プログラミング言語 14－5
（１９８７．１２．１８）

ImPP用高級言語Stream

太田　誠
日本電気(株) C&C共通ソフトウェア開発本部

本報告では、現在開発を行っているStream言語の設計方針、言語仕様、コンパイル技法について述べる。Stream言語は、当社で開発した画像処理用データフロープロセッサμPD7281(Image Pipelined Processor : ImPP)のための高級言語であり、ImPPに於けるパイプライン型処理やデータ駆動型処理との親和性を考慮して、ストリーム及び多値という概念を基本に設計を行った。また、コンパイル技法に関しては、データフローグラフ上でのトークンの流れ方をとらえるためにクラスタートークン法を提案し、この方法に基づくデータフローグラフの巨視化が同期制御や流量制御などの最適化に有効であることを示した。

"A high level language Stream designed for ImPP" (in Japanese)

by Makoto OHTA (C&C Common Software Development Laboratory, NEC Corporation, 4-14-22 Shibaura, Minato-ku, Tokyo, 108, Japan)

In this paper we discuss the specifications and compile technique of Stream language which we are developing as a high level language for μPD7281 (Image Pipelined Processor : ImPP). ImPP is a dataflow processor for image processing developed by NEC, and adopts pipeline architecture and data driven processing unit. Stream language is constructed on the basis of a concept, that is, "stream data and multiple value". It is suitable for ImPP's architecture. In relation to compile technique, we propose cluster-token method in order to analyze how tokens flow in the dataflow graph, and it is shown that macroscopic dataflow graph by this method contributes to optimization such as synchronization control and flow control.

1．はじめに

当社では、画像処理を主たる応用分野とするデータフロープロセッサμPD7281(Image Pipelined Processor:ImPP)を開発した。このプロセッサはホスト計算機の管理下で処理の一部を分担し、高速に実行するためのものである。ImPPでは高速性を実現するために、可変長パイプライン（サーキュラーパイプライン）とデータ駆動型演算ユニットを採用している[1]。ImPPはデータ駆動型演算ユニットを１個しか持っていないが、ImPPを複数個接続して用いた場合にはデータフローアーキテクチャとしての並列処理性が引き出せる。一方、ソフトウェア開発環境という観点から見ると、現時点で利用できるプログラミング言語がデータフローグラフをテキストで表現したアセンブリ言語のみ（ホスト計算機側はＣ言語）であること、データ駆動型プログラミングの思考が要求されること、ImPP特有の制約が強いことなどから、プログラム開発には大きな工数がかかる。そこで、当社ではImPP用の高級言語としてStream言語の開発を進めており[2]、現在はStreamコンパイラ・プロトタイプの評価検討を行っている。本稿では、Stream言語の設計、Streamコンパイラのコンパイル技法に関して報告する。

2．データフロープロセッサImPP

本章では後に述べるStream言語の設計やコンパイル技法に必要な、ImPPに関する前提事項について述べる。

2.1 パイプライン

ImPPのアーキテクチャは、最も基本的にはパイプラインアーキテクチャであると言える。例えば、プロセッサ内部は各ユニットがリング状のパイプラインを形成しており、内部を流れるデータ（トークン）はリングを一周する間に、データフローグラフ上で１つのノードに相当する処理を受ける（Fig.1 参照）。ImPPに於ける処理には色々な粒度でパイプライン的処理が存在しており、粒度の小さい方から列挙すると次のようになる。

(1) プロセッサ内部のユニットを単位としたパイプライン（上述）
(2) プロセッサ内の周回（データフローグラフのノード）を単位としたパイプライン
(3) 複数のImPPをリング状に接続してマルチプロセッサとして使う場合、ImPP１個を単位としたパイプライン

ImPPで速度性能を上げるためには、これらのパイプラインの効率を上げるようなプログラム、すなわち連続的にトークンを流せるようなプログラムを書くことが必要である。

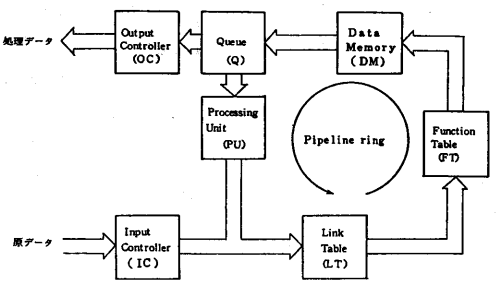

Fig.1 ImPP内部のブロック図

2.2 データ駆動原理

ImPPには、データ駆動型の演算ユニットが搭載されており、firing rule に従って処理が行なわれる。演算ユニットは１つのImPPに対して１個しか搭載されていないため、１個のImPPだけではデータフローアーキテクチャとしての並列処理性を引き出すことはできない。しかし、この「１つのImPPに対して演算ユニットが１個」という点は、後に述べるようにコンパイル技法を考える時に重要になる。

2.3 トークン生成命令（GE命令）

アセンブリ言語レベルでのImPPのプログラミングスタイルの特徴は「トークン生成とその消費」にあると言える。すなわち、通常繰返し処理としてループ変数を用いて記述するような処理は、ImPPではそのループ変数に相当するトークンを初めに生成しておき、それを消費部で使用するというスタイルに書くことが多い（Fig.2～Fig.3 参照）。そのため、ImPPには連続的にトークンを発生させる命令が用意されている。このトークン生成命令は、例えば外部メモリ（ImageMemory: IM）に連続的にアクセスするためのアドレス生成等に用いられ、うまく使えば前記のパイプライン効率向上に有効であるが、一方でプロセッサ内部のキュー（次の2.4節～2.5節参照）がオーバーフローする危険性もはらんでいる。

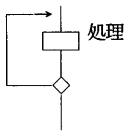

Fig.2 ループ的処理

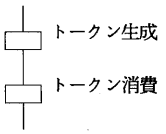

Fig.3 ImPP的処理

2.4 DMQ

ImPPには512ワードの内部メモリ（Data Memory: DM）があり、２項演算の待ち合わせキュー（DMQ）などに使われる。このDMQはサイズが16レベル以下と小さくオーバーフローしやすいので、次に述べるDQ、GQとともに流量制御が必要である。

2.5 DQ、GQ

ImPPの演算ユニットの前段には２種類のFIFO（DQとGQ）が入っている。GQは、トークン生成命令の実行を待つキューで深さは16レベル、DQはそれ以外の命令の実行を待つキューで深さは32レベルである。DQ、GQ両方にトークンが入っている時は、交互に先頭のトークンが取り出されて演算ユニットに入るが、DQに入っているトークンが８個以上になると、GQからの出力が禁止される。これは、DQにある程度トークンが入っている時に、トークン生成命令が実行されてDQがオーバーフローするのを回避しようとする、ハードウェアによる流量制御機構である。

2.6 連続出力ノード

トークン生成命令は、連続した数クロックにわたって演算ユニットからトークンを出力するが、それ以外の命令でも演算ユニットに於いてトークンを２個連続して出力するように指定することができる（Fig.4 参照）。これは、XX出力指定、XY出力指定と呼ばれるもので、データフローグラフのノードの実行順序を解析する時に重要な手掛かりとなる。

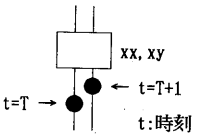

Fig.4 連続出力ノード

3. Stream言語の設計

第2章で述べたような特徴を持つImPPに対して、現在高級言語としてStream言語の開発を行なっている。本章では、まずStream言語の基本的な設計方針を述べ、次に言語仕様の中で特徴的な事項について説明する。

3.1 Stream言語の設計方針

Stream言語の設計に際しては、次のような基本方針に従って設計を行なった。

(1) ノイマン型計算機の標準的な言語スタイルを踏襲して、テキストイメージの言語とする。
(2) 基本的にはデータフロー言語の一般的な設計指針に従う。すなわち、データフローアーキテクチャと親和性がよいとされる関数型言語を採用し、副作用を極力排除し、単一代入規則を導入する[3]。
(3) アセンブリ言語レベルのデータフローグラフとの親和性を重視する。
(4) ImPPの主要な応用分野が画像処理であることを考慮する。

3.2 ストリームと多値

ImPPに於いて、データフローグラフの中を流れるトークンのイメージは2つの側面からとらえることができる。これをFig.5 で説明すると次のようになる。

(a) トークンa1、a2、a3のように同じ場所を次々にトークンが流れていくという側面。これをストリーム〈言語名のStreamはアルファベットを用いて記述し、データ構造のストリームはカタカナで記述する〉と言う。ストリームはパイプライン処理につながる概念である。
(b) トークンa1、b1、c1のように同じ世代のトークンが並行して流れていく側面。これを多値と言う。多値はデータ駆動による並列処理につながる概念である。

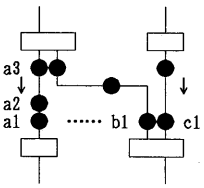

Fig.5 トークンの流れ

このように、「ストリームと多値」はデータフローグラフのトークンの流れに於いて本質的な概念である。そこで、テキストイメージの高級言語にも、「ストリームと多値」という概念を反映させるのが自然であると考えられる。

Stream言語では、ストリームを基本的なデータ構造として取り入れている。ストリームは、例えばKRCのような言語でも取り扱っており[4]、表現上、順番に並んだ一次元的データ列と見る限り、Stream言語のストリームと同じものである。しかし、ストリームをどうインプリメントするかという点では、KRC言語とStream言語は全く異なる。KRC言語では、ストリームはメモリ上に保持されており、ストリームの任意の要素は常時参照可能である。一方Stream言語に於いては、ストリームという概念が出て来た背景がパイプライン処理という点にあるため、ストリームはImPPの内部を順序関係を保持しつつ実際に流れるトークン列として実現される。

Stream言語に於けるストリームの表現は次の4種類がある。

(1) 列挙型
全要素を陽に書く方法。
ex) {1,3,5,7} → 1,3,5,7

(2) 個数指定型
等差数列に対して、初項、項数、増分を記述する方法。
ex) {1;4;2} → 1,3,5,7

(3) 条件指定型
変数を使って、初項、生成条件、変数値更新規則を記述する方法。
ex) {i=1;i<8;i=i+2} → 1,3,5,7

(4) 集合演算型
1つの名前でストリームを表現しておき、後で(1)～(3)の方法により定義する方法。
ex) {a:a={1;4;2}} → 1,3,5,7
この方法を入れ子で用いると、より複雑なストリームを表現することができる。
ex) {{a;4;1}:a={3;2;0}} → 3,4,5,6,3,4,5,6

またStream言語では、関数への入出力を含め処理の全過程に於いて、多値として式評価を行なうような記述の仕方をする。基本的に多値は複数の式を',' で区切って表わす。例えば、入力x,y に対して加算、減算、積算を施した3つの演算結果を評価値として返す関数はFig.6 のように書かれる。

```
func-3op(x,y:int) =
            x+y, x-y, x*y
end
```

Fig.6 多値の記法

この',' で区切って並べる表現法は、一見すると複数の評価結果を同期をとって返すように見えるが、ImPP内の実際のトークンは firing rule に従って動作する。すなわち、トークンの流路を複数個並列に用意するということを表わすにすぎない。3.3節で述べるlet式、if式、for式はその評価結果として多値を返す。例えば、ローカル名aを定義し、2aとa+1を求めるプログラムはFig.7 のようになる。

```
let a=P*P+Q*Q+R*R
in
    2*a, a+1
end
```
Fig. 7 多値式let

3.3 基本的な制御構造

Stream言語の制御構造は簡潔であり、let式、if式、for式の3つがある。let式は

let A部 in B部 end

の形を取り、A部でローカル名に値を結合し、B部でその名前を用いて多値式の評価を行なう。if式は

if A部 then B部 else C部 end

の形を取り、A部の条件判定の真偽値に従ってB部またはC部の多値式の評価を行なう。for式は

```
for ローカル名 in ストリーム式
    [ var A部 ]
    [ endval B部 ]
do
    C部
end
```

の形を取り、最も基本的にはストリーム式で表わされるストリームの要素を先頭から順次取り出してローカル名に結合し、C部でその名前を用いて多値式の評価を並列的に行なう。例えば、Fig. 8 はFig. 9 のような処理を表わす。またループ変数を用いて、逐次的な繰り返し処理を表わすことも可能である。例えば、Fig. 10はFig. 11のような処理を表わす。ループ変数の定義は「var A部」で行ない、値の更新はnext式（next～end）で行なう。Stream言語に於いて単一代入規則が破られるのは、このnext式と条件指定型のストリーム式（3.2節参照）のみである。for式に関連してそのほか、for式の評価の途中で、ある多値を持ってfor式を抜けるbreak式や、ストリームの全ての要素に対する評価が終了した時点で、同期用のデータを発生するendval式（endval B部）などが用意されている。

```
for x:int in {1;N;1}
do
    f(x)
end
```
Fig. 8 基本的for式

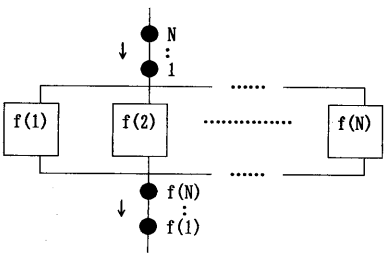

Fig. 9 基本的for式（模式図）

```
for x:int in {1;N;1} var s:int=0
do
    next s=s+x in f(s) end
end
```
Fig. 10 逐次的for式

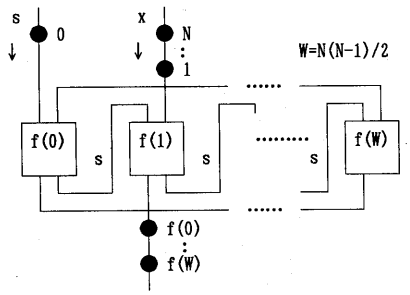

Fig. 11 逐次的for式（模式図）

3.4 C言語との比較

Stream言語に於いても、2.3節に述べたような「トークン生成とその消費」というプログラミングスタイルは踏襲されている。Fig. 12とFig. 13は同じ内容の処理をC言語とStream言語で書いた例である。

```
void
func(X,Y)
int X,Y;
{
    int ax,ay;
    for (ay=0; ay<Y; ++ay) {
        for (ax=0; ax<X; ++ax) {
            func1(ax,ay);
            func2(ax,ay);
            func3(ax,ay);
        }
    }
}
```
Fig. 12 C言語による記述

```
func(X,Y:int) =
    let Ax:int={{p;X;1}:p={0;Y;0}}
        Ay:int={{q;X;0}:q={0;Y;1}}
    in
        for ax,ay:int in Ax,Ay
        do
            let _ =func1(ax,ay)
                _ =func2(ax,ay)
                _ =func3(ax,ay)
            in
                _
            end
        end
    end
end
```
Fig. 13 Stream言語による記述

Fig.12とFig.13をデータフローグラフとして模式的に表わすと、Fig.14及びFig.15のようになる。これを見ると、並列評価可能な繰返し処理に於ても、C言語では逐次的なループ処理のスタイルになるのに対し、Stream言語ではストリームデータに対するベクトル処理というImPP本来のプログラミングスタイルになっていることがわかる。その意味でStream言語はImPPのデータフローグラフと親和性が良いと言える。

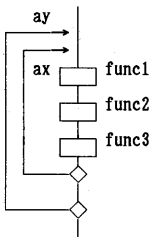

Fig.14 C的スタイル

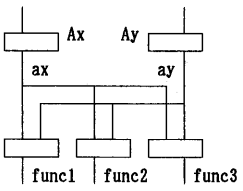

Fig.15 Stream的スタイル

3.5 データ型

ImPPではハードウェアとして浮動小数点演算機能を持っていない。この事を考慮して、Stream言語として用意している基本的なデータ型は、整数型（16ビット）、2倍長整数型（32ビット）、整数型の要素のみからなるストリーム、2倍長整数型の要素のみからなるストリームの4種類である。また、基本的浮動小数点演算機能はライブラリ関数として提供することにしている。

3.6 ボトム

Stream言語では名前の現われる文脈でボトム '_' を用いることができる。ボトムの意味は「無」である。すなわち、次のような性質を持つ。

(1) 名前を定義する式の左辺に '_' が現われると、右辺の評価結果を捨てる。
ex)
```
_ = x*y
```
(2) 式中に現われると、その式全体の値を '_' 自身にする。
(3) ストリーム中の要素として '_' が現われると、それは要素として無視される。
ex)
```
A:int = for a:int in {1,2,3,4,5,6}
        do
            if a%2 == 0 then a else _ end
        end
```
（この例では、A は {2,4,6} となる。）

ボトムは不要なデータを消滅させ、ImPP内部の動作を効率化するのに有効である。

4. Stream言語のコンパイル技法

Fig.16はStreamコンパイラ完成後のImPPの言語処理系を表わしている。Streamコンパイラはソースプログラムを中間表現に1度翻訳し、流量制御や負荷分散などの最適化の過程を経て、ImPPのアセンブリ言語プログラムを出力する。以下、Streamコンパイラに於けるコンパイル技法について述べていく。

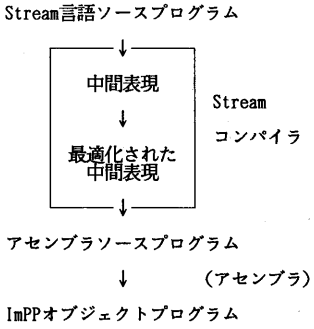

Fig.16 ImPP言語処理系

4.1 クラスタートークン法による実行順序の解析

本節では、データフローグラフ中のトークンの流れ方を調べる手法として、クラスタートークン法を提案する。これは同期制御や流量制御に応用される。

まず非トークン生成命令のみからなるデータフローグラフについて考える。この場合、全てのトークンはDQを経て演算ユニットに入力する。Fig.17のような2出力ノードに於いては、a2のアークとa3のアークへは、この順にトークンが連続して出力される。この性質とトークンがImPP内を周回する道筋が1本しかないことを考慮すると、各ノードの実行順序はデータフローグラフの静的な解析から完全に定めることができる。例えば、Fig.18でa1からトークンが1個入力したとすると、以後各ノードが実行される順序は、

n1→n2→n3→n5→n4→n6→n7→n8　　(★)

となることがわかる。

この解析をシステマティックに行なうためにクラスタートークンという概念を導入する。Fig.17でノードn2とn3は連続して実行されるため、これを新しい1つの巨視化ノードと見る。そして巨視化ノードへの入力トークンは、n2へ入力するトークンとn3へ入力するトークンがクラスター化しているものと見なす。これをクラスタートークンと呼ぶが、巨視化ノードへ入力するとクラスターが分解され、各トークンがしかるべきノードへしかるべき順番で入力する。この様子を示したのがFig.19である。巨視化されたデータフローグラフではノードを○で表わし、また、巨視化ノードを{n2,n3}のように表わす（n2→n3の順に実行されることを意味する）。巨視化ノードの重要な性質として、複数本の出力（例えばFig.19のa4,a5）がある時は、通常の2出力ノードと同じように左側のアークから順にトークンが出力される。従って、Fig.17を巨視化してFig.19を得たのと同様の操作を繰返し施すことができる。Fig.18に対してデータフローグラフを巨視化していく過程を示したのがFig.20～Fig.28である。

Fig.28を見るとノードの実行順序が(★)のようになることがただちにわかる。一般的には、Fig.29のようなトークンが分流するノードやFig.30のようなトークンが消滅するノードもあるが、これらはクラスタートークンの構成要素に対して重み付けをすることで対処できる。例えば、Fig.29を巨視化するとFig.31のようになる。ここで、2n2/3+φ/3 はクラスタートークンが3回入力するうち最初の2回はn2へ入力するトークンが存在し、あとの1回はそうようなトークンが存在しないことを表わす。2φ/3+n3/3 も同様である。

一般に、非トークン生成命令のみから構成される、ループが存在しない1入力のデータフローグラフは、Fig.28のように必ず直線的なグラフに巨視化される。もしループが存在すれば、巨視化後もループ構造は残る。また、入力が複数個あるデータフローグラフは各入力のタイミングによってその後のトークンの流れ方が変ってくるので、全体を1つのグラフに巨視化することはできない。例えばFig.32を巨視化すると、T1+M1<T2+M2の時はFig.33になり、T1+M1)T2+M2 の時はFig.34になる。ここでTiはi番目の入力口からの相対入力時刻をデータフローグラフの段数を単位として表わしたもの、またMiはi番目の入力口から合流部までの段数（Fig.32ではM1=3,M2=2）である。

これまで取り扱ってきたのは全ノードが非トークン生成命令の場合であったが、次にトークン生成命令のノードも存在するデータフローグラフについて述べる。この場合はDQとGQ両方から演算ユニットにトークンが入力するため、上記の複数入力の場合と同様に、1つのグラフに巨視化することはできない。しかし、いくつかの部分巨視化グラフの集合体に変換することはできる（Fig.35参照）。すなわち、部分巨視化グラフ間の実行順序は静的な解析からはわからないが、部分巨視化グラフ内のノードの実行順序は定まる。このようにデータフローグラフを巨視化することにより、コンパイル時の同期制御解析や流量制御解析等が考えやすくなる。

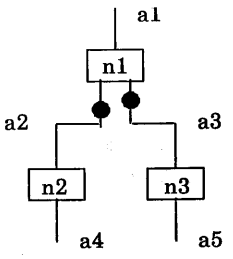

Fig.17 連続出力ノード

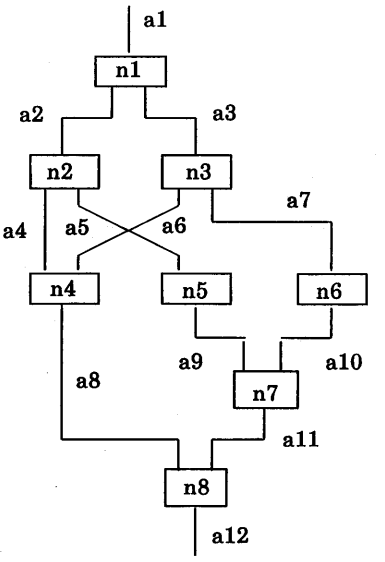

Fig.18 巨視化前

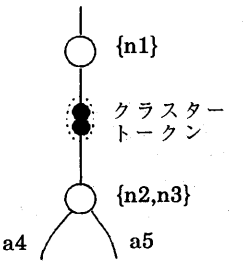

Fig.19
クラスタートークン
と巨視化グラフ

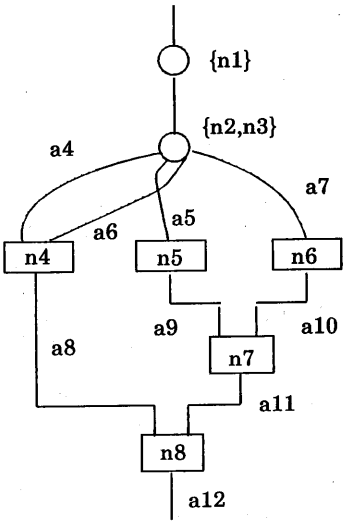

Fig.20 n2,n3を巨視化

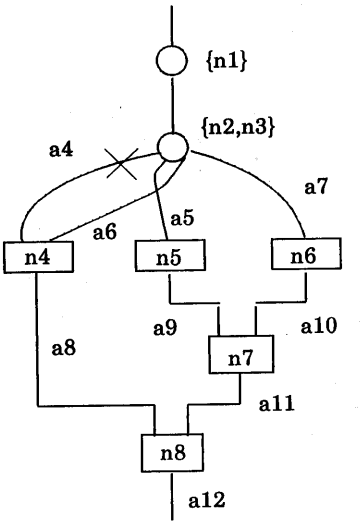

Fig.21 n4が発火するタイミングは
a6で決まるためa4を切断

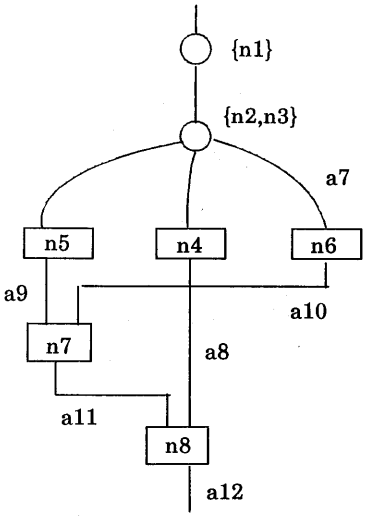

Fig.22 n4,n5,n6を順序化

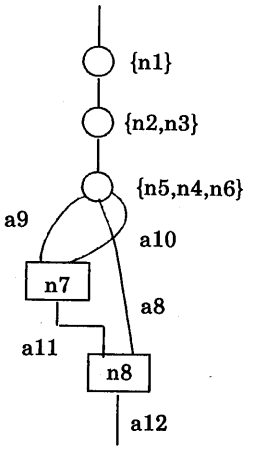

Fig.23 n5,n4,n6を巨視化

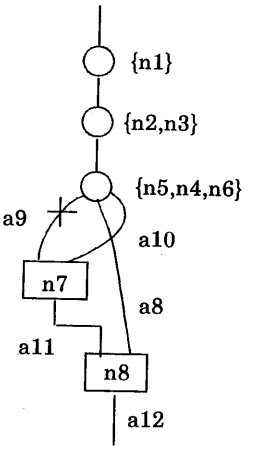

Fig.24 a9を切断

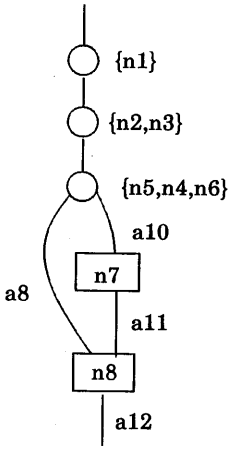

Fig.25 n7,n8を順序化

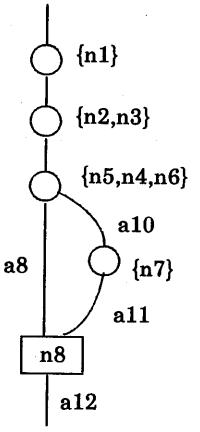

Fig.26 n7を巨視化

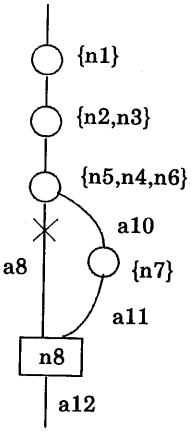

Fig.27
a8を切断

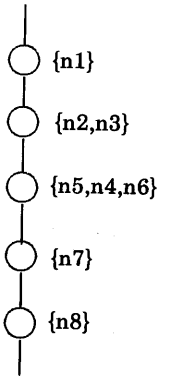

Fig.28
n8を巨視化

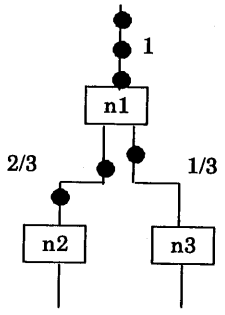

Fig.29 分流

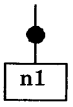

Fig.30 消滅

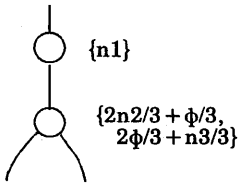

Fig.31 重みつき巨視化

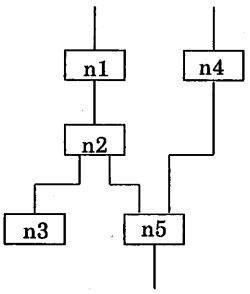

Fig.32 巨視化前

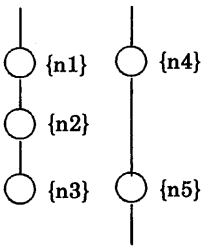

Fig.33
(T1＋M1＜T2＋M2)

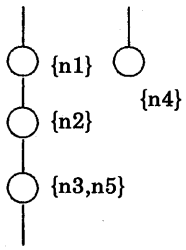

Fig.34
(T1＋M1＞T2＋M2)

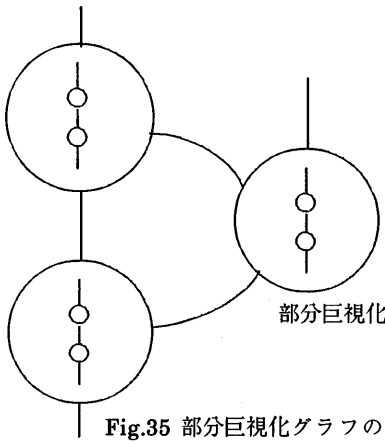

Fig.35 部分巨視化グラフの集合

4.2 同期制御

まず、DM（内部メモリ）に対しデータを読み書きする場合の同期について考える。例えば、非トークン生成命令のみからなるFig.36のようなデータフローグラフに於いて、クラスタートークン法によりグラフの巨視化を行なうとFig.37のようになる。Fig.37によれば、wrcycs（DMへの書き込み）が実行されてからrdcycs（DMからの読み出し）が実行されることが容易にわかる。今仮に、Fig.36で wrcycsをしてから rdcycsしなければならないとすると、Fig.37に於けるノードの実行順序の解析から、wrcycsのノードと rdcycsのノードの間に同期制御は必要ないことになる。逆に、Fig.36で rdcycs をしてからwrcycsをしなければならないとすると、2つのノードの間に同期制御が不可欠となる。すなわち、Fig.38に示すように待ち合わせノードqueueを挿入しなければならない。また、ストリームデータが入力される場合は、1つの入力トークンに対する読み書きの順序関係だけでなく、次に入力するトークンの読み書きタイミングとの順序関係も考えなければならないが、基本的には同様に解析することができる。

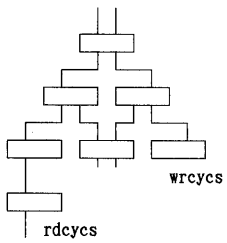

Fig.36 巨視化前

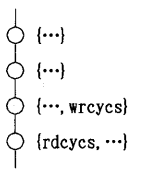

Fig.37 巨視化後

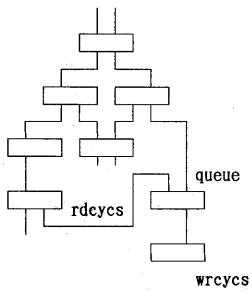

Fig.38
同期用queueの
挿入

次に分岐・合流機構の同期について考える。Fig.39のように条件判定によって一旦分岐したストリームが合流する場合、一般的には合流後の順序性は保証されない。今、Fig.39を巨視化するとFig.40が得られる。分岐と合流の間にある巨視化ノードの段数をそれぞれM1、M2、入力トークンの時間間隔をノードの段数で表わしてT、とすると

$$T \rangle |M1 - M2| \qquad \langle 1 \rangle$$

が成立すれば、特に同期のための機構をデータフローグラフに付加しないでも合流後の順序性は保証される。さらにFig.40を一般化した巨視化データフローグラフFig.41では、〈1〉式の代りに

$$T \rangle \max\{Mi + Bi\} - \min\{Mi + Bi\} \qquad \langle 2 \rangle$$

が成立すれば、同様に合流後の順序性が保証される。ここに、Mi は i 番目の分岐処理部のノードの段数を表わし、Bi は i 番目の分岐処理部に至るには幾つの分岐ノードを通過しなければならないかを表わしている。〈1〉あるいは〈2〉が成立しない場合は、何らかの同期機構を持ち込まなければならないが、この点については、4.4節の「if式のコンパイル方法」のところで詳しく述べる。

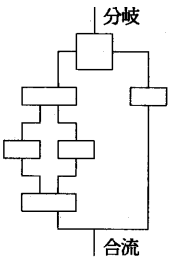

Fig.39 巨視化前

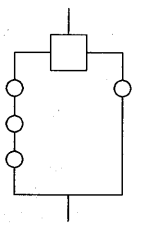

Fig.40 巨視化後

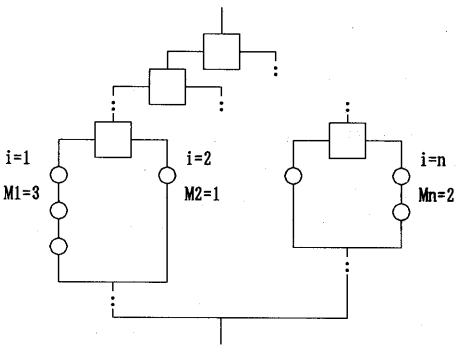

Fig.41 一般的な分岐・合流

以上のように、処理の順序性が保証される場合には同期機構の付加を行なわない。これは、オブジェクトサイズの縮小と速度性能の向上に寄与すると考えられる。

4.3 流量制御

DMQオーバーフロー防止のための流量制御に関しては4.4節の(3)で述べる。そこで、すでにDMQに関して流量制御がされているとする。すなわち、データフローグラフの任意の場所に待ち合わせ命令（queue命令）を挿入できるとする。4.1節で述べたように、クラスタートークン法によって任意のデータフローグラフは部分巨視化グラフの集合に変換することができる。部分巨視化グラフ i の最大トークン流量を、トークン生成命令、非トークン生成命令それぞれについてFGi、FDiとすると、GQ、DQがオーバーフローしないための条件は

$$\Sigma FGi \leqq 16 \qquad \langle 3 \rangle$$

$$\Sigma FDi \leqq 32 \qquad \langle 4 \rangle$$

となる。ここで、各部分巨視化グラフについて入力トークンのタイミングがわかれば、FGi、FDiは静的に求まる。従って逆に、〈3〉、〈4〉を満たすように、queue命令を挿入して各部分巨視化グラフ間で同期を取れば、GQ、DQがオーバーフローしないようにすることができる。

4.4 基本的な制御構造のコンパイル方法

本節では３つの基本的制御構造のコンパイル方法を簡単に述べる。

(1) let式のコンパイル方法

let式のコンパイルは、letとinの間で定義されているローカル名の値を計算するデータフローグラフを作り、その値をinとendの間で参照している場所へ分配すればよい。もし、Fig.42のx のようにfor式の中で参照されている場合はforの処理に於いて定数として振る舞うので、forの処理に入る前にDMに書き込んでおき、forの中ではそれを読み出して用いる。

```
let x:int = A + B
    U:int = {0; N; 2}
in
    for u:int in U
    do
        x*u, x*u*u
    end
end
```

Fig.42 for中でのローカル名参照

(2) if式のコンパイル方法

if式をコンパイルしようとする時に問題になるのは、4.2節で述べた合流時の順序性保証である。4.2節の〈2〉式が成立する場合は、if式はFig.43の模式図ようにコンパイルされる。これは、if式の最もコンパクトなデータフローグラフである。しかしながら、一般的にはFig.44のようにフィードバックをかけて、分岐・合流間に１世代のトークンしか存在しないようにしなければならない。この方式ではパイプライン的動作が妨げられ、分岐・合流間の処理量が大きくなると速度性能の低下が著しくなる。そこで、第３のコンパイル方式としてフィードフォワードを用いたFig.45のようなコンパイル方式を考案した。この方式では、３つのバッファを用意する。１つはthen側の処理結果の格納（then側バッファ）に、１つはelse側の処理結果の格納（else側バッファ）に、そして残りの１つは「次の合流時にthen側とelse側のどちらからトークンを出力させるべきか」という合流情報（例えば0ならthen側から、１ならelse側からの出力を意味すると定める）の格納（合流情報バッファ）に用いる。分岐部に入力したトークンは、then側またはelse側に分岐するだけでなく、その時どちら側に分岐したかという情報、すなわち合流情報を合流情報バッファに送る。then側およびelse側の処理は、各種キューがオーバーフローしない範囲内でパイプライン的に行なえる。その結果は非同期にthen側バッファやelse側バッファに送られ、合流情報に従って合流され出力される。

この「合流情報がトークンを読み出す」という動作を考えてみると、それは要求駆動的であると言える。しかし、ImPPの命令セットはあくまでデータ駆動として考えられているため、ソフトウェア的に分岐部、合流部の機能を実現しなければならない。従って、そこにはある程度のオーバーヘッドが生じ、パイプライン的に動作しない時には、かえってFig.44より処理性能が悪くなる。シミュレータ実験によると、2世代以上ずつのトークンをif部で処理すれば、Fig.45の方がFig.44より性能が良くなることがわかっている。またFig.45は、then部あるいはelse部に於いて、ボトムでbreakしたり、新たにストリーム生成したりする場合には対処できない。Fig.43～Fig.45はどれが良いというものではなく、状況に応じて使い分けなければならない。

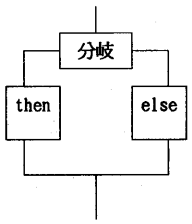

Fig.43 基本的if式

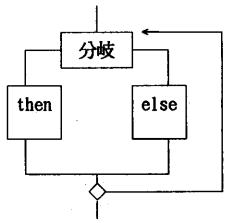

Fig.44 feedbackによるif式

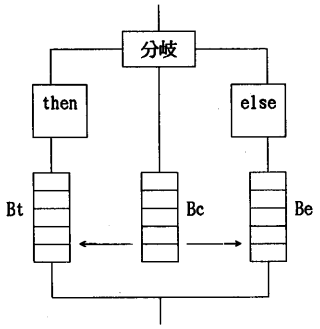

Fig.45 feedforwardによるif式

(3) for式のコンパイル方法

for式は基本的にストリーム生成部から要素を取り出し、for式の評価部に供給するのが役目である。従って、for式のコンパイルで注意しなければならないことは、供給過剰による各種キューのオーバーフローが起らないように流量制御をすることである。DMQオーバーフローには、次のように対処している。ストリーム生成命令が一度に生成できるトークンの数GSは16以下であり、待ち合わせキューのサイズQSも16まで取れる。従って、次の2つの条件を満たすようにすれば、1重のforの場合にはforの評価部でDMQがオーバーフローする可能性はない。

(a) GS=QSとする。
(b) 生成されたストリームデータが全て消費されてから次のストリームデータを生成する。

forが多重にネストされている場合は、1番内側のforには(a)、(b)を適用し、他のforには(a)の代りに次の(c)を適用する。

(c) GS=1 とする。 (Fig.46参照)

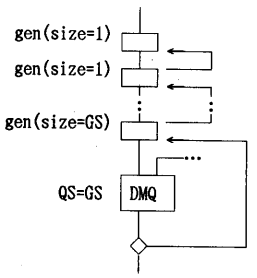

Fig.46 多重for

5. おわりに

本論文では、データフロープロセッサImPP用高級言語Streamについて、言語仕様面とコンパイル技法面の双方から報告を行なった。まず、言語の設計を行なう時に考えなければならないImPPの特徴について簡単に述べた。次に言語仕様面について述べ、その中で「ストリームと多値」という概念がImPPのプログラミングスタイルと親和性が良いことを説明した。さらに、コンパイル技法に関してクラスタートークン法を提案し、この方法によるデータフローグラフの巨視化が同期制御や流量制御に有効であることを示した。現在はStreamコンパイラ・プロトタイプの評価を行なっているが、本稿で述べたことをさらに発展させ、また、非常に重要な負荷分散の問題に取り組みたいと考えている。

6. 参考文献

[1] μPD7281ユーザーズマニュアル
[2] 太田, 佐治, "ImPP用高水準データフロー言語Streamの設計", 第34回情処全大, 1U-1, pp.709-710, 1987.
[3] データフロー特集, IEEE Computer, Feb., 1982.
[4] Turner, D.A., "The semantic elegance of applicative languages", Proc. Symp. on Functional languages and computer architecture, pp.85-92, Oct., 1981.