2進木構造並列処理システムCORALによる関数型プログラムの処理

Processing of Functional Program
by Binary Tree Parallel Processing System CORAL

西山 和義　山根 義孝　高橋 義造
Kazuyoshi NISHIYAMA　Yoshitaka YAMANE　Yoshizo TAKAHASHI
徳島大学 工学部
Faculty of Engineering,University of TOKUSHIMA

1 はじめに

関数型の言語は、従来多く用いられてきた逐次型の言語に比べて、計算の機構の簡明さ，記述力の高さ，等の利点を持っている。しかしながら，関数型の言語を、従来の計算機で処理しても言語の特性を生かすことができず，逐次型の言語に比べて十分な効率が得られない。

ところが，新しいアーキテクチャである並列処理システムにおいては、逐次型での表現は、その言語の構造的な制約により十分な並列性を示すことができないが、関数型の言語には、この制約がなく、十分に並列的な表現ができる。

さらに、関数型の特徴である，副作用がなく、関数の実行順序はデータの入出力のみに依存することが、実際の並列処理に対しても非常に有効である。

関数型言語を，木構造の並列処理システムで処理するのを目指して，Keller，Magoによる研究が行なわれている。

偏微分方程式の超並列処理を目指して徳島大学で研究中の，2進木構造マルチプロセッサCORALは、その構造から考えて，特に関数型言語の並列処理に適していると思われる。そこで，このCORAL上で，関数型言語FPで書かれたプログラムを並列処理するシステムを作成し，CORALプロトタイプ上で、行列の乗算を行なうFPプログラムを並列処理して，このシステムによる並列処理の効率について評価を行なった。

2 CORAL[5]

CORALは，数百，数千台のプロセッサを用いた，2進木構造をそのアーキテクチャとする，高次並列システムである。各ノードは，プロセッサとメモリよりなり，各ブランチは，双方向のデータバスより構成されている。(図1)

2進木の根にあたるプロセッサは，Root Processor（RP），と呼び Control Processor（CP）として働く。他のプロセッサは、Node Processor（NP），Leaf Processor（LP）と呼ばれる。ルート方向を上方向，リーフ方向を下方向と呼ぶ。

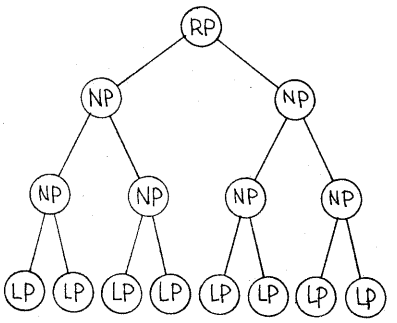

図1 CORAL

2.1 CORALプロトタイプ

CORALにおける並列処理プログラムの検証や，効率，性能の評価用に開発されたシステムで、現在15台のプロセッサより構成され，稼動中である。

昨年までに開発されていたシステムは，NP

LPに，TK－80，TK－85，Ex－80と，3種類のプロセッサを用いていたため，プログラムを作成する場合や，性能を評価するために，それぞれのプロセッサの処理能力や，I/O空間の違いを考慮する必要があった。そこで今年度，NPをすべてTK－85とし，各プロセッサでの処理能力をほぼ等しいものとした。（写真1），またプロトタイプでの通信の状態を表示する，表示装置を製作した。（写真2）これにより，各プロセッサ間での通信状態を視覚的にとらえることが可能となり，ディバッグやアルゴリズムの検証，さらに並列処理の効率の観測などに役立っている。

現在 RPはソードM223Ⅱ（CPU Z－80，メモリ 64KB，I/O装置 フロッピーディスク，ディスプレイ，プリンタ），NP，LPは，NECのTK－85シングルボードコンピュータ（CPU i8085，メモリ 16KB）より構成されている（図2）

プロセッサ間結合は，I/Oインターフェイス方式で i8255（Programmable Peripheral Interface）を使い，割り込み制御では i8259（Programmable Interrupt Controller）を用いている。転送速度は 10KB/s とおそいが，アルゴリズムの検証に使用するというプロトタイプの使用目的からいってさしたる支障はない。

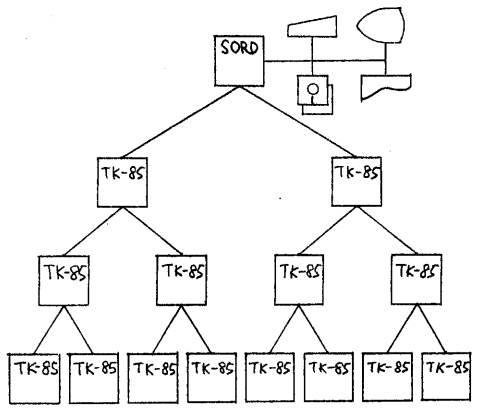

図2 CORAL プロトタイプ

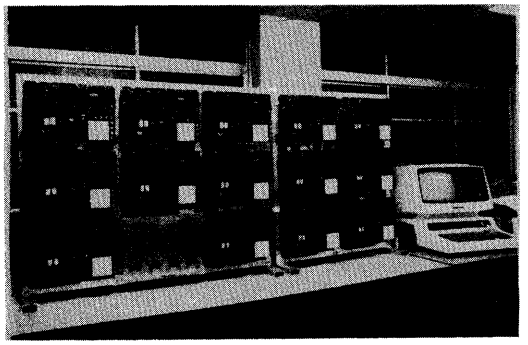

写真1

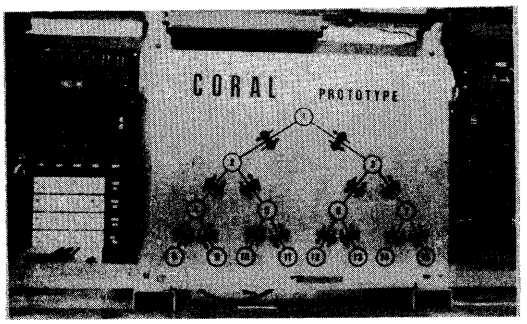

写真2

3 FP言語[1]

関数型言語は、並列処理を行なう上で、次の様な利点を持っている。

(1) 副作用を持たない。

関数の実行は，本質的には，入力データを，出力データに変換することであって，その結果は入力データのみに依存し，処理の環境や，過去の結果には影響されない。

(2) 関数の実行順序は，データの入出力関係にのみ依存する。

(3) 階層的な表現により，処理の大きさに関係なく，同様の関数として表現できる。

(1)により，並列処理時に，各関数は，他の関数の影響を受けることなく独立して処理することができる。

(2)，(3)の特徴により，関数型言語で書かれたプログラムは，さまざまな大きさの並列処理を十分に表現することができる。

3.1 FP言語

FP言語は John Backus により提案された関数型の言語である。その特徴は、強力な関数形式と、関数のタイプを、1入力1出力としたことにより得られた変数のないプログラム方式をとることである。

変数がないことは、並列処理時に必要となる複雑な、変数－値の対応づけや、各変数の参照により発生するデータ通信、あるいはコピーの作成といった処理が不要となる。また関数形式のいくつかは、直接的な並列処理の可能性を示す。

FP言語は、仕様が小さく、そのままで、実用的な問題を解決するのに十分な表現力があるかは問題であるが、上に述べた様に、並列処理を記述するのに適した面がある。。そこで今回は、このFP言語で書かれたプログラムを実行するシステムを作成した。

言語仕様は文献〔1〕に書かれているFPのなかから、2,3の原始関数をのぞいたサブセットとした。仕様について簡単にのべる。

a 原始関数

```
selection,tl,id,atom,null,eq,distl,
distr,trans,+,-,*,/,and,or,not,apndl
apndr,tlr,rotl,rotr,length,less,greater
```

b 関数形式

```
(i)   Composition
   (f:g):x ≡ f:(g:x)
(ii)  Construction
   [f1,f2, ..,fn]:x ≡ < f1:x,f2:x, ..,fn:x >
(iii) Condition
   (p -> f;g):x ≡ (p:x)=*t* → f:x
                  (p:x)=*f* → g:x
(iv)  Constant
   'x':y ≡ x
(v)   Insert
   (in f):< x1,x2, ..,xn > ≡
                  f:< x1,(in f):< x2, ..,xn >>
(vi)  Apply to all
   (aa f):< x1,x2, ..,xn > ≡
                  < f:x1,f:x2, ..,f:xn >
(vii) While
   (while p;f):x ≡
          (p:x)=*t* → (while p;f):(f:x)
          (p:x)=*f* → x
```

x,y ：対象パラメータ

f,g,p：関数

関数形式では、Construction, Apply to all が並列処理可能であることが明である。

以上の様な、関数、関数形式を用いたプログラムの例を示す。

```
def ip = (in +):(aa *):trans endef    (1)
def mm = (aa (aa ip):distl):distr
                 :[ 1,trans:2] endef   (2)
```

(1) は 内積を求めるプログラム

(2) は 行列の乗算を求めるプログラム

4 関数型言語の並列処理方式

関数型言語を並列処理する方式として種々の方式が考えられる。マスター・スレーブ方式は、1つあるいは、複数のプロセッサが、並列処理の可能性を発見し、その実行を他の複数の処理プロセッサへ分配し、その結果を受けとるという方式である。この方式では、並列度が高くなるとプロセッサ間通信がネックとなる。[7]

これに対して階層方式がある。この方式としては木構造の並列処理システムを用いた Keller Mago 等の研究がある。

4.1 Keller マシン[4] の処理方式

このシステムの特徴は、リーフにあたるプロセッサが、実際の処理を行なう、他のノードは通信用のネットワークを構成している。

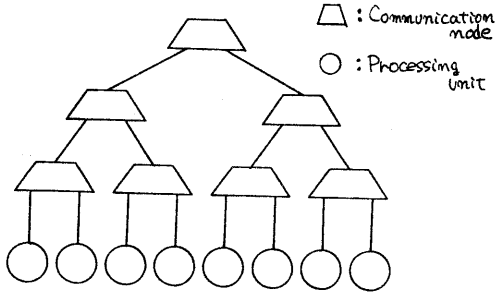

図3 Keller マシン

このシステムでの処理は、使用される言語から考えられる、副作用と、並列処理時にネットワーク上を流れる通信の管理に問題があると思われる。

4.2 Mago マシン[2]の処理方式

このシステムは、リーフのプロセッサが、1シンボルづつの情報を持ち、FP言語を直接実行する。木の部分部分に、かなり多数の独立した処理が行なえるが、実行結果を再びリーフへ移動するため、かなり多量の複雑な通信が行なわれる。

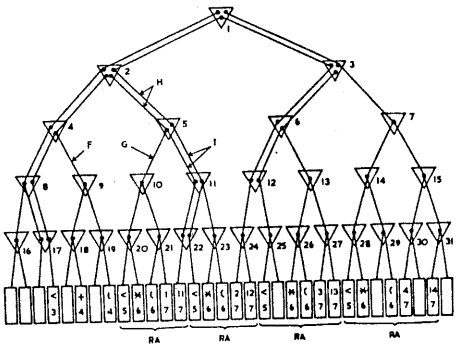

図4 Mago マシン

4.3 CORALの処理方式

以上の2例は、システムの木構造を生かして関数型の言語を並列処理するシステムであるが通信の制御等の面でかなり複雑な処理が必要となると思われる。

ここで、CORALの特徴の1つである、すべてのプロセッサが同じ機能を持つ、という点に注目すると、次の様な方法で関数型言語の並列処理の可能性が考えられる。

CORALのすべてのプロセッサで次のような処理を行なう。

1. 上方向のプロセッサよりプログラムとデータを受け取る。
2. 受け取ったプログラムを実行する。
3. 実行時に並列処理の可能な部分に出会ったならば、下方向のプロセッサへ分割した処理をわたす。ただし、プロセッサがリーフならば、その処理はすべてこのプロセッサで実行する。
4. 結果を下方向より受け取る。
5. さらに実行可能であれば、3.へもどる。
6. 結果を上方向へ返す。

最初にルートのプロセッサへ、データとプログラムを渡すことにより、そのプログラムの並列処理が開始される。

この処理を行なう時の、システムにおける通信は非常に簡単となる。1つのブランチについて見れば、必ず、上方向から下方向、次に下方向から上方向の順で行なわれ、それ以外の通信の流れは発生しない。

一般に、並列処理を行なう場合は、効率よく通信等を行なうために、OSが必要な場合が多いが、上に述べた様な処理の流れを考えると、OSは、非常に簡単なものでよく、あるいは、OSがなくても十分な処理を行なうことができると思われる。このことは、特にプロセッサ数が非常に大きいシステムを考えると利点であるといえる。そこで今回作成したシステムでは、いままでに開発したCORALプロトタイプ用OSの効率が十分でないこともあって、OSなしで動く様にした。

5 CORAL-FP システム

CORAL-FPは、今回作成した、CORAL上で、関数型言語FPを直接実行するシステムである。

5.1 プログラムの構成

CORAL-FPは、システムコントロール

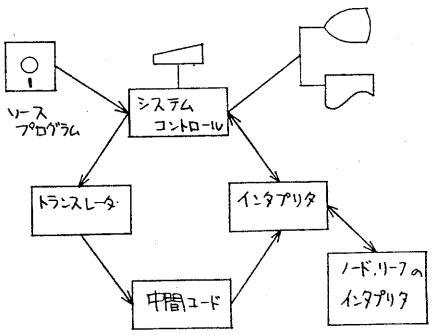

図5 プログラムの構成

トランスレータ，インタプリタの3つのプログラムよりなる。（図5）

システムコントロールは，入出力を管理する。ディスクファイルより，FPで書かれたリースプログラムの読み込み，トランスレータ，インタプリタの起動，処理結果の出力を行なう。

トランスレータは，インタプリタでの処理を軽減するために，プログラム，データを中間コードへ変換する。プログラムの中間コードは，変換後すべてのプロセッサへ送られる。データの中間コードは，ルートプロセッサにのみ置かれる。この中間コードはインタプリタでの内部表現である。

インタプリタは，トランスレータにより得られた，中間コードに従い，FP言語の関数を実行する。インタプリタは，すべてのプロセッサにまったく同じものが置かれる。

5.2 関数の内部表現

インタプリタでの処理が簡単に行なわれるように，プログラムとデータは内部表現に変換される。プログラム，即ち関数の内部表現について述べる。

原始関数は 1 または2バイトのコードで表わされる。ユーザーにより定義された関数も1バイトのコードで表現される。

0 0 0 1 x x x x	原始関数
0 0 1 0 x x x x	関数形式の区切
0 0 1 1 x x x x	関数形式の終り
0 1 0 0 x x x x	関数形式の始まり
0 1 1 1 1 1 1 1	プログラムの終り
1 x x x x x x x	定義された関数

図6 関数等の内部表現

関数形式は図7の様に，関数形式の種類を示す1バイトのコード，関数の並び，関数の区切り，関数形式の終りを示すコードより構成される

$(f_1 \rightarrow f_2 ; f_3)$

⇒ [CND | f_1 | | f_2 | | f_3 | END]

図7 関数形式の内部表現

定義された関数は，その定義された順にコードが与えられる。このときに図8のように，コード順に，プログラムの本体の在る場所を示すテーブルが作られる。これもプログラムと一緒にすべてのプロセッサへ送られる。

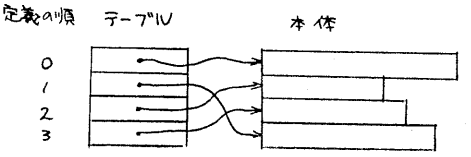

図8 定義された関数のテーブルと本体

5.3 データの内部表現

今回作成したシステムでは，データはアトムと列と呼ばれる2つのタイプがある。

FPの原始関数の多くのものは，データタイプ列の書き替えを行なう。このとき列の入れ子の深さがわかっていると便利である。そこでアトムを次の様に表現する。（図9）

Levelにより入れ子の深さを表現し，headでアトムの種類を示し，bodyがその内容を示す。

アトムの構造

Level	head	body
1B	1B	0〜8B

アトムの種類

	head	body
空(%0)	00	なし
論理(%t*, %f*)	1*	なし
数	20	4バイト
文字列	3N	Nバイト

図9 アトムの表現

列の表現は，列のレベルを示す部分と1バイトの列の先頭を示すコードよりなる。またデータ全体の終を示すコードがある。（図10）

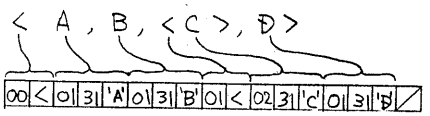

図10, 列の内部表現

5.4 プロセッサ間通信方式

CORAL-FPでの通信は，図11のようなフォーマットのパケットを交換することによ

て行なわれる。

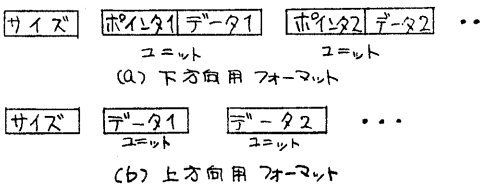

図11　通信用フォーマット

下方向用のフォーマットは，サイズは送るユニット数，ポインタはすでに送られているプログラム上での実行開始場所を示すポインタである。データは、ポインタからの処理に対応し、そのまま実行できる様に、レベル等が整理されている。ユニット，即ちポインタとデータの一組が通信の基本単位である。

上方向用のフォーマットでは，ポインタが省略される。またデータは，関数実行により得られた結果のレベルより1だけ大きなレベルとなる。これは下方向用のフォーマットで，データのレベルを操作したのに対応した処理である。

各ユニットは、可変長で送られるのでこの処理を簡単にするために、送信側で必要なだけ通信を行なうと，割り込みを行ない，受信側にそのユニットの終りを示すことにした。

5.5　インタプリタの機能

インタプリタは各プロセッサに、まったく同じものが置かれる。インタプリタの構成は、図12の様になっている。通信ハンドラーは、プロセッサ間通信に関する処理を、関数実行部は原始関数の実行や定義された関数の呼び出しを行なう。関数形式実行部は、関数形式の処理を行なう。ここでプログラムの並列性を発見し、並列処理を行なう。

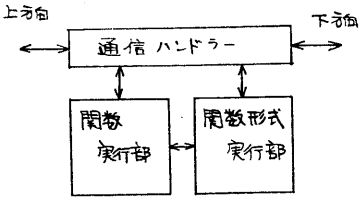

図12　インタプリタの構成

5.5.1　関数形式の並列処理

CORAL-FPシステムでは、関数形式に応じて並列処理を行なう。Apply to all, Construction は並列処理が可能であり，また Condition, Insert も実行方法を検討すれば，並列処理が可能であることがわかる。

Apply to all，Construction は，実際には，次に示す様に行なわれる。

(aa f):<d1, d2, d3>

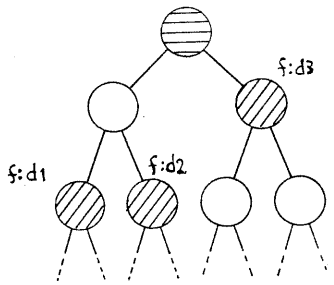

図13．Apply to all の処理

インタプリタは，関数形式 apply to all の発見により、データ内に含まれる，レベル1のデータ数を数え，その2分の1をサイズとし，ポインタとして関数形式の関数パラメータの先頭をさすアドレスを，データとしてレベル1のデータの各々を，前に述べた、通信用フォーマットで，下方向のプロセッサへ渡し，処理を依頼する。

関数形式 Construction もほぼ同じであるが、この場合には、関数形式の関数パラメータの数を数え、その半分をサイズとして，各関数パラメータの先頭アドレスをポインタとし，データはそのまま、通信用フォーマットで、下方向へ送る。

ここで受け取り側では，前節で述べた処理を行なう。ただし，送られてきたサイズが1以上であれば，そのサイズを半分にして、下方向へ送り、以後送られてくるユニットを，Left，right の順にすべて送る。そして後で述べる Insert の実行の場合を除いて結果を下から受けとり、そのまま上へ送る。

この様にすると、受け取ったサイズが1以外のプロセッサでは、処理は行なわないことにな

る。しかしながら、図Bの様な並列処理で、各プロセッサで実行される関数がさらに並列処理が可能である場合に、この処理を、他の遊んでいるプロセッサへ依頼することを考えると、この時、プロセッサが十分にあるとすると、通信効率の点から考え、この様な方式が有効と思われる。

関数形式 Condition は図14の様に並列処理できる。

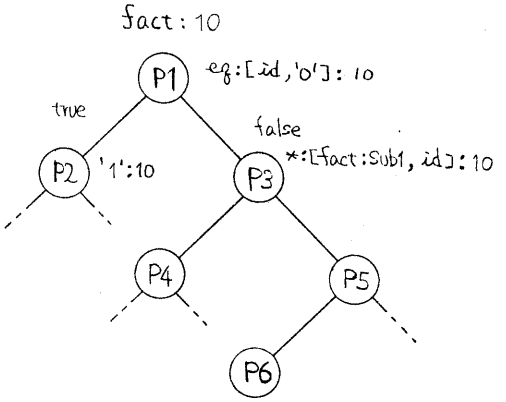

図14 Condition の処理例

これは階乗をもめるプログラムである。ここでP1 は条件の部分を実行し、そのLeftのP2は、条件がtrue の場合を実行し、rightのP3は、条件がfalse の場合を実行する。この3つが同時に実行される。そして条件に合う方より結果を受けとり、他方は、割り込みにより処理を中止する。

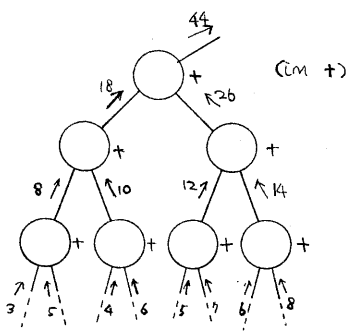

図15 Insert の処理例

関数形式 Insert は apply to all や Construction が、データやプログラムを分割しながら並列処理を行なうのに対し、2進木の構造を利用して、下方向からの結果を受けとり、それに対し演算を行なう。この時前に述べた、処理を中継するのみに働いていたプロセッサがすべて使われる。

しかしながら、この方法でInsertの処理を行なうと、関数形式の定義通りの順では実行できなくなる。

5.5.2 原始関数の処理

FP言語の原始関数の多くは、データの並び替えや、選択を行なう。一般にこれらの処理は、ポインタを用いて行なうと便利である。しかしながら、並列処理を考えると、ポインタを用いた方法では、その参照が複雑となる。そこで、CORAL-FPにおいては、ポインタ等を用いず、コピー等の発生によるオーバーヘッドがあるものの、直接データの変換を行なう。またすべての原始関数は、非並列で処理される。

5.5.3 問題点と改良

上に述べた様に、並列処理を行なうインタプリタを作成し、実際に処理を行なってみると、問題点があることに気がついた。

(1) 1つの関数形式を並列処理するためには必ず1度下方向へ通信し、その結果を受け取る必要がある。しかし、プログラム上では、しばしば並列処理を行なう関数形式が連続していて、この様な方法では、通信にむだがある。

(2) 原始関数は、すべて非並列的に処理されるが、原始関数の中には、並列処理用に、コピーを作成するものもあり、これは次に来る関数形式と組み合せて並列処理ができるのではないか。またこの原始関数は非常に多くのメモリを使う。

(3) すべての関数に対して同じ表現で並列処理を示せるため、必要以上に小さな処理まで、並列処理をし、リーフプロセッサまですぐに処理が行ってしまい、有効にプロセッサが使えない。

そこで、次の様な改良を行なって、前述の(1),(2)の問題を解決することができた。まず(1)については、1度配られたプログラムは、

その関数形式内の実行が終了した後に、その関数形式に続く関数形式が並列処理が可能か調べ、可能であればそのまま続けて実行する様にした。これにより不要な通信をはぶくことができる。

(2)については、並列処理が次に行なわれることの多い関数 distl, distr を改良した。この関数の次に並列処理の関数形式がある場合は、コピーを作らず、直接、次の関数形式で行なわれる通信に合せた情報を下方向へ送る。これにより、メモリの使用量をかなり減少することができた。

また関数形式 Insertは、ほとんどの場合、他の並列処理の次に出現するので、Insertの実行のために、プログラムを分配することはやめ、他の関数形式の並列処理の結果に対して作用する場合にのみ並列処理とした。

(3)については、特にプロトタイプの様にプロセッサ数の小さい場合には問題である。今のところ解決策を発見するに到っていない。

6 並列処理効率の評価

以上の様にして、CORALプロトタイプ上に作成した、CORAL-FP上で、実際に、FPプログラムを実行してみた。

実行したプログラムは行列の乗算である。

```
def ip = (in +):(aa *):trans endef

def mm = (aa (aa ip):distl):distr:
                    ↑2      ↑1
                [ 1,trans:2] endef
```

行列の乗算のプログラム

サイズ＼プロセッサ数	3	7	15
16×16	21,2	12,3	8,6
18×18	29,8	18,1	12,3
20×20	40,5	23,0	15,8

表1 実行時間 (単位 秒)

実行結果は、表1に示す通りである。ここで各サイズの時の、プロセッサ数に対する速度の向上率は、3台の場合を1とすると表2の様になる。

サイズ＼プロセッサ数	3	7	15
16×16	1	1,76	2,47
18×18	1	1,65	2,42
20×20	1	1,76	2,56

表2 速度向上率

ここで、この行列の乗算のプログラムを展開して考えてみると、20×20の行列の場合、2乗目の並列処理(プログラムの↑のところ)で20個の処理が発生する。この処理は、各プロセッサ数に対し図16のように配られ、すべてリーフで処理される。

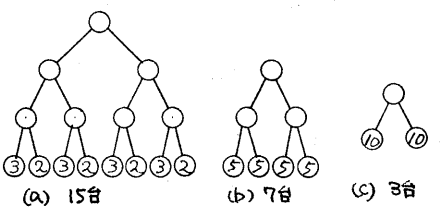

図16 処理の分配1

ここで、それぞれの並列処理は、同じ計算量を持っているので、リーフでの処理時間は、受け取った処理の数により決定する。従って、通信等のオーバーヘッドがなければ、15台で3、7台で5、3台で10の時間がかかる。同様に、16×16, 18×18の行列の場合を考え、3台の場合の処理を1として、向上率は表3のように予想される。

サイズ＼プロセッサ数	3	7	15	32	64
16×16	1	2	4	8	16*
18×18	1	1.8	3	4,5	9
20×20	1	2	3.3	5	10

表3
(*は3乗目の並列処理を考えている)

これは、通信等によるオーバーヘッドを考えると、ほぼ表2に等しい。ここでサイズの小さいものが、効率に開きがあるのは、処理量に対し通信等のオーバーヘッドが大きいためと思われる。

ここで、CORALプロトタイプの様な、プロセッサ数の少ない場合には、プログラムがリーフのみに集中してしまうため、3番目（プログラムの位のところ）に発生する並列処理が生かせないことがわかる。つぎに、2番目に発生する並列処理を、すべてのプロセッサに同数づつ分配した場合を、前に述べた方法で考えてみると図17、表4の様になる。

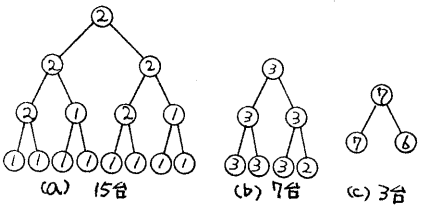

図17　処理の分配2

サイズ ＼ プロセッサ数	3	7	15
16x16	1	2	3
18x18	1	2	3
20x20	1	2,3	3,5

表4

台数が少ない場合この方が、プロセッサが多数使われるために、当然速くなることがわかる。

次に台数が少ない場合で、OSの存在する場合を考える。ここで注目するのは、すべてのプロセッサへプログラムを分配した場合である。図17の(a)に注目する。ここで2番目の並列処理により、処理数が、2と1のプロセッサができる。全体の処理速度は、その大きい方となる。ここでOSを使うことにより、処理量1のプロセッサが処理を終る後に、処理量2のプロセッサで発生する3番目の並列処理を受け持つことができると思われる。

ただし、あまり処理の小さな関数まで並列処理すると、OSのオーバーヘッド等のために、効率が上がらないと考えられる。

7　まとめ

関数型言語を、2重木構造並列システムCORAL上で並列処理を行なうシステムを作成しその上で、実際にプログラムを実行し、その効率を評価した。実行したプログラムは小さな問題ではあったが、関数型プログラムの並列処理を効率よく行なうための、種々の問題点の検討に役だった。今回は、どの程度の効率が得られるかを中心に見てきたが、今後は　すべてのプロセッサを有効利用するための方策、専用OS、並列処理に適した言語仕様の拡張　などについて研究を続けて行きたいと思う。

最後に、本研究に御協力くださった、桑原昭氏、日立製作所信友義弘氏に感謝します。

参考文献

1　John Backus "Can Programming Be Liberate from von Neumann Style? A Functional Style and Its Algebra of Program" CACM vol.21 No.8 1978 PP 613～641

2　Gyula A.Mago "A CELLULAR COMPUTER ARCHITECTURE FOR FUNCTIONAL PROGRAMMING" IEEE Computer Society COMPCON (Spring 1980) PP 179～187

3　Gyula A.Mago "A Network of Microprocessor to Execute Reduction Languages Part I, II" International Journal of Computer and Information 1979 vol.8 No.5 PP 349～385 No.6 PP 435～471

4　Robert M. Keller, Gary Lindstrom, Suhaspatil "A Loosely-coupled applicative multiprocessing system" national computer conference 1979 PP 613～622

5　Yoshizo Takahashi, Naoki Wakabayashi and Yoshihiro Nobutomo "A Binary Tree Multiprocessor : CORAL" Journal of Information Processing Vol.13 No.4 PP 220～237

6　Gyula A.Mago, Roy P.Pargas "SOLVING PARTIAL DIFFERENTIAL EQUATION ON A CELLULAR TREE MACHINE" 10th IMACS World Congress System Simulation and Scientific Computation 1982 August

7　伊藤，斉藤，星子 "関数型データフロー計算機システム" 第35回計算機アーキテクチャ研究会 1981.10

8　西山，高橋 "2重木構造マルチプロセッサCORAL上での関数型言語中の並列処理" 情報処理学会第25回全国大会