計算機アーキテクチャ 38－2
（1980. 6. 25）

データフローマシンの魅力と可能性について

内田俊一　（電子技術総合研究所）
樋口哲野　（慶応大学　工学部）

1．はじめに

計算機の応用分野の拡大は，従来の数値計算の枠組を越えたパターン情報処理，データベースや記号処理を含む知識情報処理までも，その処理対象とするまでに至った。この結果，解くべき問題の論理構造と，マシンのアーキテクチャ間のギャップが顕著となりこのギャップを埋めるべきソフトウェアの負担が著しく過大となった。G.J. Myers は，このギャップをセマンティックギャップと呼んでいる。[1]

一方，ハードウェアについては，VLSI などの素子技術，マルチプロセッサなどのアーキテクチャ技術の進歩により，その性能は目ざましく向上した。そこで，マシンを高レベル化し，ソフトウェアの負担を軽減することが試みられた。しかし，高級言語マシンや汎用マルチプロセッサ開発の試みは，そのプログラム言語や計算モデルに関する理論的指針に明確さを欠いたこともあり，順調には進まなかった。

データフローマシンは，この理論的基盤を持っているが，その価値を理解するためには，関数型言語との対応や並列処理のための言語の性質の明確化，ノイマン型モデルとその言語の問題点の整理など，ソフトウェア工学的な研究の進展が必要であった。[12]

このほか，知識表現や推論など，より高度な問題を記述するためのモデルや言語との整合性のよいことも，データフローマシンを，魅力的なものとしている。さらに，VLSI技術が，多数のプロセッサ，メモリ，通信路を必要とするデータフローマシンの実現の基盤となっていることも見逃せない。

本論文では，並列処理の問題点を絞って，データフロー言語とマシンの利点について述べる。

2．並列処理におけるセマンティックギャップ

2.1　並列処理の記述と望しい条件

解くべき問題が与えられ，それを並列マシンにより解こうとするとき，その過程は，図－1のようなものとなる。この過程において，望しい条件は，以下のようなものであろう。

a) マシンのアーキテクチャや構成の細部が，アルゴリズムの選択やプログラムの記述に直接影響しないこと。
　すなわち，問題の性質に従った素直な記述ができること．（言語レベルが高いこと．）

b) 並列実行部分を，プログラマが細部にわたって明示しなくてよいこと。
　すなわち，マシンが自動的に並列化してくれること。

c) 言語プロセッサや OS などが複雑化しないこと。　すなわち，プログラムとマシン間の並列計算構造の写像が簡単であること。

2.2　従来の方法における制約と問題点

2.1で述べたような条件を満たすためには，図－1に示した次のような制約を検討する必要がある。

① 解くべき問題の性質からくる制約．
② アルゴリズムによる制約．

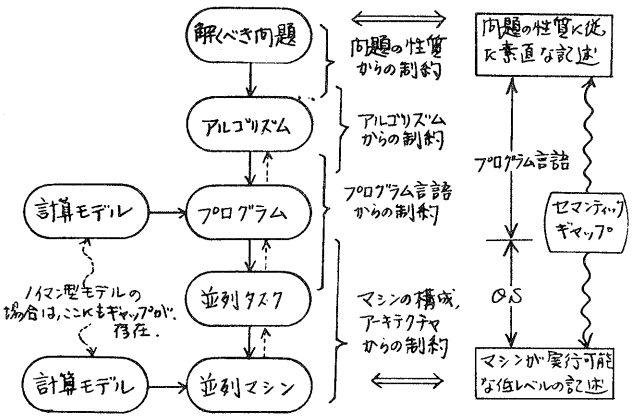


図-1 並列プログラム実行のための種々の制約とセマンティック・ギャップ

③プログラム言語とその計算モデルのための制約
④マシンのアーキテクチャと計算モデルの制約

問題自身は，何らかの形での並列計算構造を持つものとして，②のアルゴリズムを考える。アルゴリズムは，当然のことながら，逐次処理を前提としたものが多く，並列処理を行おうとするとしばしば問題となり，並列アルゴリズムの研究の進展が待たれる[2]。

③の例としては，FORTRAN等のノイマン型言語を用いて，並列処理を記述する場合がある。このような場合の並列処理は，多くの場合，多重にネストしたループ構造として記述される。一旦，このように記述（コーディング）されてしまうと，これから再び並列計算可能な部分をとりだすことは，極めて困難である。言語仕様を拡張して，アレイ用のデータ型や演算，for-all文等を導入[3]しても，記憶領域の共有関係など，不自然な制約を課すことが多く，プログラマビリティは，向上しない。また，スカラー演算などの細部の並列計算構造の検出は，コンパイラの巨大化を伴い，特定のマシンを対象とする場合以外は，実用的でない。

④の例としては，従来のマルチプロセッサに多く見られるような，計算モデルの不整合の場合がある。すなわち，要素プロセッサは，通常のノイマン型マシンであるが，全体としては，並列マシンとなっており，要素プロセッサ間の結合が密で，データの共有等をいろいろなレベルで行うような場合である。

このようなマシンでは，プログラマに，要素プロセッサへ割当てるタスクの細かな指定や，タスク間のデータ共有関係，タスク実行順序，排他領域制御などを明示させざるを得ず，プログラム言語やアルゴリズムにまで，悪影響を及ぼすこともまれでない。（図-1中の点線）

信号処理や行列演算など問題の性質が並列処理に適したものでは，このような制約内でも，有効に使用できる場合もあるが，汎用性という点では不十分といえよう。

以上のような問題点に対し，データフローマシンは，有力な解決法を与える。

3. データフローマシンによる並列処理

3.1 データフロー言語

データフローモデルやその言語については，論文や解説も多く，[4][5][6][7]詳細は省略するが，モデルのプリミティブの違いにより，いくつかの方言が存在する。データフロー言語としては，2次元のグラフ言語と，これに対応する文章型言語があるが，これにも，色々と方言がある。代表的なものとしては，AckermanのVAL[14]，ArvindのId[9]，ツールーズ大学のLAU，ロンドン大学のCAJOLEなどがある。

実行時の形態も，VALがコンパイ

ラ型言語であるのに対し，Idがインタプリタ型言語である等の違いがある。

しかし，関数型言語であること，単一割当規則を持つことは，ほとんどのものについて共通している。

計算モデルや言語仕様の細部は，マシンの実現を考慮してつめていくことで，いろいろと言わってくる。著者らは，Idを元としたモデルと言語を拡張したMID(Modified Id)と呼ぶ言語を用いている。[8][9][10]

文章型言語とグラフ言語は，ほぼ完全に対応し，変換は，極めて容易である。実行に際しては，文章型言語は，グラフ言語の各プリミティブと対応するアクティビティテンプレート[4]に変換される。

アクティビティテンプレートは，データフローマシンの機械語に対応し，MIDでは，マシンがこのテンプレートを解釈実行することで，次々と並列実行可能なタスクが生成される。

3.2 データフロー言語による記述の利点

データフロー言語によって並列処理を記述すると次のような利点がある。

①個々の演算は，データ共有などに伴う副作用を持たず，細かなスカラー演算のレベルまでの関数性が保証される。

②モジュール(関数)間のインタフェイスが明確である。

③記述レベルが高く，マシン・アーキテクチャから独立した言語レベルの記述が可能。

④プログラム間の変換が容易であるほか，検証なども行いやすい。

①の条件により，問題に含まれている並列計算構造が不規則なものでも，マシンの側でそれを生かすことが可能となる。ノイマン型マシンの命令のオペランドレベルまでの並列化も可能である。

②により，並列タスクの含まれる計算量を，いろいろなレベルで調整できる可能性がある。マシンの負荷に応じたタスク割当の最適化等の自由度も大きい。

③により，並列計算部分を逐次的に記述したり，要素プロセッサ数を陽に考慮してプログラムする必要はなく，問題やアルゴリズムに従った素直な記述が可能。

④により，MIDからpure LISPへの変換なども容易である。

以上，数学的基盤にもとずいていることから，いろいろな利点があり，解くべき問題の論理構造が複雑化した場合にも，機械的検証手法などが適用できる可能性もあり[11]，ノイマン型言語に比べ，扱いやすい。

3.3 データフロープログラムの例

データフロー言語MIDで書かれたプログラムから，並列タスクが作られるまでの過程を，簡単な例で説明する。

より本格的なプログラムは文献[8]を参照されたい。

1)行列乗算プログラムの場合

図－2は，文章型言語で書いたプログラムで，行列a_0(n行×m列)と，b_0(m行×n列)の積を計算する。説明の便宜上，Mmult，F1，F2という3つの関数(サブルーチン)に分けてあるが，これらを入れ子構造にしても同様に扱える。それぞれの関数に対応するグラフ言語表現は，図－3，図－4，図－5に示す。グラフ言語表現では，文章言語にはないプリミテ

イブ L, L^{-1}, D, D^{-1}, C, app, sel が含まれている。 LとL^{-1}は，文章型言語のカッコに対応し，それらで囲まれた文脈（コンテクスト）が，一つのモジュール単位として独立していることを示し，トークンのタグ（名札）のスコープを決めている。 DとD^{-1}は，ループ構造の中で，単一割当規則が成立するよう，トークンのタグを更新し，その出口で，リセットする。 Cは，その文脈中で定数としておわれる引数をとり入れ，保存する。 appとselは，配列のような構造を持つデータに対するappendと，指定された要素を抜き出すSelectを行うオペレータである。 また，2重の四角で囲んだものは，関数呼出しを示している。 nil（Λ）は構造データの初期化である。

```
Mmult(a0,b0,h,m,n):=
 (Initial c0 ← nil
   For i From 1 To h Do
    New c0[i] ← F1(i,a0,b0,m,n)
  Return c0)


F1(i,a0,b0,m,n):=
 (Initial c1 ← nil
   For j From 1 To n Do
    New c1[j] ← F2(i,j,a0,b0,m)
  Return c1)


F2(i,j,a0,b0,m):=
 (Initial c2 ← nil
   For k From 1 To m Do
    New c2 ← c2+a0[i,k]*b0[k,j]
  Return c2)

Program Example: Matrix Multiplication
```

図-2 行列乗算プログラム（MIDによる）

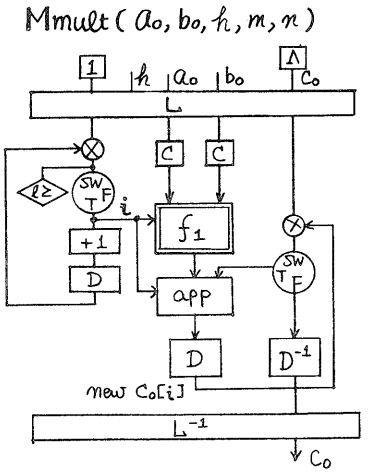


図-3 関数 Mmultの データフローグラフ（MIDによる）

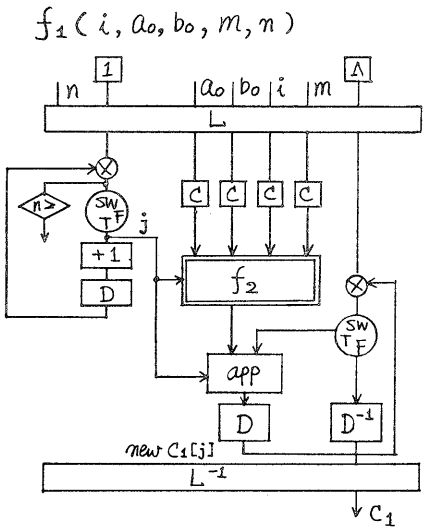


図-4 関数 F1の データフローグラフ（MIDによる）

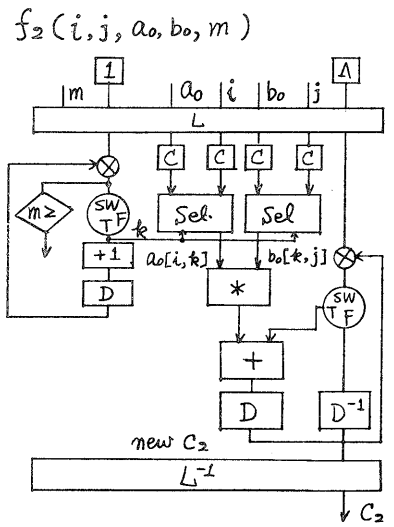


図-5 関数 F2の データフローグラフ（MIDによる）

実行に際しては，マシンへは，グラフのプリミティブに対応するアクティビティテンプレートが与えられるが，ここでは，グラフ表現をもとに，実行過程を説明する。

まず，Mmult が呼ばれると，Lオペレータの入力がすべてそろうため，この関数が実行可能となり，1つのタスクとして，要素プロセッサ（たとえば，PE_{00}）へ割当てられる。（図-6(a)）

Mmultの内部では，ループ変数 i を計算する部分が先行して実行され，次々と関数 f_1 の呼出しが行われる。f_1 の他の入力は定数であるから，i が与えられるたびに，f_1 は，並列タスクとして，次々と空いている要素プロセッサへ割当てられ実行される。f_1 の一つが割当てられた要素プロセッサ（たとえば，$PE_{1,i}$）は，Mmultと同様，ループ変数 j の計算を先行させることにより，関数 f_2 を，次々と呼び出す。f_2 は f_1 と同様，j が与えられれば，実行可能となる。f_2 は，与えられた行列 a_0, b_0 のコピーから，$a_0[i,k]$, $b_0[k,j]$ の要素を Select し，乗算と加算を行う。乗算は，入力が到着すれば，次々とオーバラップしながら，並列実行される。加算は逐次的にしか，行われない。乗算や加算も，小さいながら，並列タスクと考えられる。f_2 は，計算が終ると，f_2 を呼出した親関数 f_1 に値を返し，f_1 は，残りの演算を行う。f_1 の結果は，Mmultに集められ，答がでる。

以上述べた関数 f_1, f_2，ループ変数の計算，乗算，加算，select，append などは，引数（又は入力）と出力の受け渡し以外は，依存関係はなく，データ駆動の原則に従い，それぞれの資源の許す範囲で，非同期的に並列実行される。これら，タスク間を行きかうトークンは膨大な数となるが，それらに対しユニークなタグをつけるメカニズムも，容易に実現できる。

このように，データフロー言語では，並列タスクとして，仰々の演算までもとりだすことができ，マシンの側で，1ケのタスクの計算量を調整する等の操作が可能である。図-6(a)では，f_2 が，最小のタスクとしてホした。また，上の説明では，図-6(a)のような説明をしたが，図-6(b)のような対応づけも可能である。（但し，ハードウェアや OS の負担は増える。）

行列計算や信号処理演算では，多くの並列タスクの計算量を，同一にそろえることも可能で，上の例の f_2 を，プリミティブな演算と考えれば，従来のマルチマイクロプロセッサ等に，そのままのせることも可能と思われる。

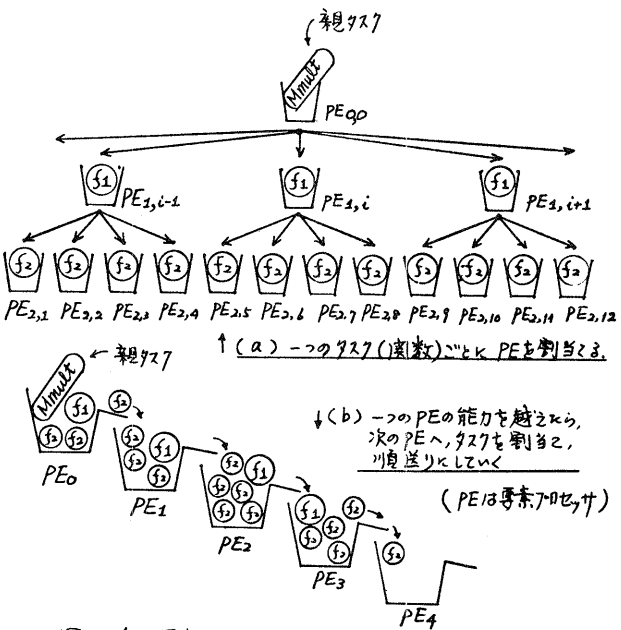


図-6 要素プロセッサとタスクの関係の2例

(2) Quicksortの場合

データフロー言語では，さらに，不規則な並列計算構造を有する問題についても，有効な処理が可能

```
Qsort(a,n):=
 (If n=<1 Then a Else
   (Initial below ← nil; j ← 0;
            above ← nil; k ← 0;
     For i From 2 To n Do
      New below, New j, New above, New k ←
   f { (If a[i]=<a[1]
       Then below+[j+1]a[i], j+1, above, k
       Else below, j, above+[k+1]a[i], k+1)
     Return Coalesce(Qsort(below,j)+[j+1]a[1], j+1,
                     Qsort(above,k),k)))

Coalesce(s,j,t,k):=
 (Initial r ← s
   For i From 1 To k Do
    New r[j+i+1] ← t[i]
  Return r)
```

図-7 Quicksortのプログラム
(MIDによる)

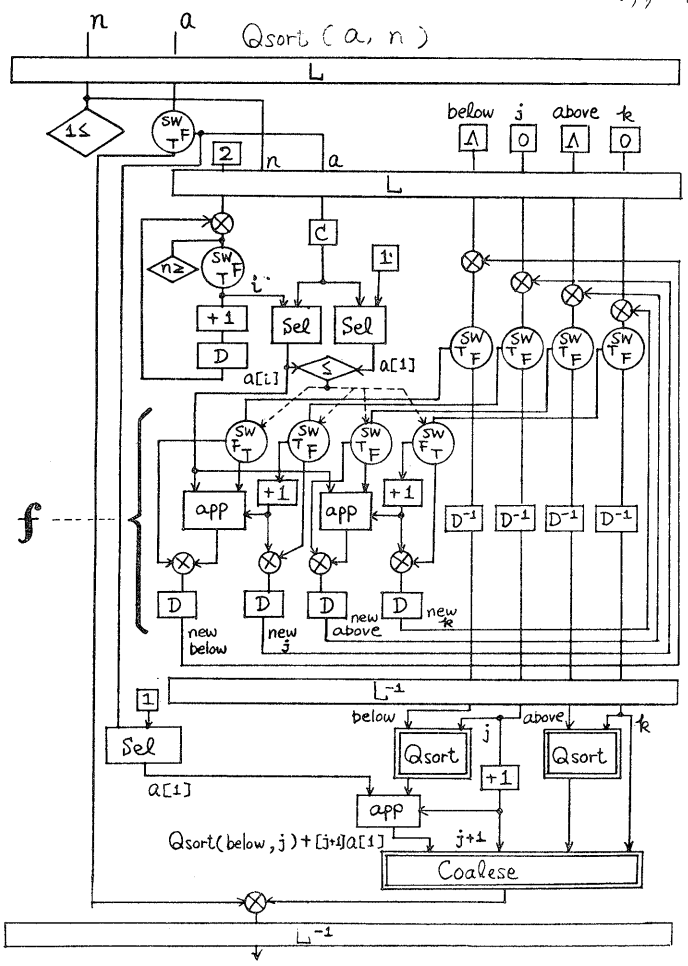


図-8 Quicksortプログラムの
データフローグラフ (MIDによる)

である。Quicksortは、その一例である。そのMIDによる文章型表現を、図-7、グラフ表現を図-8に示す。説明の便宜上、この関数の中央のIf文の範囲をfと書く。

この関数Qsortの実行過程を、少しマクロに関数レベルで示すと、次頁の図-9のようになる。図中で、くくりで囲んだ数字は、ソートの中間結果である。実行の詳細は省略するが、問題自身の中に含まれている並列性が素直に現れていることがわかる。もちろん、このレベル以下の、ループ変数の更新や、select, append などの操作も、データ駆動の原則のもとに並列実行可能である。

Quicksortと行列乗算の大きな違いは、その問題(アルゴリズム)の性質に起因するタスク(関数)ごとの処理量の変化の度合であろう。Quicksortの場合は、図-9からもわかるように、関数Qsortが再帰的に呼ばれるたびに、入力となるデータ量は、不規則に変化する。このため、要素プロセッサとタスクとの対応は、図-6(b)のようなものである必要があろう。

また、構造データに対するselectとappend操作がほとんどであることから、マシンとしては、これらの操作が能率よく実行できるハードウェアサポートを持つ必要がある。

以上、2つの例を考えるが、データフロー言語を用いることで、並列処理に関する限り、従来の方法に比べ、きわめて自然に、並列

タスクにまでおとすことが可能である。

しかし，これは，データフローモデルに基く高レベルマシンを前提としており，以下，そのアーキテクチャについて考察する。

また，データフローモデル自身についても，パイプライン処理など，さらに並列性を強化したり，経過依存性をとり入れるためのストリームの導入などの拡張が望まれる。[10]

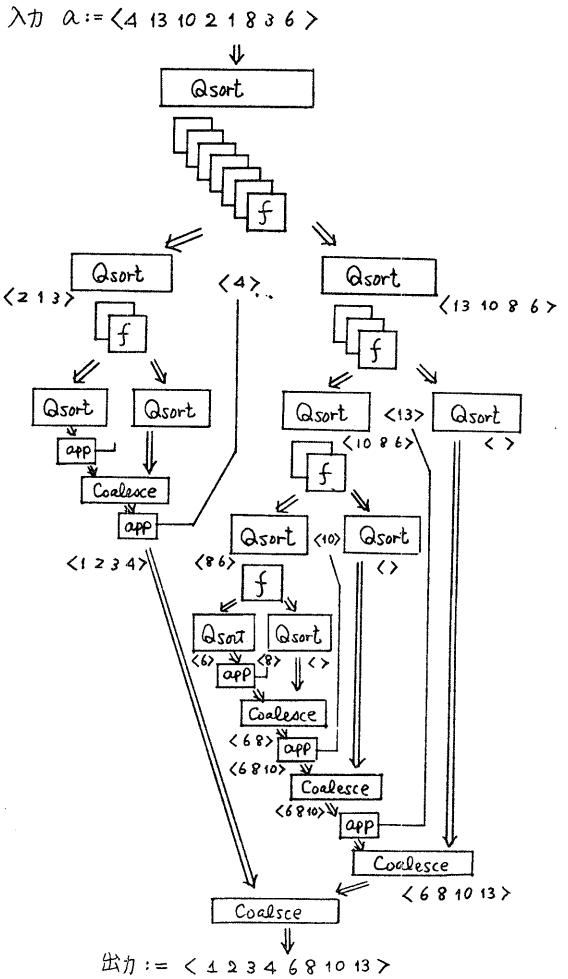


図-9 Quicksortの実行過程

4. データフローマシンのアーキテクチャ

4.1 マシン構成の考察

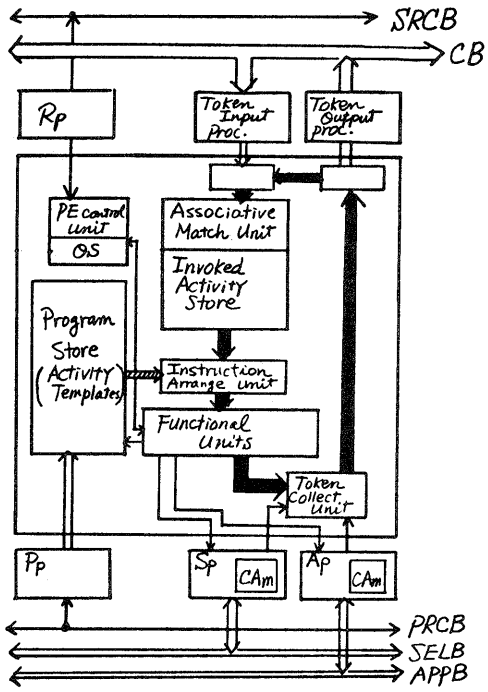
「データフローマシン」という言葉は，「ノイマン型マシン」と同様，その計算モデルに基く抽象的マシンの呼称であり，その具体的なアーキテクチャは，その用途やパーフォマンスに基き，いろいろなものが存在し得る。共通する特徴は，計算モデルに処理の非同期的実行メカニズムが本質的に含まれていることと，さらに，対応する関数型言語を持つことである。

このため，数値計算用データフローマシンのような専用機を作った場合でも，そのプログラマビリティは，現在のアレイプロセッサよりも，格段に優れたものとなる。行列算算の例からも推測できるように，このような目的の専用マシンは，データフローマシンの場合でも，当然作りやすくなる。

一方，より汎用的な記号処理マシンの場合には，Quicksort+の例にも見られたような細かで，不規則な並列計算構造を効果的に実行する機能のほか，構造をもつデータ操作を高速に行う機能が必要である。

言語レベルの処理では，これらの演算や操作の細部にわたる並列性は，もし，問題やアルゴリズムの中に存在するなら，そのまま，マシンへと伝えられる。よって，そのセマンティックギャップは，大幅に狭められており，あとは，上のような機能を持つマシンを作ればよいということになる。そして，そのマシンは，十分汎用であるとともに，プログラマビリティもよいことが期待できる。

マシンの構成については，すでにいくつかの提案があるが[7][13]全体構成としては，図-10に示すようなマルチプロセッサ構成が，自然である。この要素プロセッサ（DFPE$_i$）としては，



図－11　データフロープロセッシングモジュール（DFPM）の構成例

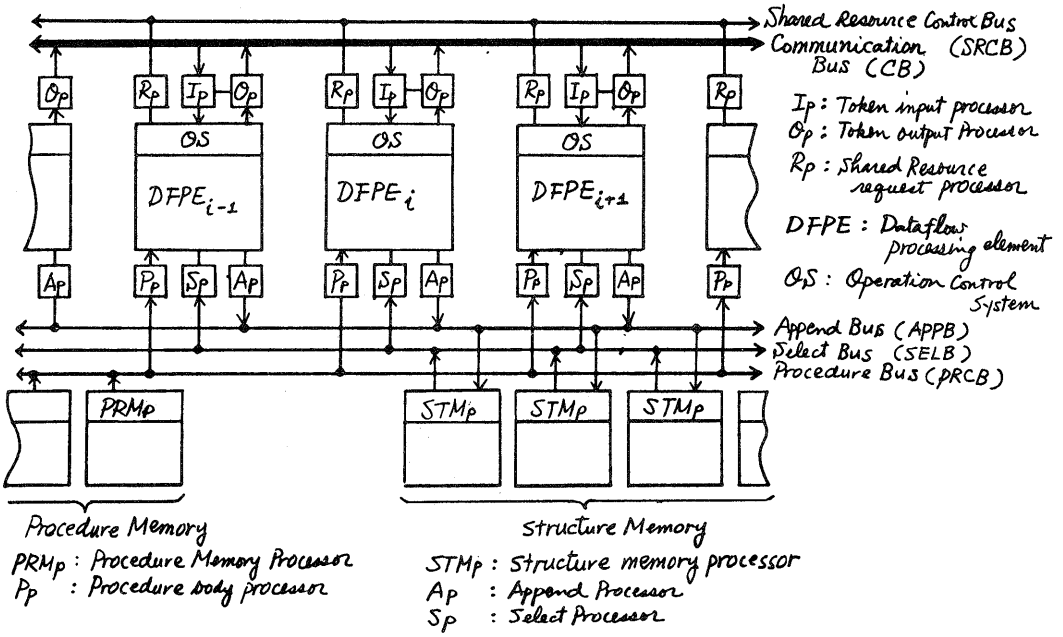
著者らは，図－11に示すような構成のものを考え，アクティビティテンプレート（待機語）などを検討している。

プリミティブは，演算や操作は，この要素プロセッサ中の，黒矢印のループをまわりながら，並列実行される。

配列や木構造などの構造を持つデータについては，関数の引数として渡されるとき，毎回コピーすることは容量的に無理があり，構造メモリ（Structure Memory）[15]を用い，共通データの共有化を行うとともに，実際にデータの中身が必要となるまでは，ポインタを渡していく。

また，要素プロセッサに新たなタスクが割当てられたとき，その関数本体（procedure body）を，読み込む待機を設ける。（図－10のprocedure memory）記号処理では，関数本体もデータとしておうことから，構造メモリへ格納する。

そのほか，要素プロセッサ間で，トークンをやりとりする通信路（Communication



図－10　データフローマシンの構成例

Bus)や，空きプロセッサの管理や，データ入出力などのための通信路(Shared Resource Control Bus)を設ける。

要素プロセッサと，これらのバスとの間は，それぞれの目的に合ったプロセッサを介して接続する。以上，いろいろな特徴を考慮しているが，図-1はあくまで，概念的なものであり，シミュレーション等により精密化されるべきものである。

4.2 要素プロセッサの構成と動作

要素プロセッサの構成は，図-4に示すものであるが，その動作を，行列乗算の例に従って，簡単に説明する。

いま，図-5に示す関数f_2が，1つのタスクとして，このプロセッサに割当られたとする。その呼出しのトークンがToken Input Processorを経由して送られてくると，まず関数本体が，Program Storeへ読み込まれる。(すでに格納されていることもある。)その後，Lオペレータが駆動され，図中の黒矢印のループを一周すると，ループ変数kを計算するオペレータが起動される。これらへの入力は，1個であり，待ち合せの必要はないから，次々と黒矢印のループをまわり，kの値を作りだす。すると，Selectオペレータが起動され，トークンとして，Functional Unitへ送られる。次に，Sp(Select processor)が起動され，Cオペレータにより与えられたポインタに従って，行列a_0, b_0を，構造メモリから，読みだしてくる。Spは，キャッシュを持ち，構造メモリのアクセス回数を減らしている。

Selectorの出力として，$a_0[i,k]$, $b_0[k,j]$がそろうと乗算，加算が起動される。乗算のように，2つのデータを待ち合せる演算は，一旦Inworked Activity Storeへ格納され，もう一方のトークンが到着すると実行される。

計算が終了するとLオペレータは，計算結果C_2をToken Output Processorへ送り，親プロセッサへ戻すとともに，関数f_2の実行のために専有していた資源を開放する。このほか，関数呼出しが行われるような場合には，OSへ資源割当要求が出され，空きプロセッサのaddressを得て，そこへトークンを送る。

また，同一プロセッサ中で，いくつもの関数が実行される場合には，マルチプロセッシングのための管理が，行われる。(図-6(b)のような場合)

5. おわりに

データフローマシンの魅力として，並列処理に関するセマンティックギャップが，大幅に改善される点を主に示し，不規則な並列計算構造についても適用できる可能性を示した。

不規則な並列計算構造を効果的に実行できるマシンは，汎用データフローマシンともいうべきものと考えられる。

まだ，マシンの中の各要素プロセッサの稼動率等を検討する段階ではないが，このようなマシンの評価を行う場合，従来の発想を変更しなければならない点がいくつかでてきそうである。

その一つは，プロセッサを通信の中継点として使うことを，どう考えるかである。記号処理のような問題の処理では，関数を次々と呼びだし，プロセッサ間を関数呼びだしの結果できる通信路で結んでいく。そして，できあったプロセッサ間の従属関係の構造(たとえば，図-6(a)の木構造)が処理の途中結果を示しており，この構造を変化させることが，処理とも考えられる。当然のことながら，ほとんどのプロセッサは，アクティブなトークンを持たず，一種の動的停止状態にある。従来の考え方によれば，他のタスクを割当てるところであるが，大規模な分散処理システムであるとの

前提にすると，これは，それ程容易なことではない。むしろ，通信の中継点として機能することをタスクと考えた方が自然のように思える。

数値計算の場合でも，同様の問題があり，計算と通信の差が不明確となる。このような点をも含め，今後，マシンのシミュレータを作成し，いろいろな問題について，検討を加えていく予定である。

最後に，御指導，御討論を戴く，電子技術総合研究所 横井俊夫技官，石川康一技官，淵一博パターン情報部長に深謝する。また，研究の機会をそろえられる井田ソフトウェア部長，榛葉昭男情報システム研究室長に深謝する。

参考文献

1. Myers, G.J.: Advances in computer architecture, John Wiley & Sons, (1978)

2. Kung, H.T.: The structure of parallel algorithms, CMU Tech. Memo. CMU-CS-79-143, (August 1979)

3. Perrott, R.H.: A Language for Array and Vector Processors, ACM Trans. on programming languages and Systems, vol. 1, No. 2, pp. 177-195, (Oct., 1979)

4. Dennis, J.B.: The varieties of data-flow computers, Proc. 1'st Int'l Conf. on Distributed Computer Systems, pp. 430-439, (1979)

5. 樋口, 山口: データフロー言語の研究動向, 信学本誌, vol. 63, No. 1, pp. 83-87, (Jan. 1980)

6. Ackerman, W.: Data flow languages, Proc. NCC, pp. 1087-1095, (1978)

7. 宇都宮: データフロー計算機, bit, vol. 12, NO. 3～7 まで連載, (1980)

8. 樋口, 内田: 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, 情処学会, 計算機アーキテクチャ研資, (1980年6月25日)

9. Arvind, etal: An asynchronous programming language and computing machine, Tech. Rep. #114A, Univ. of California, Irvine, (Dec. 1978)

10. 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)

11. Achcroft, E.A, Wadge, W.W.: Lucid, a Nonprocedural Language with Iteration, CACM, vol20. No. 7, pp. 519-526, (Jul. 1977)

12. Wegner, P(Ed): Research directions on Software Technology, MIT Press, (1979)

13. Rumbaugh, J.: Dataflow multiprocessor, IEEE Trans. on Computer, vol. C-26, No. 2 pp. 138-146, (Feb. 1977)

14. Ackerman, W.B., Dennis, J.B: VAL - A Value-Oriented Algorithmic Language, Preliminary Reference Manual, LCS, MIT, (Sep, 1978)

15. Ackerman, W.B., A structure memory for data flow computer, MIT/LCS/TR-186, (Aug. 1977)