計算機アーキテクチャ 38－3
（1980. 6. 25）

汎用データフローマシンの実現における問題点について

樋口哲野（慶応大学 工学部）
内田俊一（電子技術総合研究所）

1. はじめに

スループットの増大を並列処理によって図ろうとする計算機システムの満たすべき要件は、単にハードウェアの性能だけにあるのではない。イリアックIVやベクトルプロセッサのようにどちらかというとハードウェア先行型で、ときとして並列性の明示をユーザに求めるようなアプローチは望ましくない。ユーザはアーキテクチャにとらわれずにプログラムを書くことができ，それでいてマシンの潜在的能力を生かせることが重要である。こういった形態を実現するためには並列処理タスクを機械的に検出する手段が計算機システムに組み込まれる必要がある。この機械的な並列性検出を行なうためには，その前提として，並列処理を行なうべく分割された個々のサブタスクが明確な関数的性質を保持していることが大切である。関数的性質とは，そのタスクに対する入力は，その出力の決定にのみ関与し，且また，そのタスクの出力は入力以外の何ものの影響も受けないということが明示されていることである。この性質が満たされずに複数のタスクが並列に実行されると，時間に依存した誤りを生じ，結果の正当性は保証されなくなる。この関数的性質の並列処理における重要性は、今日までの汎用マルチプロセッサの不成功を逆説的に裏付けているとも言える。つまりフォンノイマン型プログラムでは関数的性質を保持したタスクへの分割が一般にむずかしいために，本格的なマルチプロセッサの構築が困難となっている。この原因は，関数的性質を明確にするためには副作用の除去が必須の条件であるにもかかわらず、フォンノイマンモデルが副作用の意図的な利用によって計算を行なっているためである。フォンノイマンモデルはメモリセルへの逐次的な参照と代入を計算機構の基本にすえているために副作用の根本的な除去は困難である。

しかしながらフォンノイマン型言語においても，プログラムに関数的性質を与えることは可能である。プログラム中に現れる変数は高々一回しか定義されないという単一割当規則に基づいて記述されたプログラムは，シンタックスはどうであれ，その性質は関数的となる。1968年にテスラによって提案されたこの概念によって，ステートメント1つ1つは入出力を持つ関数とみなすことができ，プログラムの実行は、ステートメントの並ぶ順序には依存しなくなる。つまりステートメント相互間の入力，出力の関係が計算を進行させていくという解釈が成り立つ。1つの出力が複数の入力の源となっている場合には，明示的な並列性の制御情報を伴わなくても，そのまま並列処理を表現する。ここで言う入力，出力とはオペランドレベルのデータに対応しており、実行のシーケンスの流れはまさにデータの流れと一致する。このようなモデルをサポートする計算機をデータフローマシンと呼ぶが，以上の議論から明らかなように，データフローマシンが走るまでには数多くの仕事がソフトウェアサイドでなされなくてはならないことがわかる。したがってデータフローモデルの精密化，言語のあり方などソフトウェアサイド

の研究はデータフローマシンの仕様決定に大きな影響を与えていると言える。

プログラムは単一割当規則の集合から成るという考え方は，次のような利点をもたらす。第1は並列性の明示をわざわざプログラマが行なわなくても，ステートメントの入出力，つまりデータの従属関係をシステムが解析することによって，機械的な並列性検出が行なえることである。この事実は先にも述べたようにマンマシンインタフェイスの高機能化を進める可能性がある。第2にソフトウェアの検証に有用である。つまり，プログラム全体が表明(assertion)の集合であると考えられるから，従来の帰納表明法(inductive assertion)のように，ある表明がどこで成り立つかを知るためにプログラムの状態遷移を精密に追う必要がない。

以上，2つの利点を取り上げてみたが，このうち第1に述べたことが果たして長所だけをもたらすのかどうかは一概に言えない。つまり，並列性の明示は避けられたものの，単一割当規則に基づいて書かれたプログラムは非手続き的となる。一般に言語が非手続き的になるということは，howからwhatへの転換ということで，記述レベルの高度化を示唆するが，データフロー言語におけるそれは決してそうとは言いきれない。プログラムのテキストから，そのふるまいを直観的に知ることはむずかしくなり，果たして本格的なプログラムが書けるのかという疑問が生ずる。並列性の機械的検出のために単一割当規則を強要され，それによってプログラマビリティが悪くなったのでは，マンマシンインタフェイスの向上にはつながらない。データフローの論文に示される例は，toyプログラムという感を免れず，プログラマビリティの問題については説得力に欠けている。

本稿の目的の第1は，本格的な応用プログラムにおけるデータフロー言語のプログラマビリティがどうなるのかを知ることにある。ここで言う本格的プログラムの条件は，まず，配列に対する処理を含んでいることである。データフローモデルでは配列に対する更新は許されず，論理的には処理のたびに新たな配列を生成しなくてはならない。しかしこの定義通りに処理を行なうとメモリへの通信量が大きくなるため，実際には配列を生成せず共有で済ませるメカニズムを導入することが一般に認められている。この機能(appendとselect)を生かして配列の処理をうまく記述できなくてはならない。また，本格的なプログラムの条件として，経過依存性(history-sensitive)の計算を記述できることがあげられる。しかしながら通常の関数型言語の枠組の中では，状態の保存(副作用)の考え方がないため，そういった処理を行なうためには，概念的な拡張が必要である。そのため本稿では，これについては扱わないことにする。

本稿の目的の第2は，種々提案されているデータフロー言語の中から，実際に必要かつ有用なプログラム構造は何かを考察することにある。これは筆者らが計画しているデータフローマシンシミュレータの作成の準備としてである。筆者らは最終的にはマシンの構築を目指しているが，アーキテクチャの研究のためには，特にメモリまわりについての統計のとれるシミュレータの必要性を感じている。なぜなら，見かけ上の副作用の除去を保証するストラクチャメモリに対してはかなりの通信量が予想され，それに対する定量的な解析なくしてはアーキテクチャの検討が行なえないからである。

この2つの目的のために，筆者らは

実際に大きなプログラムをデータフロー言語で書いてみることにした。応用プログラムとしてガウスの消去法と1次元FFTを選び，MITのデータフロー言語Idをたたき台にした言語(MIDと略す)によって記述した。そして並列実行の形態を机上で解析した。
2章ではIdの選択理由を，3章ではIdとそのベース(グラフ)言語について加えた拡張について述べ，4章では解析結果について考察する。

2. データフロー言語 — Idの選択

データの流れによるプログラムの実行形態を最も直観的な形で示せるものは，2次元のグラフ表現形式である。MITのDennisは70年代の初めに，グラフ表現に基づく言語を提案し，データフローの概念を形式化した。以後これを契機に各所でデータフローマシンの研究が活発化した。Dennisの言語はグラフ表現であるため，プログラマビリティの観点から，textualな高級言語の研究もハードウェアの研究と並行してすすめられてきたが，その根底にあるのは単一割当の概念である。つまり単一割当規則で書かれたプログラムは必要なデータがそろい次第実行可能であり、Dennisのグラフ言語の形式にきわめて容易に変換できるからである。これに基づき種々の高級言語が提案されているが，汎用データフローマシンシステムの一環として総合的な見地から開発されている言語は数少ない。現時点ではMITのVALとIdの2つが最も充実していると考えられる。VALはDennisとAckermanによって設計された高級言語であり，同じくMITのデータ抽象言語CLUの影響を受け、その結果厳密なタイプチェックや例外処理機能，豊富な組み込み関数など，データフロー言語としての完成度は最も高い。VALはコンパイルされて，先のDennisのグラフ言語と等価な機械語を生成すると考えられる。IdはもとUCIのArvindによって開発された言語であり、従来の数値計算に加えて，OS管理のためのリソースマネージャー，経過依存性の導入など，データフローに対する概念的な拡張を図っている。Idはコンパイルされると，Dennisのグラフ言語に類似した機械語を生成する。この「類似」という意味は，Dennisのそれに加えて，並列タスクを次々に発動するためのコードを含んでいることをさす。つまりIdのコンパイル結果はインタプリタに対する入力であり、Idをサポートするマシンはハードウェアのインタプリタとなる。これに対しVALのコード，つまりDennisの機械語はそのまま実行コードに対応する。つまりVALはコンパイラベースである。

今，図1のようなデータフローグラフがあるとする。Dennisの言語の約束ではAの位置，つまり，加算オペレータの出力端にあるデータが次の乗算オペレータによって吸収されていないうちは加算オペレータに対する入力は許可されない。したがってパイプライン処理によってxとyが次々に加算オペレータに送られてきたとしても，それらを加算オペレータの入力端上で待たせることはできない。この実行規則は，Activity Templateと呼ばれるデ

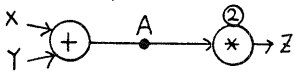

図1 データフローグラフ

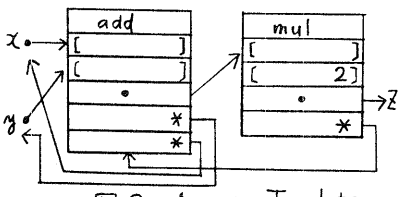

図2 Activity Template

ータフローマシンの機械語表現（図2）にもそのまま反映されている。つまりDennisのActivity Templateではオペランドの待ち合わせが実際のメモリの上で行なわれているために，xとyのデータが次々に送られてきたとしても同時に複数個待ち合わせることはできない。しかしながら，本来この加算という操作は内部状態を持たないリエントラントなコードである。xとyのデータが同時に複数個きても，加算器が十分にありさえすれば並列に実行できるはずである（このような並列性をここでは動的な並列性と呼ぶことにする）。したがって，Dennisの言語，及びテンプレートでは本来の並列性を生かすことはできない。加えて，入力を許可するためのAcknowledge信号（図2の＊）が要るために，全体としてトークンの量が倍になるという欠点もある。

これに対しIdでは，動的な並列性を許している。たとえ加算オペレータの入力端にトークンがたまっても並列実行させることができる。ただし，それら個々の並列タスクの関数的性質が保証されていることが前提である。つまり、ある時点における加算操作の結果は，必ず，ある特定された乗算オペレータによって受け取られねばならない。これを実現するために，Idでは，Dennisのデータトークン（値と行き先を持つ）にさらにタグを設け、これによって並列に起動されたリエントラントなコードを識別し，関数的性質を保証している。タグに関する操作を行なうものとして，D及びD⁻¹オペレータが設けられている。この操作をトークンラベリングと呼ぶこともある。その効果によって，単にオペレータに限らず，ループや関数も同時に並列に発動することができ，大きな利点となっている。VALの方がIdよりも言語としての完成度が高いと認めながらもIdをたたき台として選ぶ第1の理由はここにある。

第2の理由は，VALにおいて用意されている言語の機能が高級すぎることにある。実行時のエラーチェックの種々の機能，十数種の配列操作命令，例外処理などをはじめ，プログラムを書くための機能はきわめて充実している。しかし，アーキテクチャを考える立場からすると，高級な言語機能を基本にして考えることは，セマンティックギャップを生じさせ，望ましくない。高級な機能はよりプリミティブな機能からブートストラップできるものである。したがって，Idのような比較的簡潔な言語の方が望ましい。

Idを選ぶ第3の理由は，並列タスクの検出，及び実行制御機能にある。Idのように，リエントラントコードを前提とするものでは，オペランドの待ち合わせを，Dennisのテンプレートのように実際のメモリでやる必要はない。同一の行き先とタグを持つデータペアをパターンマッチングで知ればよい。そのために連想メモリ、あるいはハードウェアハッシングによる方法が提案されている。従来のプログラムカウンタに代わり、パターンマッチングによる実行制御機能が発展していけば，SAIL等人工知能向き言語における演えき機能のハード化や，あるいはPrologマシンの実現の可能性も出てくる。したがってトークンラベリング型の言語を前提にアーキテクチャの研究を行なうことは他の分野への影響を持つと考えられる。

3. Idについての拡張

計画しているシミュレータは数値計算を対象としているため，Idを採用するにあたっては，データベースの更新

など，経過依存性の処理を行なうために導入されたデータフローモニタ等は除外した。データ構造操作と通常の数値計算に必要なものに限った。採用したオペレータに対応してActivity Templateを作成し，それに基づいてデータフローマシンの実際の挙動に近いレベルでプログラムの実行を解析してみたが、その際Idのオペレータの定義が不明確なために実行を厳密に規定できない部分や，並列性が制限される部分が見出された。このため，それらに対して以下に述べるような拡張を図ることにした。

① 定数の取り扱い（Cオペレータの導入）

定数には，あるプロセスが起動されてから消滅するまでの間に1回しか使われないものと，くり返し使われるものとの2種類がある。前者の場合，トークンラベリング型言語をサポートするマシンでは，トークンをハッシュテーブルで待ち合わせることを利用して、そのプロセスが起動されたときに，定数となるトークンをハッシュテーブルにおいておく。しかし，くり返し用いられる場合には，ある演算において，その定数とペアになるべきデータに付随するタグと同一のタグを定数は持っていなければならない。そのため動的にタグを設定し，定数のトークンを生成する機能が要求される。しかしIdでは定数発生について，このような関数的性質や意味付けが不明である。そこでCオペレータというものを導入した。これは，ある実行環境において定数として扱われるデータを外部より吸収し，その後，くり返しループの判定信号がtrueのとき（くり返しが許されるとき）に，内部の定数データを発生する。その定数の持つタグの値はCオペレータをトリガしたtrueの判定信号内のタグと同一とする。

図6のデータフローグラフは，Cオペレータの使用例である。

② プログラムのセグメント化

現在提案されているデータフローマルチプロセッサにおいて，構成要素である個々のPEは，並列処理が可能な命令がキューにつまれていても，ほぼword-at-a-timeにしか命令を発動できない。つまり命令フェッチの能力という点ではプログラムカウンタと大差がない。したがって並列性を多く含む1つのプログラムを少ないPEに割り当てたのでは並列性が生かしきれない。プログラムをできる限りモジュール化して，数多くのPEに割りふらなければならない。このときプログラムの分割は同時に計算負荷の均一な分散を果たしていることが望ましい。しかし動的な負荷特性を予測した方法は現状では見出しにくい。そこで，プログラムのテキストから，ループ本体，関数呼出し等，できる限りの論理レベルのモジュール化，つまりセグメント化を図り、その1つ1つのセグメントをPEへの割付けの単位と考えることにした。このような積極的なセグメント化によって，通信コストと計算量とのバランスが悪くなる場合も考えられるが、まず第1段階としてこういったアプローチをとった。

③ ループ，及び関数の呼び出し

Idではループの実行開始はLオペレータによって行なわれる。すなわちループの実行に必要なデータトークンに対し，そのタグフィールドを初期化してループ本体に引きわたす。ループの実行結果のもどり先も同時に記憶される。ループの終了時にはLオペレータによってもどり先が呼び出されもとのタグにもどして結果を転送する。これによりループ自体は独立した環境

で実行することができる。 ノつのループを並列に発動しても，L，L⁻¹オペレータによってプログラムの関数的性質は保証される。 一方，関数手続きの呼び出しはAオペレータによって行なわれる。 Lと同様，新しく独立した論理空間を生成する。 ただしLオペレータと異なるのは引数が全部そろってからでないと呼出しが行なえないという点である。 そこで引数がノつでも到着したら呼出しを許可するようにすれば，Lオペレータ，Aオペレータは統一的に扱うことができる。 本稿ではこの統一化したオペレータを改めて，L，L⁻¹オペレータとして定義する。 この結果，4章で述べるように非同期性を高めることができた。 並列処理の形態としてはlazy evaluationに近いと考えられる。 また実行環境の生成と消滅に関しては，Idよりもさらに具体的な意味付けを与えることにした。 Lオペレータはノつ以上の引数を持ち，それらが非決定的に到着するうちで最も早いものを資源分配のOSへのスーパーバイザコールとする。 L⁻¹オペレータについては一番最後の出力を転送した段階で，たとえハッシュテーブルにトークンが残っていたとしても，OSへの終了信号とし，それまでの資源をフリーにする。

4. 考察

図3，図4にIdに準じた高級言語（MID）で記述したガウスの消去法，及びFFTのプログラムを示す。 筆者らはそれらのプログラムを，Idの規則に基づき，図5に示すようなデータフローグラフに変換した。 そしてそれらをActivity Templateにおとし，実際の並列動作を机上で解析した。 その様子を図7に示す。 図5の中で，コロンの付された数字は命令番号を示しており、図7のテンプレートに対応している。 テンプレートにはオペランドを待つための領域が設けてあるが，これは単に理解を容易にするためであって，2章で述べたように実際には必要ない。 またデータトークンの構成は図8のようになっている。 このようにスナップショットを作成していくことによって実際にガウスの消去法の解を求め，プログラム，及び並列処理形態への変換規則の正当性を確認した。 FFTについては1ステージ分の計算結果を求め、動作を解析した。 ところで前章で述べたように，本稿でのアプローチは1セグメントにつき1プロセッサとしており、スナップショットの作成もこれに準じている。 したがってプロセッサの稼動状況についても知ることができる。 FFTの場合の並列実行環境を図9に示す。 図9の横軸には，1ステージ分の計算の過程で生成される26個のプロセスをとってあり，たて軸に時間（単位はユニットタイム）をとってプロセスの生成，消滅の様子を示している。 1プロセスは1セグメントに対応しており，図4のプログラムの各セグメントがどのプロセスであるかについても合わせて示した。 ただし，この解析には次のような前提を設けている。 すなわちデータフローグラフの各オペレータの実行時間はすべて同一とする。 したがって，データ構造操作のappend，select，加減乗除，スーパーバイザコールに相当するLオペレータもみな1ユニットタイムとしている。 これについてはシミュレータの作成段階で精密化する予定である。 図9から，実際に計算を行なっているプロセッサののべ台数と，タスクを割り付けられているプロセッサ（この中には，実動しているものに加えて，結果待ちの状態のものが含まれている）ののべ台数を求めると，各々，317台，

```
Gauss-elimination(a0,b0,m):=
 ! a0 is a matrix of m x m. b0 is a matrix of m x 1.!
 ! ansx is the answer of gauss elimination!
(Initial x0 ← nil;!segment 0!
 a0',b0' ← (Initial a1 ← a0, b1 ← b0;!segment 1!
            For k From 1 to m-1
            New a1, New b1 ← (Initial a2 - a1, b2 - b1
                              ! segment 2!
                              For i From k+1 To m
                              c ← a2[i,k]/a2[k,k]
                              t ← a2+[i,k]0
                              New b2 ← b2+[i](b2[i]-c*b2[k])
                              New a2 ← (Initial a3 ← t,a4 ← t
                                        !segment 3!
                                        For j From k+1 To m
                                        p ← a3[i,j]-c*a3[k,j]
                                        New a4 ← a4+[i,j]p
                                        Return a4)
                              Return a2,b2)
            Return a1, b1)
 x1 ← x0+[m](b0'[m]/a0'[m,m])
 ansx ← (Initial x2 ← x1, a1' ← a0', b1' ← b0';!segment 4!
         For i From 1 To m-1
         it ← m-i
         (Initial sm1 ← 0, it' ← it, a2' ← a1', x3 ← x2;!segment 5!
          For ix From it+1 To m
          New sm1 ← sm1+a2'[it,ix]*x3[ix]
          Return sm1)
         New x2 ← x2+[it]((b1'[it]-sm1)/a1'[it,it])
         Return x2)
 Return ansx)
```

図3 ガウスの消去法のプログラム

```
One-dimensional FFT   <segment 0>
TOPFFT(rdata,idata):=
!outer-most loop of FFT to 4 data points of
!complex data (real part data-rdata,imaginary
!part data-idata).
(Initial rdata ← rdata, idata ← idata
 (Initial nrb ← 2, cnt ← 0, brl ← 2
          rdata ← rdata, idata ← idata;
  For stn From 1 To 2
  New nrb ← nrb-1
  New brl ← brl/2
  New rdata, New idata
   ← ONESTN(nrb,brl,idata,rdata,stn)
  Return newrdata ← rdata, newidata ← idata)
  !
  ansrdata, ansidata ← SHUFFLE(newrdata,newidata)
 Return ansrdata,ansidata)

One-dimensional FFT  <segment 1>
ONESTN(nrb,brl,idata,rdata,stn):=
(Initial   nrb ← nrb, stn ← stn, brl ← brl,
           rdata ← rdata, idata ← idata
           rdbuf ← nil, idbuf ← nil, cnt ← 0
     while cnt<4
     rst,dxr,dxi,dyr,dyi,indx,indxy ← DATACS(cnt,nrb,rdata,idata,brl)
     !access of data pairs!
     pr, pi ← ROTFAC(stn,rst,tabcos,tabsin) !access of roration factors!
     ndxr,ndxi,ndyr,ndyi ← BUTFLY(dxr,dxi,dyr,dyi,pr,pi)
     !butterfly multiplication!
     New cnt←cnt+2
     New rdbuf←(rdbuf+[indx]ndxr)+[indxy]ndyr
     New idbuf←(idbuf+[indx]ndxi)+[indxy]ndyi
Return  rdata ← rdbuf, idata ← idbuf)
```

図4 1次元FFTのプログラム（次頁に続く）

896台となる。その比はプロセッサ稼動率と考えることができ，この場合35%となる。ガウスの消去法の場合には実動台数213台，全台数644台となり，稼動率は33%である。以下，この解析結果を踏まえて，プログラマビリティ等，種種の点について考察する。

[プログラマビリティ]

逐次型プログラムから変換しようとせず，アルゴリズムにおける結果の生成の過程を認識していれば，単一割当規則に従うことはあまり気にならず，想像以上に容易であった。ただし，配列を次々と更新していく典型的なフォンノイマン型のアルゴリズムの問題，例えばガウスの消去法のような場合には注意が必要である。つまり，同一名の配列を更新するということが単一割当規則では許されないから，その場合には名前を付け換えることが必要である。図3のa0，a2等のaで始まる変数はフォンノイマン型言語においては同一名ですむ。一見繁雑ではあるが，変数名ごとに論理的な意味付けがあるので，よりdefinitionalに問題を考えられるとも言える。

```
One-dimensional FFT   <Segment 2>
DATACS(cnt,nrb,rdata,idata,brl):=
!get pointers (indx,indxy) to data pairs according to coun-!
!ter(cnt). get values of data pairs.!

(Initial   cnt ← cnt, nrb ← nrb, brl ← brl, rdata ← rdata,
           idata ← idata, n ← 4
 bcn ← DETOBI(CNT)
 rst ← REVBIT(bcn,nrb)
 indx ← BITODE(rst,n)
 indxy ← indx+brl          !get pointer of another pair!
 dxr ← rdata(indx)         !real part of data pair X!
 dxi ← idata(indx)         !imaginary part of pair X!
 dyr ← rdata(indxy)        !real part of data pair Y!
 dyi ← idata(indxy)        !imaginary part of pair Y!
 Return   dxr, dxi, dyr, dyi, indx, indxy)
```

```
One-dimensional FFT  <segment 3>
ROTFAC(stn,rst,tabcos,tabsin):=
(Initial stn ← stn,rst ← rst,tabcos ← tabcos,
         tabsin ← tabsin
 (Initial pvec ← nil,stn ← stn ;!<segment 3a>!
  For k From 1 To stn -1
  New pvec ← pvec+[k]0
  Return pvec)
!
 (Initial lstn ← stn,ppvec ← pvec,rst ← rst,kl ← 1
          !<segment 3b>!
  While lstn=<1
  New ppvec ← ppvec+[lstn]rst[kl]
  New lstn ← lstn+1
  New kl ← kl+1
  Return ppvec)
!
 (Initial ptemp ← nil,n ← 1,scale ← 2;!<segment 3>!
  While scale>0 Do
  New ptmp ← ptmp+[n]0
  New n ← n+1
  New scale ← scale-1
Return ptmp,n)
!
 (Initial ppvec ← ppvec,ptmp ← ptmp,n ← n,i ← 1;
  !<segment 3c>!
  While n=<3
  New n ← n+1
  New ptmp ← ptmp+[n]ppvec[i]
  New i ← i+1
  Return ptmp)
!
 (Initial pwr 0,power ← ptmp;!<segment 3d>!
  For i From 1 To 2
  New pwr ← pwr+power[4-i]*2**(2-i)
  Return pwr)
!
 (Initial pwr ← pwr,n ← 16;!<segment 3e>!
  If pwr=0
   Then ppwr ← pwr+1
        cos ← tabcos[ppwr]
        sin ← tabsin[ppwr]
  If pwr=1
   Then ppwr ← n/4-pwr*2+1
        cos ← tabsin[ppwr]
        sin ← tabcos[ppwr]
  If pwr=2
   Then ppwr ← pwr*2-n/4+1
        cos ← -tabsin[ppwr]
        sin ← tabcos[ppwr]
  If pwr=3
   Then ppwr ← n/2-pwr*2+1
        cos ← -tabcos[ppwr]
        sin ← tabsin[ppwr]
  Return cos,sin)
Return pr ← cos,pi ← sin)
```

```
One-dimensional FFT   <segment 4>
BUTFLY(dxr,dxi,dyr,dyi):=
(Initial dxr ← dxr,dxi ← dxi,dyr ← dyr,
         dyi ← dyi,
         pr ← pr,pi ← pi;
 ndxr ← dxr+dyr
 ndxi ← dxi+dyi
 t1 ← dxr-dyr
 t2 ← dxi-dyi
 ndyr ← pr*t1-pi*t2
 ndyi ← pi*t1+pr*t2
Return ndxr,ndxi,ndyr,ndyi)
```

```
One-dimensional FFT   <segment 5>
SHUFFLE(newrdata,newidata):=
(Initial srdata ← newrdata,
         sidata ← newidata;
 For cnt From 0 To N/2 Step 2
 sbcn ← DETOBI(cnt)
 srst ← REVBIT(sbcn,n)
 rcnt ← BITODE(srst,n)
 srtemp ← srdata+[cnt+1]srdata[rcnt+1]
 sitemp ← sidata+[rcnt+1]srdata[cnt+1]
 New srdata ← srtemp+[rcnt+1]srdata[cnt+1]
 New sidata ← sitemp+[rcnt+1]srdata[cnt+1]
Return ansrdata ← srdata,ansidata ← sidata)
```

```
One-dimensional FFT   <segment 6>
DETOBI(A  ):=
!produces binary vector!
(Initial cnt ← <A >, tmp ← nil
 If cnt=0
  Then New tmp ← [0 0]
  Else (Initial cnt ← cnt
         (Initial cnt ← cnt,i ←1;
          !<segment 6a>!
          While cnt<>0 Do
          p ← cnt MOD 2
          New i ← i+1
          New cnt ← cnt/2
          New tmp ← tmp+[i]p
          Return tmp,i)
         (Initial tmp ← tmp,i ← i;
          !<segment 6b>!
          While i=<2
          New tmp ← tmp+[i]0
          New i ← i+1
          Return tmp)
        Return tmp)
 Return < > ← tmp)
```

```
One-dimensional FFT   <segment 7>
REVBIT(A,B):=
!bit reverse of binary vector!
(Initial templ ← <A>,num - <B>,
         temp2 ← <A>
 For i From 1 To num
 il ← num+1-i
 New temp2 ← temp2+[il]templ[i]
Return < > ← temp2)
```

```
One-dimensional FFT   <segment 8>
BITODE(A,B):=
!transform binary vector to a value!
(Initial sum ← 0,temp ← <A>;!<segment 8>!
 For j From 1 To <B>
 New sum ← temp[j]*2**8j-1)+sum
 Return  < > ← sum)
```

〔lazy evaluation〕

引数が全部そろっていなくても関数や手続きの発動を許すことによってプログラム全体における並列性が増加した。たとえばループの場合，ループ変数は外部から供給される主要なデータが到着するのに先んじて，タスク実行要求を生成し，時には要求を出しきっている場合もある。この形態は lazy evaluation と考えることができる。また主要な計算の開始に先立って，プロセッサの確保，手続きのコピー等，実行環境の生成要求が出されるので，OSの見かけ上のオーバーヘッドは低減される．

〔データ構造操作〕

本稿に示したプログラムではnilデータに1要素ずつ逐次的にデータをアペンドしながら配列を生成するという処理が少なからずある。しかしながら，その多くの場合，並列にアペンドすることが可能であり，問題の持つ並列性を，Idのstrictアペンドがつぶしている。これに対しVALでは，並列に生成するデータの数だけ実際にActivity Templateを用意するので並列実行が可能である（forall 文）。これはVALがコンパイラベースであることの利点である。しかしデータ数が実行時にならなければわからないような場合，いくつテンプレートを用意したらいいのか決めることができない。そのため最悪の場合には有限の資源以上にテンプレートを生成し，デッドロックを生ずることもおこりうる。これを回避するために制御情報をユーザに書かせることも検討されているらしい[3]が，これは明らかに退歩である。しかしIdの場合はインタプリタであるから，そのような問題は回避できる。要はインタプリタの速度を論理的にも早めることである．そのためにArvindはnon-strictアペンドを提案している[2]．これはアペンドすべきデータがまだ到着しない段階で，それを必要とする次のプロセスに空のまま手渡すというもので，lazy evaluationの一種と考えられ

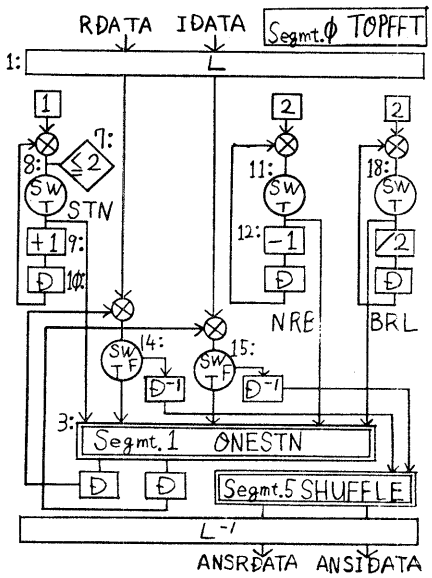

図5 FFTメインプログラムのデータフローグラフ

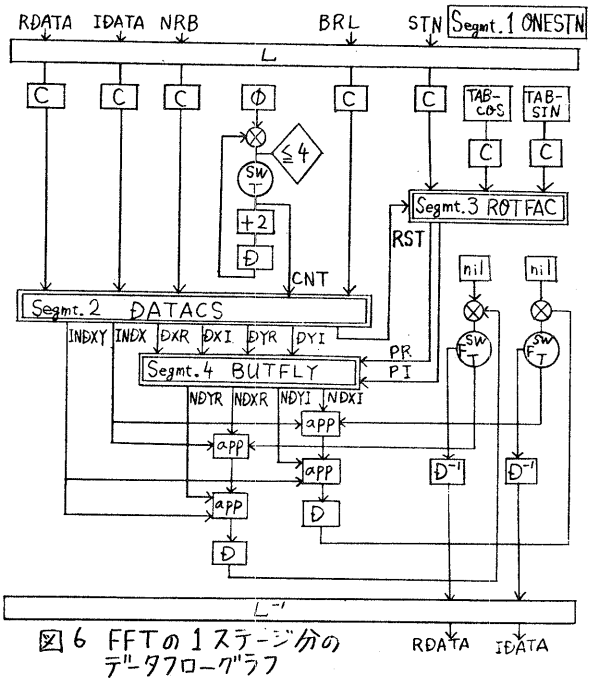

図6 FFTの1ステージ分のデータフローグラフ

る。 これによって論理レベルでの並列性は明らかに増加する。しかし，non-strict アペンドはストラクチャメモリの高機能化を前提としており、単に論理レベルのシミュレーションでは実際的な意義に欠ける。つまりストラクチャメモリの基本構成をシミュレーションの対象に加え，その中での挙動も含めて全体をとらえなければ，アーキテクチャの検討材料にはならない。この問題についてはまだ検討中である。

［謝辞］
末筆ながら、日頃御指導頂く慶應義塾大学相磯秀夫教授に深謝する。御指導，御討論頂く，電子技術総合研究所，横井俊夫技官，ならびに情報システム研究室の方々に深謝する。

［参考文献］
1. 内田，樋口：データフローマシンの魅力と可能性について，情処学会，計算機アーキテクチャ研資(1980.6.25)
2. Arvind：データフローアーキテクチャの研究開発，昭和54年度特別セミナー講演録，55-C-391，電子協(Mar. 1980)
3. McGraw, J.R.: Data Flow Computing: Software Development, 1st conf. on Distributed Computing Systems, Oct. 1979

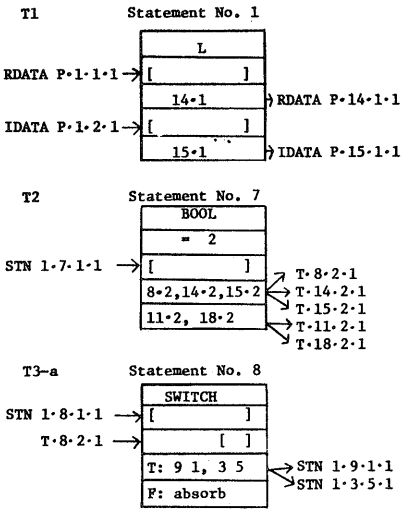

図7 スナップショットの例
(TOPFFT(図5)の実行の初期)

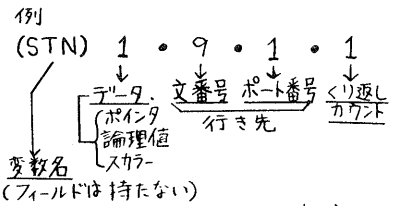

図8 データトークンの構成

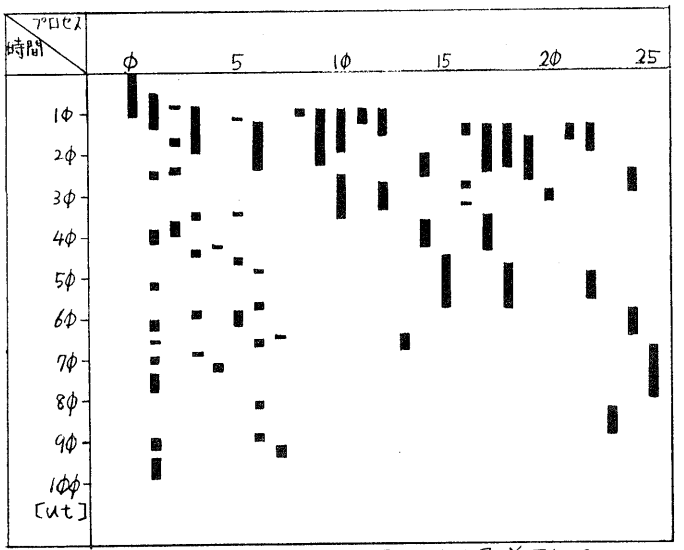

図9 FFTにおける並列プロセス

プロセスとセグメントの対応

プロセス番号	セグメント	プロセス番号	セグメント
0	0	13	3e
1	1	14	3c
2	2	15	3d
3	3	16	6
4	4	17	7
5	5	18	8
6	3	19	6a
7	4	20	6b
8	6	21	3a
9	7	22	3b
10	8	23	3e
11	3a	24	3c
12	3b	25	3d