一的に取り扱うことにより，サイクリックな処理をプログラム上から消すことができるし，ループリカージョンのオーバヘッドが，プロシジャコールより少ないため，ループ処理をリカージョンで代用するよりも高速に実行できる。しかも，一度使用した色を再利用できることにより，色の数を少なくできるので，この面でもハードウェア量の減少が図られる。

ここに述べられたプロシジャコール方法はすでに試作されたデータフローコンピュータにインプリメントされている［５］。また，簡単な理論計算によると本ループリカージョンは，単なるリカージョンより２～３倍高スピードであることも報告されている［６］。

文　献

［１］ Plas, A.D., Comte, D., Gelly, O., and Syre, C.,"LAU system architecture: Paralell data-driven processor based single assignments,"IEEE Proc. 1976 Int' Conf. on Parallel Processings, August 1976.

［２］ Dennis, J.B.,"First version of a data flow procedure language,"MIT Computation Group Memo, MIT/LCS/TM-61, May 1975.

［３］ Arvind, and Gostelow, K.P.,"The U interpriter,"COMPUTER, Vol.15, No.2, Februaly 1982.

［４］ 曾和，ラモス，”データフローコンピュータＤＦＮＤＲにおけるプロシジャのインプリメンテーション，”信学会電子計算機研究会，ＥＣ８３－１６，１９８３年７月.

［５］ Sowa, M., Ramos, F.D., and Murata, T.," Construction and structure of prototype dataflow computer DFNDR-1 and subroutine implementation," Proceedings of IEEE First China International Conference on Computers and Apprications, pp.490-497, June 1984.

［６］ 菊池，”データフローコンピュータにおけるループ及びプロシジャコールに関する研究，”群馬大学修士論文，昭和６０年３月.