リスト処理向きデータフローマシンアーキテクチャとそのソフトウェアシミュレータ

List Processing Oriented Data Flow Machine and Its Software Simulator

雨宮真人　長谷川隆三　三上博英
Makoto AMAMIYA　Ryuzo HASEGAWA　Hirohide MIKAMI
（日本電信電話公社　武蔵野電気通信研究所）
Musachino Electrical Communication Laboratory, N.T.T.

1．まえがき

将来の情報処理システムには高度の知能的処理機能が要求され、非数値処理を高性能に実現するマシンの必要性は益々高まるものと思われる。

データフローマシンは問題に内在する並列処理性を素直に反映し、且つ関数概念に基づくプログラムを効率的に実行するマシンとして、計算機アーキテクチャ上重要な意味を持っている。しかし、これを非数値処理の領域に適用する場合、データフロー方式の欠点の一つとされている構造体データ処理の問題を解決していかなければならない。

先に我々はVLSI技術を前提とし、Logic in memoryの概念に基づく構造体メモリの構成法を示し、この構造体メモリを用いたリスト処理向きデータフローマシンアーキテクチャの概要を示した。(1)　一般に、リスト処理に於いては並列処理性が低いという見方があるが、我々はまた文献(1)で関数性の遵守により、リスト処理に於いても高い並列処理性が得られることを示した。

本稿では先ず、Lenient consの概念(2)を用いることにより、活性化された関数間での処理がパイプライン化され更に高度の並列処理性が引出されることを示し、Lenient consの実現法について論ずる。　更に関数性を遵守したリスト処理に於けるデータコピーの問題にふれ、無駄なコピーをなくす方策を示す。

次に、Lenient cons及びリスト操作の効率化を実現する構造体メモリの構成法を示す。我々のマシンに於ける構造体メモリの設計思想は、メモリを多バンク構成にしconsに於けるセルの生成を全バンクに亘って分散・平均化させ、メモリアクセスネックの緩和を図るという点にあるが、ここではデータフロー方式に於けるメモリ分散化効果の簡単な評価を示す。

最後に、リスト処理に於いて、これらの機構が現実のプログラムにうまく機能するかどうかを評価するシミュレーションの問題について述べる。　シミュレーションの考え方、評価方法について論じ、この目的に沿うシミュレータの構成法を述べ、簡単なシミュレーション例を示す。

2．データ駆動制御によるリスト処理

本章ではデータ駆動制御によるリスト処理に於ける並列処理性，及びリストデータの効率的操作の問題について議論する。

データ駆動制御により得られる効果のうち、特に次の2点が大きな特長としてあげられる。

(1)　プログラム記述された問題の並列処理性を、低いレベル（プリミティブな演算）から高いレベル（関数の起動）迄、最大限に引出すことができる。

(2)　プログラム変数と副作用（グローバル変数の値の変更）の概念を排除した関数概念に基づくプログラムを効果的に実行することができる。

これらの特長はデータ駆動制御方式の持つ特性、即ち

(i) 関数の各引数の評価を並列的に行うことができる。(引数の並列評価)

(ii) 関数への引数値或いは関数値が一つでも求まれば、(全ての値が求まる迄待つことなく)直ちにその関数の実行を起動或いは再開させることができる。

(関数本体の部分的実行・再開)

によって裏付けられる。

ところで、一般にリストデータを動的に操作していく問題の場合、ある関数でリストデータが生成されている間、そのリストデータを使用する関数の実行は待たされるので、関数の部分的実行の効果が余り活かされないという問題がある。しかしこの問題は、Lenient cons の導入により解決され、リスト処理に於いてもパイプライン効果による高度の並列性が得られる。

またリスト処理に於いて関数性を遵守すると、古いリストデータの一部を変更する場合でも、そのデータのコピーを作る必要があるため、データ操作が非効率的になるという問題が生じる。しかしこの問題もデータ構造を変更する Append 操作を効率化することで解決することができる。

以下、リスト処理に於ける関数的処理性の問題とリスト処理の効率化について具体的に論ずる。ここで用いたプログラム例はデータフロー用高級言語 Valid (Value identification Language)[1][3] によって記述されている。

Valid は単一代入規則に基づき関数概念を徹底させた言語である。 所謂プログラム変数の概念はなく、プログラムは value定義と function定義からなる。特に value の定義では、value定義式

$[x_1, x_2, \cdots x_n] \Leftarrow$ < Expression >

により、複数 value を同時に定義できる。

また、式の集合である Block の概念を持ち、Block 中では Algol と同様の scope rule に従う local value の定義が許される。Block 中の式の評価順序は陽に示されない。Block もまた式であり、その生成する value は return 式によって示される。

本稿では、Valid の扱うデータタイプは Lisp のそれと同じ構造のリストタイプであるとする。また陽に定義されない関数は Lisp 1.5[4] で定義された関数を意味するものとする。

2.1 関数的処理と並列処理性

(1) 引数の並列評価

Valid で記述されたプログラムは等価的に純関数的表現に変換でき、また等価なデータフローグラフにおとすことができる。

例えば、全レベルのリストを反転させるプログラム1の block 2 の部分は等価的に

fulrev (cdr (x),
cons (fulrev (car (x), nil), y))

と書くことができ、データフローグラフでは図 2.1 のように表わされる。

関数 Fulrev は2つの引数を持っており、各引数 cdr(x) と cons(…) は並行して評価することができる。(図 2.1 に於いて、ノード(a)と(b)の実行が並行に行われる。) さらに、cons の2つの引数 Fulrev (…) とyが並列に処理される。

このようにデータ駆動による実行制御は関数の処理が内側から外側に向かって進められる。

(innermost evaluation)

(2) 関数本体の部分的実行

データ駆動の原理に従えば、命令(ノード)はそのオペランドが全て揃ったとき実行可能と

```
Program 1
 function fulrev(x,y)
 = case
   | null(x) => y;
   | atom(x) => x;
   | others  => begin                    ← block1
               |  u <= cdr(x);
               |  v <= fulrev(car(x),nil);   ← block2
               |  w <= cons(v,y);
               |  return fulrev(u,w)
               |  end
 end
```

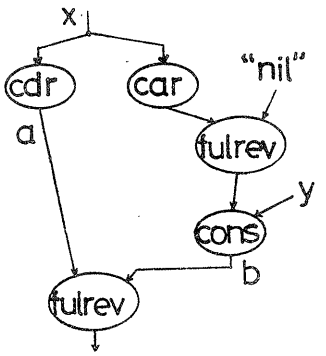
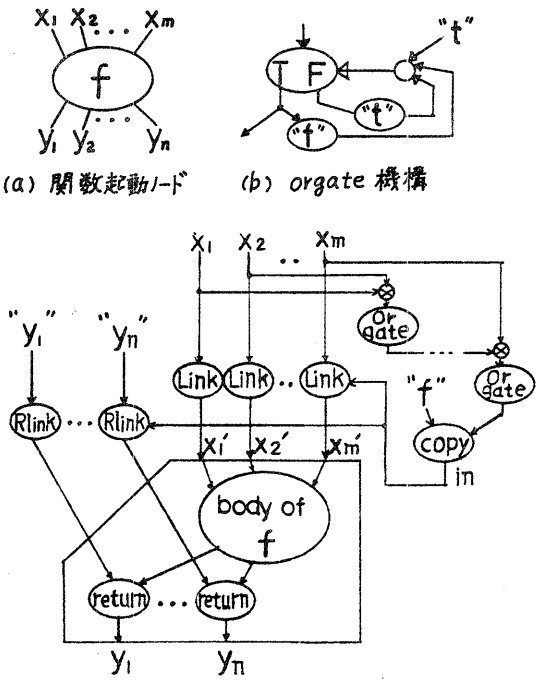


図 2.1 block 2 のデータフローグラフ

なる。データフロープログラム上では関数も一つのノードであり、命令実行の原理をそのまま適用すると、全ての引数の値が求まる迄関数の起動が待たされることになる。[5] しかし、引数の値が求まったものは直ちに関数本体側に送り、そのデータで実行できる部分を先に実行させておくようにすれば、関数起動の待ち時間をなくし実行の効率化を図ることができる。

図2.2は関数 $[y_1, y_2, \cdots y_n] \Leftarrow f(x_1, x_2, \cdots x_m)$ の起動メカニズムを示すデータフローグラフである。関数を起動する copy ノードは orgate ノードにより、トークン $x_1, x_2, \cdots x_n$ のいずれか一つが到着したときに発火し、新たに関数 f を活性化する。(orgate は T/F switch により図2.2(b)のように実現できる。)

活性化の準備が完了すると、トークン in (instantiation name) を Link ノードと Rlink ノードに伝える。Rlink ノードは直ちに本体側に値の戻し先情報 $y_1, \cdots y_n$ を伝える。Link ノードはそれぞれの引数トークン $x_1, \cdots x_m$ が到着すると直ちにそれを本体側に送る。本体では $x_1', \cdots x_m'$ にトークンが到着すると、それを待つ命令が実行可能となる。従って本体の実行はトークンが到着する度に部分的に進行していき、最後のトークンが到着したとき終了する。また各関数値 $y_1, \cdots y_n$ もそれぞれそのトークンが生成されると直ちに呼側に返され、呼側では返された値により直ちに実行を進行させていくことができる。



(a) 関数起動ノード　(b) orgate 機構

(c) 引数授受のメカニズム

図2.2　関数起動のメカニズム

2.2 Lenient cons とパイプライン効果による並列処理性

2.1で述べた特性はリスト処理に於いてもうまく機能するであろうか。プログラム2 (quicksort) の例について考えてみよう。

sort 本体中の partition の関数は3つの値 y_1, y_2, y_3 を生成する。y_1, y_2, y_3 を引数とする関数 sort, append は、それぞれ y_1, y_2, y_3 が生成されると直ちに起動されるので、関数実行の並列度が最大限に得られることが期待できる。しかしこのプログラムは、最良の場合には $O(n)$ の時間で sort を行うが、最悪の場合 $O(n^2)$ の時間を要する。

この原因は返される各値 y_1, y_2, y_3 の性質に依っている。即ち、y_1, y_2, y_3 の値は partition本体での append 操作が終了する迄生成されないために、次の sort 関数の実行もそれだけ遅れ、その待ち時間は append で作られるデータの長さに比例するためである。もし append で生成されるデータのうち、後方部分が append されている間、既に作られた前方部分を先に返してしまえば、その値を用いる実行を先行して進めることができる。

```
Program 2 (Quicksort)
 function sort(x)
 = if x=nil then x
     else begin
            y <= list(car(x));
            [y1,y2,y3] <= partition(cdr(x),y);
            return
              append(sort(y1),append(y2,sort(y3)))
            end;

function partition(x,y)
= if x=nil then (nil,y,nil)
    else begin
           [w1,w2,w3] <= partition(cdr(x),y);
           x1 <= car(x);  y1 <= car(y);
           return
            case
              x1=y1 => (w1,append(list(x1),w2),w3);
              x1>y1 => (append(list(x1),w1),w2,w3);
              x1<y1 => (w1,w2,append(list(x1),w3))
            end
          end
```

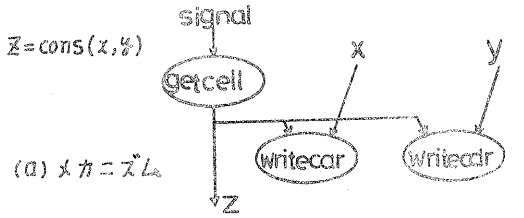
append操作はプログラム3に示す通り、consの再帰的実行である。

```
Program 3 (Conventional append)
 function append(x,y)
 = if x=nil then y
     else cons(car(x),append(cdr(x),y))
```

従ってcons実行にLeniencyを持たせることにより、この問題を解決することができる。

Leniencyとは次のような機能である。cons(x,y)の実行に於いて、オペランドx,yが到着する前にセルを1つ確保し、そのアドレスを値として先ず返しておく。セルのcar部，cdr部への値の書込みはオペランドx,yが到着した時点で行うこととし、オペランドx或いはyの到着が遅れても他方の値を用いる演算を先行的に実行させる。

Lenient consは次のように実現される。cons(x,y)は図2.3のようにgetcell,writecar,writecdrの3つの基本ノードに分解される。getcellノードはsignalトークンが到着すると発火して新セルを確保し、そのアドレスをconsの値を待っているノード及びwritecar,writecdrに伝える。セルにはgarbageタグの他、car部及びcdr部へのアクセスを制御するためのreadyタグが付随している。getcellではcar,cdrのreadyタグをオフにして、そこへのアクセスを禁止する。writecar,writecdrの各ノードはオペランドx,yが到着したとき発火して、そのオペランドをcar部又はcdr部に書込み、readyタグをオンにしてアクセス禁止を解除する。



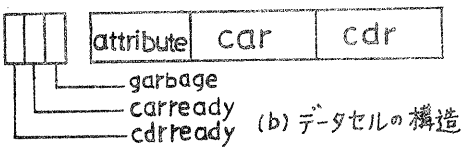


図2.3 Lenient consの実現

新セルのアドレスはconsの値として先に伝えられるので、そのセルへ値が書込まれる前（readyタグがオフのとき）に参照されることがあるが、この時は値が書込まれる（readyタグがオンになる）迄待たされることになる。

getcellの起動信号であるsignalトークンは、このconsを含む関数本体或いはブロックが起動されるときに生成される。

リスト処理に於いてLenient consの効果は大である。Lenient consの導入により、プログラム上陽に意識しなくとも、処理系は自然にStream処理[6],[7]の機能を実現する。

以下に例を示そう。プログラム4は素数を求めるプログラムである。primenumber(n)はintseq(2,n)が生成した数列(2,3,…n)をSieveして素数列を求める。Lenient consによりintseqは実質的には数列2,3,…nを順次Sieveに送っている。Sieveは送られた数列のうち先頭(car(n))を保持する。残り(cdr(n))のうちcar(n)で割切れるものをdeleteによって振い落し、その結果を再びSieveする。deleteで振い落されずに残った数列でも、順次Sieveに送られる。かくて図2.4に示すようにintseq,Sieve,deleteはおいに処理をパイプラインで実行していることになる。

```
Program 4 (prime numbers)
 function primenumber(n)
 = sieve(intseq(2,n));
 function sieve(n)
 = if n=nil then nil
     else cons(car(n),
               sieve(delete(car(n),cdr(n))));

 function delete(x,n)
 = if n=nil then nil
     else if remainder(car(n),x)=0
          then delete(x,cdr(n))
          else cons(car(n),delete(x,cdr(n)));

 function intseq(m,n)
 = if m>n then nil
     else cons(m,intseq(m+1,n))
```

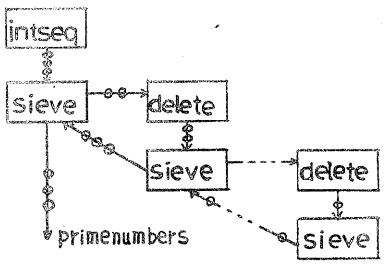


図2.4 素数計算に於けるStream処理

これはStream処理の概念そのものである。

先のquicksort（プログラム2）の場合、Lenient consを用いれば活性化されたsort, partitionの各関数がパイプラインで完全にオーバラップして実行されることになり、O(n)の時間でソートすることが可能となる。（図2.5参照，ソートに要する時間はトップレベルでのpartitionに要する時間に支配される。）

Lispインタプリタをデータフローマシンで実行させる（プログラム5）場合[1]について考えてみよう。ここでは、evlis関数で引数が並列に評価されるが、最後にこれらの評価値をconsしているため、次のapplyの実行が遅れるという問題があった。しかしながらこれもLenient consにより解決されることになる。

2.3 リスト処理の効率化

リスト処理ではcar, cdr, eq, atomによるリストデータの参照，consによる新リストの生成の他に、Append, Substitutionのようなリストを変更する操作が必要とされる。

このような操作でリストの変更を行う場合、関数性を徹底させた処理では必ずコピーが作られる。（例えばプログラム3のappend(x,y)ではxのコピーが作られている。）

```
Program 5 (evlis function in Lisp interpreter)
 function evlis(m,a)
 = if null(m) then nil
     else begin
            x <= eval(car(m),a);
            y <= evlis(cdr(m),a);
            return cons(x,y)
          end
```

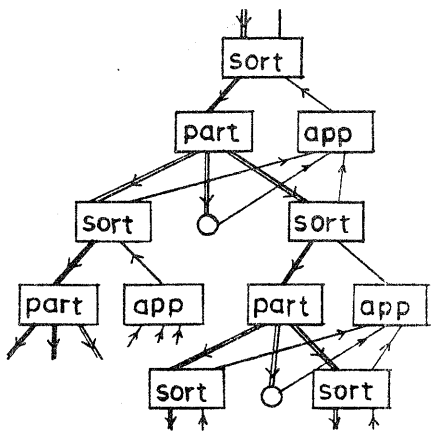


図2.5 sortingに於けるStream処理

一般に、Appendや Substitutionの操作を受けるリストの多くは他からの参照がないと予想される。このようなリストを全てコピーによって操作することは無駄なガーベジを多く作り、メモリ管理の負荷を増加させる原因となる。

この問題はrplacdの機能により、参照カウントを利用して無駄なコピーの生成をなくすことで解決される。rplacdは図2.6のように、resetcdrとwritecdrを用いて実現される。

resetcdr(x)はセルxのcdr部のreadyタグをオフにし、セルxの値をそのままrplacdの値としてrplacdの値を待つノード及びwritecdrに伝える。Appendはこのrplacdを用いてプログラム6のように実現される。

rplacdに於いてもLeniencyは満たされる。尚、rplacdはソフトウェア（ユーザ）からは見えないようにしておくことが必要である。即ち、ソフトウェアにはappendが基本関数として定義されているものとする。

3. マシン構成

本章では、構造体メモリを中心にリスト処理向きデータフローマシンの構成について述べる。

3.1 全体構成

マシンの全体構成を図3.1に示す。本マシンは実行制御部である多数のインストラクションメモリ（IM），IM間を結合するIM間通信

```
Program 6 (Efficient version of append)
 function append(x,y)
 = if x=nil then y
     else if refc(x)=1
            then rplacd(x,append(cdr(x),y))
            else cons(car(x),app(cdr(x),y))
   (* comment: refc(x) returns
      the reference count of cell x *);

 function app(x,y)
 = if x=nil then y
     else cons(car(x),app(cdr(x),y))
```

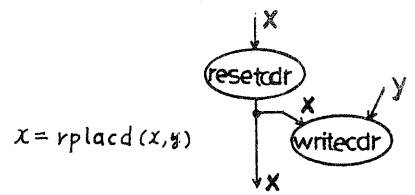


図2.6 rplacdの実現

ネットワーク，複数のバンクから構成される構造体メモリ（SM），及びIM－SM間を結合する調整ネットワーク（A-net）・分配ネットワーク（D-net）からなる。

インストラクションメモリはデータ駆動による実行制御を行う中核部であり，プログラムメモリ（PM），データメモリ（DM），命令取出し・結果分配機構より構成される。PMは読出し専用の連想メモリで実現され，プログラム（関数本体が格納される。連想機能は結果の分配先を検索することに用いられる。一方DMは到着オペランドを保持するバッファとして使われる。

IMは独立動作可能な複数個のブロックで構成される。各IMには関数プログラムが割付けられており（1つのIMに関数プログラムが複数個ロードされてもよい），IM間の通信はプログラム実行時に生起する関数間のCall，Returnを契機として行われ，Call/Returnパラメータ がIM間通信ネットワークを介して転送される。従ってIM間の論理的結合関係は木構造である。IM間通信ネットワークはプログラムの構造及びIMの負荷に応じて，論理的に動的な木構造を実現する。(1)

SMは多数個のバンクに分割され，各バンクには構造体操作専用の演算ユニットが埋込まれている。IMから送出された構造体操作命令パケットは，A-netでオペランドアドレスのデコードがなされ，対応するSMバンクへ転送される。D-netはSMから送出された結果パケットを受取ると，結果パケット中に保持されているIM番号を基に要求元IMを選び，そこへ結果パケットを転送する。

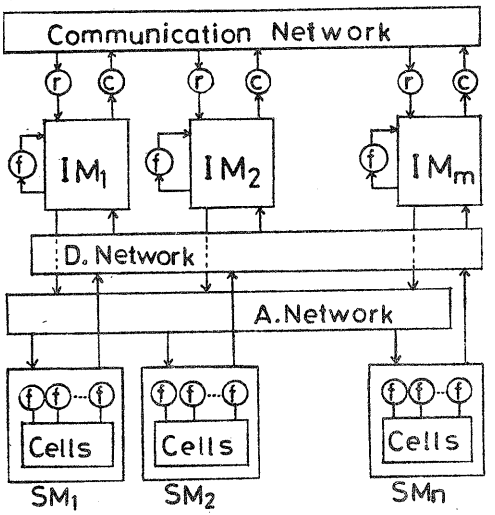


図 3.1 マシンの全体構成

A-net，D-netは各ステージにバッファを持ったn×nのroutingネットワークで実現される。

本マシンはIM－SM間の縦方向のパイプライン及びIM－IM間のパラレル実行により，高度の並列処理性を達成可能である。

3.2 構造体メモリの構成

本節では構造体データを効率的に操作する構造体メモリの構成法について述べる。

リストデータを主体とする構造体データを取扱う場合，ベクトルやマトリクスデータ等の規則的なデータ構造を扱う数値処理とは異なり，データの参照に関数対応の局所性があることは一般に少いと思われる。

このような場合，データをコピーして関数対応の局所性を持たせるか，又はコピーを行わずに参照を分散させるかが，アーキテクチャを決定する上でのトレードオフ点となる。

データ参照の性質はcons strategyに依存しており，ここではアドレス空間上均一化するように新セルを生成し，データ参照の分散化を徹底させることにする。

又，プリミティブなメモリオペレーションレベルの並列実行性を追求するため，メモリを独立動作可能な多数個のバンクで構成する。SMの各メモリバンクには，ガーベジコレクション用のcleanupモジュールが備えられる。

並列実行環境下でのガーベジコレクション法として，参照カウント方式を採っているが，参照カウント更新操作のネックを解消するには，メモリ中で演算できるLogic in memoryに近い機能が要求される。

図3.2にメモリバンクの機能構成を示す。

メモリバンクは更に，独立動作可能なtag，ref，attr，car，cdrのブロックに分割され，各ブロックにはそれぞれ専用の操作ユニットが設けられる。

car，cdrブロックは各々，atom表示タグ，readyタグ，ポインタフィールドからなる。

attrブロックは数値アトム，リテラルアトム

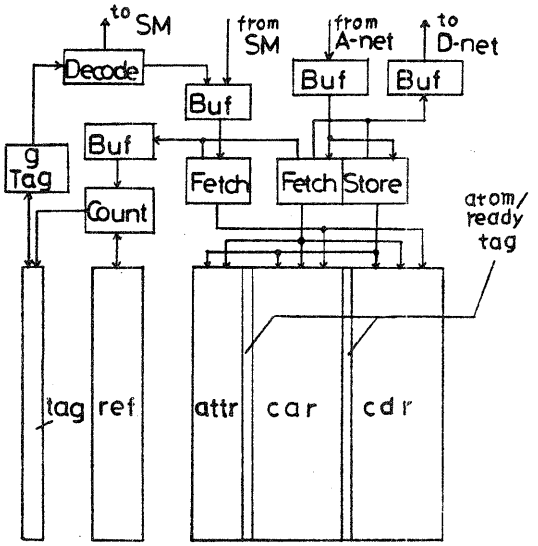


図3.2 メモリバンクの機能構成

等の属性を示すブロックである。

参照カウント更新のためのrefブロックは加減算回路組込みのRAMで実現され、アドレスデコード，メモリアクセス，及び加減算部のパイプライン化により、アクセスネックの解消を図る。

tagブロックは連想メモリで実現される。基本演算を実行した結果、セルXの参照カウントが0になると、tagブロック中のXの位置に要ガーベジコレクションという表示'g'が付される。cleanupモジュール内のget 'g'制御部は、基本演算とは独立に'g'表示のあるセルを読出し、そのcar,cdr部に対する参照カウントを1減じる。これでセルXは用済みとなるので'g'表示がリセットされ、代りに空きセルを示す'f'表示がセットされる。このときセルXに対応したcar,cdrの各readyタグはオフにセットされる。

Car操作 - car(x,d)を例にとって動作を説明しよう。(x:セルアドレス，d: 演算結果を待っている命令数)

CAR操作ユニットは命令キューから命令パケットを取出すと、セルXの参照カウントを1減じる指令をreference count制御部へ出すと共に、carブロック内のセルXのreadyタグを調べる。オフの場合(値が未到着)、再試行のため命令パケットを命令キューの最後尾につける。オンの場合、セルXの場所からデータZ(=car値)を読出し、結果を実行制御部へ送り返すと共にセルXの参照カウントをd増加させる指令をreference count制御部へ送る。

3.3 構造体メモリ分散化の効果

構造体メモリを多バンク構成し、バンクへのアクセスを分散・平均化したときの効果を簡単に評価する。

データフローマシンでは、パイプライン効果によりその動作は次のように特性づけられる。

(1) 命令メモリから構造体メモリへのアクセスは、構造体メモリからの応答とは独立に一定間隔(ta)で起こる。

(2) A-net, D-netでは各ステージ間のデータ転送はバッファを介してパイプラインで行われるので、そのスループットはステージ間の転送時間tbで規定される。

(3) 構造体メモリは演算機能を内蔵した構成のため、そのアクセス時間tcはメモリ容量mに対し $tc \propto m^g$ ($g \simeq 0.5 \sim 1$; 演算機能が高くなる程gは1に近づく)である。

図3.3に示すように、命令メモリをprocessor, 構造体メモリをMemoryとしてモデル化すると、システムのスループットは($tc > ta, tb$ と考えてよいので) tcに支配される。

Memoryの全容量をMとし、これをnバンクで構成するとしよう。メモリの分割化効果により、バンク内でのアクセス時間tcは

$$tc = C_1 (M/n)^g$$

で与えられる。

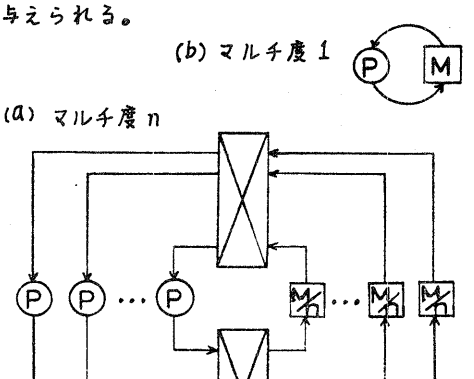


図3.3 データフローマシンのモデル化

Processorもn個あるとすると（nをマルチ度と呼ぶ）、n個のProcessorからのアクセス要求が競合し、その分だけアクセス時間が引延ばされる。今、あるバンクへd個のアクセス要求があるとすると、アクセス時間は

$$t = C_2 d \cdot t_c$$

で与えられる。

ここでマルチ度nの時の性能をP_nとして、$\eta = P_n / P_1$を求めよう。P_nはMemoryのアクセス時間に支配されるから、メモリアクセスの並列化効果を考慮して、

$$P_n = n/t = \frac{1}{C_1 C_2} \cdot \frac{1}{d} n \left(\frac{n}{M}\right)^{\beta}$$

また$n=1$のとき（図3.3 (b)）

$$P_1 = 1/t = 1/C_1 C_2 M^{\beta}$$

$$\therefore \eta = \frac{n^{1+\beta}}{d}$$

さて各Processorからのアクセス要求の発生は一様に分布し、且つMemoryバンクへのアクセスに偏りがないものとすると、ある時点に於いてn個のProcessorはMemoryへ$1/n$の確率で一斉にアクセス要求を出すことになる。
あるMemoryバンクについて考えると、要求がj（$0 \leq j \leq n$）である確率は次式で与えられる。

$$P_r(d=j) = {}_nC_j \left(1-\frac{1}{n}\right)^{n-j} \left(\frac{1}{n}\right)^{j}$$

従って、あるProcessorがあるMemoryバンクへ要求を出したとき、そのバンクの競合がjである確率は

$$P_{d\neq 0}(d=j) = \frac{P_r(d=j)}{1-P_r(d=0)} .$$

Memoryバンクでの平均競合数は

$$\begin{aligned} d &= \sum_{j=1}^{n} j \cdot P_{d\neq 0}(d=j) \\ &= \sum_{j=1}^{n} j \cdot \frac{P_r(d=j)}{1-P_r(d=0)} \\ &= \frac{1}{1-P_r(d=0)} \\ &= \frac{1}{1-(1-\frac{1}{n})^n} \end{aligned}$$

で与られる。よって、

$$\eta = n^{1+\beta}\left(1-(1-\frac{1}{n})^n\right)$$

$$\lim_{n\to\infty} \eta = n^{1+\beta}\left(1-\frac{1}{e}\right)$$

となり、マルチ度nのときの性能は$n^{1+\beta}$（$n^{1.5} \sim n^2$）に比例し、その効果は大きい。

4. シミュレータ

本章では3章で提案したリスト処理向きデータフローマシンのソフトウェアシミュレータについて述べる。

4.1 シミュレータ作成の目的

我々のソフトウェアシミュレータ作成の目的は次の点にある。

(1) データフロー言語を実行する仮想マシンを提供し、この上で関数型言語の開発及び関数型プログラミング方式の確立を行う。

(2) データフロープログラム自身の特性を解明する。

(3) データフローマシンを構成するための要求条件を明確化し、提案したデータフローマシンアーキテクチャの有効性を検証する。

従ってシミュレーション法として、GPSS等を用いた待ち行列モデルのシミュレーションといった統計的解析手法は採っていない。
ここではデータフローマシンシステムをソフトウェアで提供することにし、データフロー機械語を解釈実行すると共に、キュー長，資源利用率，スループット等のマシン特性情報を収集するインタプリタ方式を採った。

4.2 評価項目

表1,2にシミュレータの評価項目を示す。プログラム特性情報はデータフロープログラムの特質をみるものであり、シミュレーション上では資源が無限に存在し且つ競合による遅延等のない理想マシンの特性である。

一方、マシン特性情報は実際のシステムに即して各構成諸元を与えた場合のシステム性能情報である。構成条件及び諸元をパラメータとしてふらすことにより、マシン特性がプログラム特性からどの程度ずれるか、又どのように変化するかをみる。このようなシミュレーションを通じて最適なマシンアーキテクチャを追究することにする。

表 1.　プログラム特性情報

評価項目	収　集　データ
並列度	単位時間あたりの命令実行数（命令レベル） 単位時間あたりの活性化関数の数（関数レベル）
命令頻度	演算命令，制御命令，関数リンケージ命令の頻度・割合
行先数分布	命令コードに陽に書かれた行先数の分布
参照の局所性	SMバンクに入るオペレーションのオペランドアドレスパターン

表 2.　マシン特性情報

評価項目	収　集　データ
並列度	同上（但し，有限資源下で）
IM/SM分割効果	IM(SM)をブロック分割したときの活性化命令取出し数 （関数本体のローディングの仕方にIMの特性は依存する）
参照の局所性	SMバンクに入るオペレーションのオペランドアドレスパターン及び各バンクの負荷 （cons strategyと参照アドレスパターンの相関をみる）
参照カウントトラヒック	各メモリバンク内で生起する参照カウント更新要求数
処理時間 スループット	トータルクロック数 処理件数
資源利用率	IM,SMのメモリ使用量 資源稼動率，各キュー特性

4.3　シミュレータの構成

図4.1に実行制御部IMの動作をシミュレートするKernel interpreterの基本構造を示す。Kernel interpreterはIM管理部，FU(演算エニット)管理部，ネットワーク管理部，SM管理部及びモニタ部から構成され，以下の機能を持つ。

1) 実行可能命令の取出し，結果パケットの分配などの実行制御
2) 構造体操作を含む基本オペレーションの実行
3) 構造体メモリの基本動作の模擬
4) IM間通信制御
5) クロック及び状態管理
6) プログラム及びマシン特性情報の収集

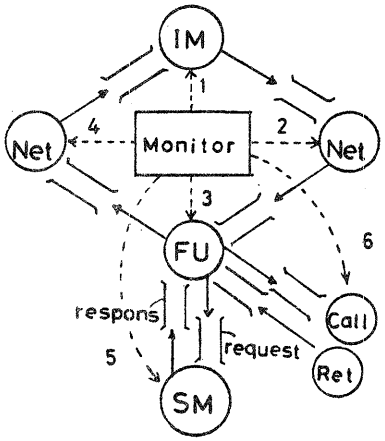


図4.1　シミュレータの基本構造

Kernel interpreterのモニタ部は図4.1に示す如く，①～⑥の順に巡回サービスを行い，各種キュー操作，クロックの更新等を行う。

本シミュレータでは，キュー長，キューからの取出し個数等のパラメータをふらせることにより，次に述べる無限資源モデル，有限資源モデルの両方の取扱いが可能である。

a) 無限資源モデル

キューの長さは無限とし，各キューからの取出し個数，プロセッサの個数にも制限を加えない。従って実行可能な命令はすべて取出されて実行されることになり，プログラム自体が有する並列度といったプログラム特性情報が得られる。

b) 有限資源モデル

i) キューの長さは無限。但し各キューからの取出し個数は有限とする。この場合，blocking現象は生じない。

ii) キューの長さは有限とし，各キューからの取出し個数にも制限を加える。キューがオーバフローした場合は，各キューからの取出しをblockする。

有限資源モデルは実際のマシンを反映しており，各キューの構成諸元を設定することによりマシン特性情報が得られる。

IM－IM間及びIM－SM間の交信動作はFU管理部内のCall/Returnキュー及びSM request / SM responseキューを介して行われる。この場合、処理がIM内に閉じていないため何時の時点で演算が終了するかは未知である。従って、パケットには演算終了時刻ではなく、到着時刻を記入しておく。キュー内のパケットのうち、現時刻より以前に到着していたパケットが取上げの対象となる。

IM，SM相互間の実行動作をシミュレートする場合もIM内と同様に行われる。即ち、活性化されたIMの分だけ順次巡回サービスが行われ、最後にSM内のサービスが行われて、1クロック分のシミュレートが完了する。

以下に、Kernel interpreterの基本構造の詳細を説明する。

(1) 関数本体の管理

関数本体の構造を図4.2に示す。関数本体はIarea (instruction area), Oarea (operand area), Sarea (successor area)から構成される。

連想記憶で実現されるPMの動作をシミュレータで模擬するとき、実際にサーチしなくてもよいように、結果を待っている命令 (successor)をチェインしておくことにする。このため関数本体にSareaを追加する。

Iareaは Icell (instruction cell) の格納領域であり、Oareaは結果の値を保持するバッファ領域である。結果のパケットが送られると、

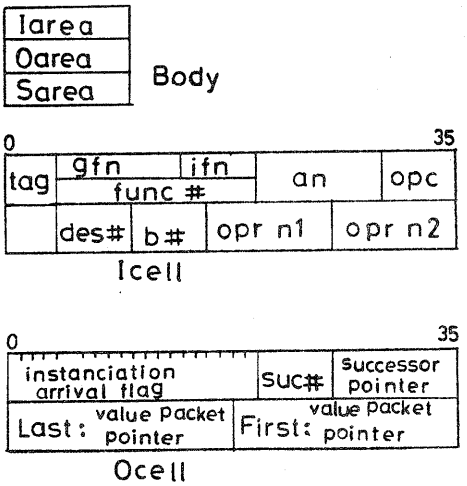


図4.2 関数本体の構造

結果の名前(an)に対応したOcell (operand cell)にvalueパケットがチェインされる。Sareaは Scell (successor cell)の格納領域であり、Ocellから指されている。

(2) 命令取出し機構

EQ (enabled instruction queue)は活性化された命令を保持するキューである。図4.3にEQの構造を示す。命令メモリがnブロックに分割されることを反映して、EQ自体もn個のブロックキューで構成される。従ってブロック毎に、活性化された命令が繋がれることになる。

EQは命令メモリの読出し時に使用され、取上げられた活性化命令は次のAQ (arbitration queue)へ送出される。

(3) 結果の分配機構

MQ (instruction memory queue)は命令メモリへの分配を待っている結果パケットのキューであり、IM内のブロックの個数だけ用意される。結果パケットには関数名(gfn)とIM内での関数識別番号(ifn)からなる関数番号が付される。ifnは関数ロード時に付与され、以後関数はこの番号で管理される。関数は複数のブロックにまたがってロードされてもよいので、分配先決定のための関数－ブロック対応表を持つ。

(4) 構造体メモリの管理

図4.4にSM関連のキュー構造を示す。各SMバンクに対し、UQ (used cell queue), GQ (garbage cell queue), RQ (reference count queue), BQ (memory bank queue)が用意される。

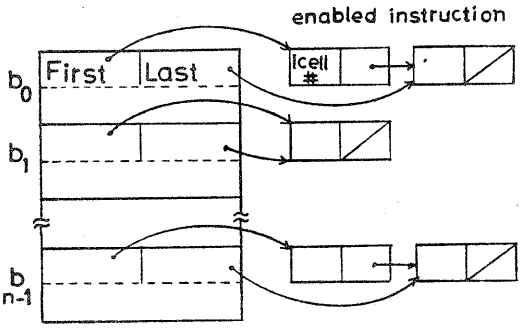


図4.3 EQの構造

UQは使用中のセルを、GQはくずセルの管理を行うためのものである。

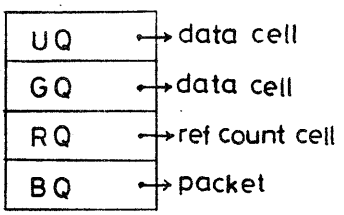
RQは参照カウント更新要求を保持しておくキューで、参照カウント更新の際のトラヒックを測定するため、自メモリバンク内で発生した更新要求と他メモリバンクから来た更新要求とを区別して計測する。

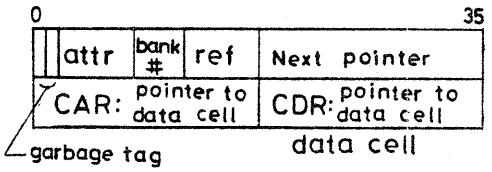
BQには構造体操作の命令パケットが保持されている。各BQでは、現時刻より以前に到着したパケットのうち、独立に動作しうる命令が複数個取出され、指定された基本オペレーションの実行が行われる。

例えば Car(x,d) の操作は次のようになる。データセルxを読出しcar値を得る。次に参照カウント更新要求として、$count_{-1}$,x (1減少), $count_{+d}$, car val (d増加) の2個の参照カウントセルをつくりRQにつなぐ。

Cons命令で新セルを作る際、シミュレータでは次の4つの方策をとることができる。

① 最初のバンクから空セルがなくなる迄集中的にとり、なくなれば次のバンクに移る。
② car部になるセルのバンクからとる。
③ cdr部になるセルのバンクからとる。
④ mod(i,n) でバンク番号を定めて、そのバンクからとる。(i:i回目の cons, n:バンク数)





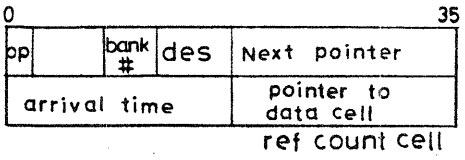


図4.4 SM管理表の構造

4.4 シミュレーション例

Divide and Conquer 方策による階乗計算(プログラム7)を例題として行ったシミュレーション結果を示す。シミュレーションでは①IMブロック分割化効果, ②単一資源下でのパイプライン効果を調べた。

① IMブロックの分割化効果

図4.5(a)のような無限資源モデルを仮定し、各活性化関数毎にIMブロックを割付けた場合の各関数の活性化状況を図4.6(a)に示す。このモデルでは各関数は完全に並列処理される。図より、このときの処理時間は約500クロックである。またその処理時間は図4.7に示すように、「log n」に比例している。一方、図4.5(b)のように全関数を1IMブロックに割付ける単一資源モデルでは、その処理時間は約700クロックとなった。このモデルでは関数レベル, 演算レベルの並列実行が混じり合い、その効果はIMブロックでの命令取出し(1命令/1クロック)に制約されることになる。

さて、両モデルでの処理時間比は5:7であり大差ない。このことはデータフロー制御により図4.5(a)の各段がパイプラインで動作し、関数, 演算両レベルの並列実行性を発揮していることを示している。ちなみに単一資源モデルの場合、各段の平均稼動率は0.74であった。(無限資源の場合平均稼動率は0.1であった。)

```
Program 7
 function factr(m,n)
 = if m=n then n
     else factr(m,quotient(m+n,2))*
          factr(quotient(m+n,2)+1,n);
 function factrorial(n) = factr(1,n)
```

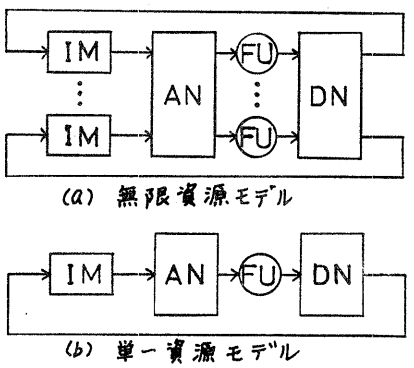


図4.5 シミュレーションモデル

② 単一資源下でのパイプライン効果

図4.5(b)のような単一パイプラインではFUの処理速度が他に比して遅く、ここでのネックが予想される。そこでFUの並列化の効果を調べると、処理時間と各資源の平均稼動率は図4.8及び図4.9のようになり、直観と一致する。図でパラメータγの値はFUと他の資源との処理速度比である。γ=4のときの処理時間及び稼動率は5台で飽和している。このときの稼動率0.59とγ=1のときの稼動率0.74との差は階乗計算の始めと終りの部分で並列性がなくなるために生じるパイプの切れによる影響である。

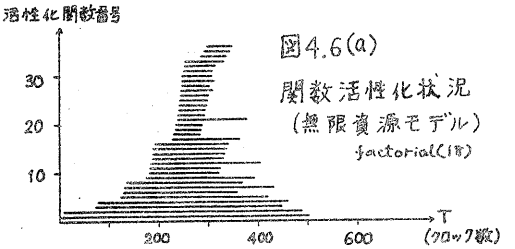


図4.6(a)
関数活性化状況
(無限資源モデル)
factorial(18)

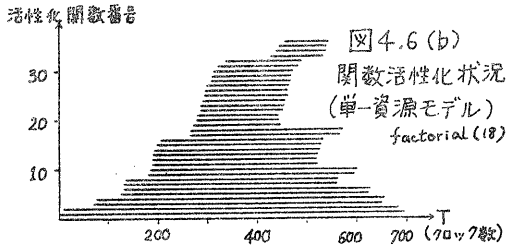


図4.6(b)
関数活性化状況
(単一資源モデル)
factorial(18)

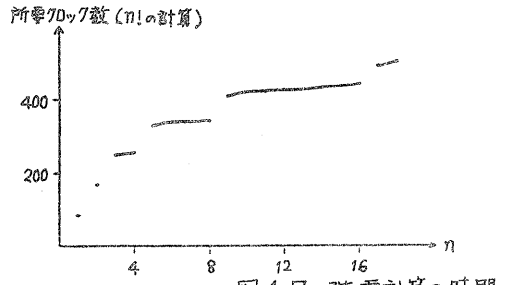


図4.7 階乗計算の時間

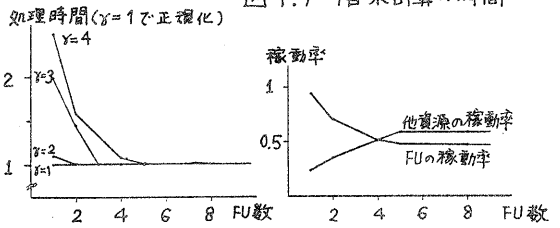


図4.8 FU並列化効果(1)　図4.9 FU並列化効果(2)

5. あとがき

リスト処理向きデータフローマシンのアーキテクチャに関して、先ずリスト処理と並列処理性の問題を考察した。Lenient consの概念を導入することにより、リスト処理そのものがStream処理と等価になることを示し、Lenient consの実現法を明らかにした。次にLeniencyを保存しつつ、無駄なコピーを排除するappend機構の実現法を示した。続いてLenient cons機構を盛込んだ構造体メモリの構成法を述べ、簡単な解析を通じ構造体メモリを多バンク構成することの有効性を評価した。最後にデータフローマシンのシミュレーションについて考察し、統計的解析手法に基づくシミュレータではなく、実際のプログラムをインタプリトするシミュレータの構成を述べた。

現在我々は文献(1)及び本稿で論じた並列処理効果が理論通り生かされるか否かをみるために、本稿で述べた立場でシミュレーションを進めているが、シミュレーション結果及びアーキテクチャの評価については稿を改めて論ずる。

最後に日頃叱咤激励して下さる山下紘一第一研究室長、並びにアーキテクチャ研究グループの諸氏に感謝する。

参考文献

(1) 雨宮,長谷川,三上：リスト処理向きデータフローマシンの検討，情処学会記号処理研究会 13-3，1980
(2) Friedman,D.P. and Wise,D.S. : CONS should not evaluate its arguments. Automata, Languages and Programming. S.Michaelson and R.Milner,Eds. Edinburgh Univ. Press, 1976.
(3) 雨宮：データフローマシン用高級言語Validの設計思想，56年度信学会総合全国大会，No.1486，1981(予定)
(4) McCarthy,J. et.al. : Lisp 1.5 programmer's Manual. The Mit Press, p.106, 1966.
(5) 長谷川,三上,雨宮：連想記憶を用いたデータフローマシンの一構成法，信学会電子計算機研究会，EC79-55，1980
(6) Dennis,J.B. and Weng,K.S : An Abstract Implementation for Concurrent Computation with Streams. Proc. of 1979 International Conference on Parallel Processing. pp.35-45, 1979.
(7) Arvind, Gostelow,K.P. and Plouffe,W.: An Asynchronous Programming Language and Computing Machine. Report TR114a, Univ. of California, Irvine, 1978.