

解説



関数型プログラムの画像処理への適用†

太田 誠††

1. はじめに

画像処理は科学技術計算や数値シミュレーションなどとともに並列処理の代表的適用分野である。実際、画像処理には空間的な並列性と時系列上の並列性が内在しており、この並列性をハードウェア的に抽出する画像処理専用マシンが多数開発されている¹⁾。画像処理の記述言語としては FORTRAN が圧倒的な実績をもっており、現在なおこの分野の中心的なプログラミング言語となっている。しかし、並列ハードウェアによる高速化が期待されるこの分野に、逐次処理をベースとする言語を用いることは本来望ましくない。一方、関数型言語は、入出力データの依存関係に従って処理が規定されるため並列性の抽出が容易である、構造が簡明でありプログラムの階層的構築がしやすい、などのすぐれた特徴をもっている。本稿は、画像処理と関数型プログラミングの適合性について論じることを目的としている。まず画像処理の特徴を特に並列性という観点から述べ、次に並列ハードウェアに関数型言語を適用した場合にどのような利点と問題点があるかについて説明する。そして最後に具体例として、画像処理プロセッサ用に著者らが開発している関数型言語 Stream^{2),3)}での並列性抽出と問題点について述べる。

2. 画像処理のもつ並列性と処理方式

一般的に、画像処理は比較的簡単な処理を大量の画像データに施すという特徴をもっている。したがって、1画素当りの処理量は小さくてもその繰り返し回数が多いために全処理量は大きくなる。そこで画像処理に内在する並列性を考えてみる。たとえば、ある画像 $A[i, j]$ に 3×3 の空間フィルタ演算を施して $B[i, j]$ を得るという処理は、着目している画素と近傍の

8画素を用いて、

$$B[i, j] = \sum_{k=-1}^1 \sum_{l=-1}^1 W[k, l] A[i+k, j+l]$$

$W[k, l]$ = 重みを与える 3×3 マトリックス

と表される。これは、 (i, j) あるいは (k, l) に関する空間的な並列処理の可能性を示している。また、たとえば原画像 $A[i, j]$ に対して

F G H

$$A[i, j] \rightarrow B[i, j] \rightarrow C[i, j] \rightarrow D[i, j]$$

のように複数の基本処理 F, G, H を施し、中間画像 $B[i, j]$, $C[i, j]$ を経て目的画像 $D[i, j]$ を得るというようなことがしばしばある。いま、中間画像 $B[i, j]$ から次の中間画像 $C[i, j]$ を得るのに、画素単位の処理を繰り返し施すとする、処理の終了した画素は $C[i, j]$ が完全に得られるのを待たずに、さらにその先のフェイズの画像 $D[i, j]$ を得る処理を始められる場合がある。また、独立な複数の画像 $A[i, j]$ に F, G, H を施す場合は、この処理系列をパイプライン的に処理することができる。これらは、時系列上の複数の処理フェイズを並列化できる可能性を示しているといえる。

画像処理には、平滑化やラベリングといった基本処理の組合せによってかなり高度な処理まで行えるという特徴があり、それらの基本処理は十分な並列性をもっている。表-1には画像処理の8つの典型的処理パターンとそれに属する基本処理、そしてその並列性についてまとめている。n を画像サイズすなわち正方形画像の一辺の画素数とすると、ほとんどの処理パターンが、 $O(n^2)$ の並列度をもっていることが分かる。

次に、それらの並列性を抽出する処理方式について見てみると次の4つがある。

(1) 完全並列方式

プロセッサを画素と同じように2次元的に配列し、各プロセッサには対応する一つの画素の処理を受け持たせる方式。

(2) 局所並列方式

局所的なウィンドウ内の画素のアクセスや演算を並

† Functional Programming and its Application to Image Processing by Makoto OHTA (C&C Common Software Development Laboratory, NEC Corporation).

†† 日本電気(株)C&C 共通ソフトウェア開発本部

表-1 画像処理の典型的処理パターンにおける並列性
 N : 画像サイズ M : マスクサイズ P : 内挿に用いる点のサイズ

処理パターン	処 理 例	計 算 量	並 列 度	並 列 性
空間フィルタ	鮮明化, 平滑化, エッジ強調	$O(N^2)$	$O(N^2)$	画素並列
		$O(M^2)$	$O(M^2)$	局所並列
論理フィルタ (二値画像)	細線化, 平滑化, エッジ検出	$O(N^2)$	$O(N^2)$	画素並列
幾何学的変換	拡大, 縮小, 回転	$O(N^2)$	$O(N^2)$	画素並列
		$O(P^2)$	$O(P^2)$	局所並列
画素データの変換	階調の非線形変換, 二値化	$O(N^2)$	$O(N^2)$	画素並列
画素間演算	論理和, 論理積	$O(N^2)$	$O(N^2)$	画素並列
ラベリング	領域分割	$O(N^2)$	$O(N)$	wavefront
FFT (二次元)	鮮明化, 平滑化	$O(N^2 \log N)$	$O(N^2)$	画素並列
ヒストグラム	面積や長さなどの特徴量計算	基本的に逐次的		

列に行う方式 (図-1).

(3) パイプライン方式

連続的な画像処理系列をパイプライン的に実行する方式.

(4) マルチプロセッサ方式

MIMD 型のマルチプロセッサを用いて負荷分散をする方式.

(1)~(3)はアルゴリズムの規則性を利用して基本的処理パターンをハードウェア化しようとするもので, 高いスループットが得られる反面, 処理が半固定的になるデメリットをもっている. 一方, (4)は種々の複雑な処理に柔軟に対処できるが, 特定のプロセッサがネックにならないように負荷分散を最適化する必要がある. これらの処理方式は本来排他的なものではなく, 実際には複数の方式を巧みに組み合わせて処理が行われている. 詳しくは文献 1) や文献 4) などを参照されたい.

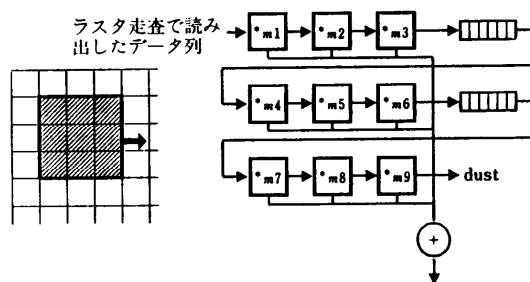
3. 関数型プログラミングの適用

画像処理は並列ハードウェアによって高速化できるが, そのためには従来の逐次型言語ではなく, 自然に並列性を抽出できかつ人間に理解しやすい言語でプログラムを書くことが望ましい. そのような言語として関数型言語が考えられる.

3.1 関数型言語の特徴

本稿では並列性の抽出に関数型言語を利用するという立場から, 主として関数型言語の次の性質に着目する.

(1) 高い参照の透明性を有する



(a) 3×3 空間フィルタの window (b) 局所並列方式による空間フィルタ処理 (mi: マスクデータ)

図-1 局所並列方式

(2) 副作用がない

(3) プログラム変換がしやすい

また, 関数型言語の計算機構は

(a) λ 計算を忠実に行う方式 (リダクションマシンなど)

(b) 環境評価方式 (通常の計算機上の Lisp など)

(c) データ駆動方式 (データフローマシン)

の3種類に分類される⁵⁾. 以下では関数型言語で書かれたプログラムを並列画像処理装置, 特に関数の一つの計算機構であるデータ駆動型の並列処理装置上で実行することを念頭に置いて説明する.

3.2 並列性抽出の一般論

関数構造にはさまざまな並列処理の可能性が内在していることはよく知られている. たとえば, $f(x, y, z)$ という関数記述に対して

(a) 引数と関数 f 本体の並列評価

(b) 三つの引数 x, y, z の並列評価

(c) 関数 f と他の関数の並列評価

(d) x, y, z の異なる組の間の並列評価

の4種類の並列処理が可能である(図-2)。(a)はパイプライン的な処理であり、(b)(c)は並行処理である。また、(d)はストリーム型並列処理と呼ばれるものであり、通常の繰り返し処理に相当している。画像処理においては各画素に対して同じ処理を施すことが多く、(d)の並列性はよく現れるものである。もちろん、関数 f が純粋な関数でなく副作用がある場合は、その変数に関する同期機構の導入が必要となる。

3.3 一様な処理での並列性

3.3.1 空間への展開と時系列への展開

全画素に対して同じ操作を施す処理では、一つの画素に対する処理内容を関数として表現するのがよい。たとえば、ある画素の処理にその上と左の画素の情報が必要な場合は

$$f(a[i, j], x[i, j-1], x[i-1, j]) \text{ (for all } i, j)$$

と書ける。ここに、 $a[i, j]$ は原画像、 $x[i, j]$ は隣接点からの情報(たとえば $a[i, 0] \sim a[i, j]$ の最大値)を表す。すべての画素に f という処理を行うわけであるが、 $a[i, j]$ 1画素を一つのプロセッサに固定し、 f を空間的に展開して実行すれば完全並列型処理方式(図-3)となり、たとえば1列ごとの処理を一つのプロセッサに固定して、 f を繰り返し施せばパイプライン型処理方式(図-4)となる。

3.3.2 computational wavefront

ここでは、3.3.1の例よりも少し複雑な場合を考える。多次元配列の要素間に局所的な依存関係があると、同時に全要素を処理できず、同時に処理できるのはより次元の小さな領域である。この領域は時間とともにさながら波頭のように伝搬するので computational wavefront と呼ばれている。以下に、画像配列上

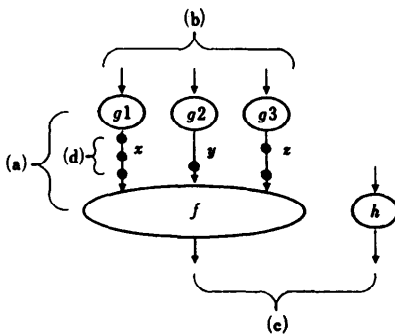


図-2 関数記述 $f(x, y, z)$ に内在する並列性
(黒丸はデータを表す)

の局所的な処理内容を表す関数記述から、computational wavefront が容易に抽出されることをラベリング処理の例で示す。ラベリングでは、図-5に示すように着目点の左上、真上、右上、左の画素のラベルが既知であれば、その点のラベルを決定できる。いま、原画像を $a[i, j]$ 、ラベルづけされた画像を $b[i, j]$ 、着

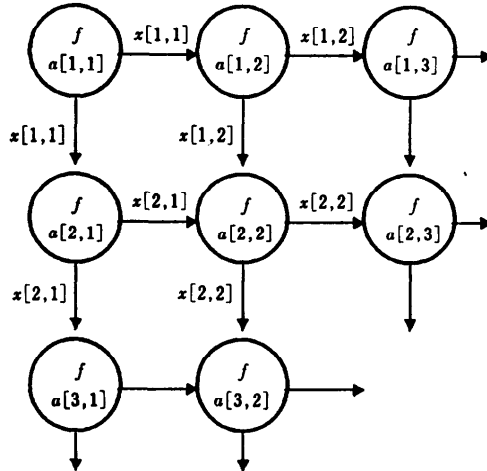


図-3 処理単位 f の空間への展開

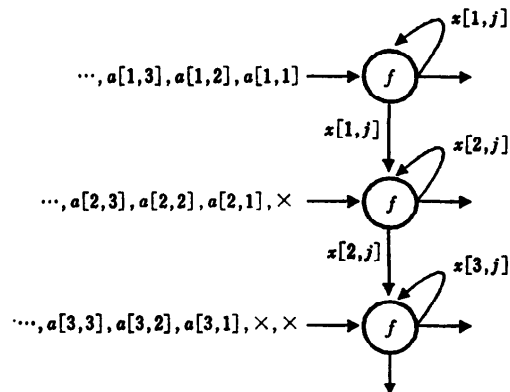


図-4 処理単位 f の時系列上への展開
(×はデータがないことを表す)

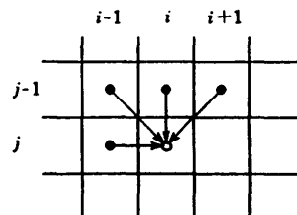


図-5 ラベリング処理のデータ依存関係

目点のラベルを求める関数を f とすると、

$$b[i, j] = f(a[i, j], b[i-1, j-1], b[i, j-1], b[i+1, j-1], b[i-1, j]) \quad (1)$$

と表される。引数に対して再度 f の表現を適用すると

$$\begin{aligned} b[i, j] = & f(a[i, j], \\ & f(a[i-1, j-1], b[i-2, j-2], b[i-1, j-2], b[i, j-2], b[i-2, j-1]), \\ & f(a[i, j-1], b[i-1, j-2], b[i, j-2], b[i+1, j-2], b[i-1, j-1]), \\ & f(a[i+1, j-1], b[i, j-2], b[i+1, j-2], b[i+2, j-2], b[i, j-1]), \\ & f(a[i-1, j], b[i-2, j-1], b[i-1, j-1], b[i, j-1], b[i-2, j])) \end{aligned} \quad (2)$$

となる。ここで、 $b[i, j]$ が求まる時刻を $T(i, j)$ と表すことにすると、(1)式と(2)式より

$$\begin{aligned} T(i, j-1) &> T(i-1, j-1) \\ T(i+1, j-1) &> T(i, j-1) \\ T(i-1, j) &> T(i-1, j-1) \\ T(i-1, j) &> T(i, j-1) \end{aligned}$$

ゆえに

$$T(i+1, j-1), T(i-1, j) > T(i, j-1) > T(i-1, j-1)$$

となる。この関係を、単位処理時間を D として図示したのが図-6 である。図-6 から分かるように $b[i+1, j-1]$ と $b[i-1, j]$ は並行して評価することが可能である。ラベリング処理は一樣な処理なので、 $b[i+1, j-1]$, $b[i-1, j]$ の相対的な位置関係と同じ位置関係にある画素のラベルはすべて並行して計算できる (図-7 の破線)。これが computational wavefront である。以上のように関数表現 (1) は wavefront 上の並列性を陰に含んでいる。

3.3.3 Wavefront array

systolic array や wavefront array は、画像処理を適用分野の一つとしている。systolic array は多次元

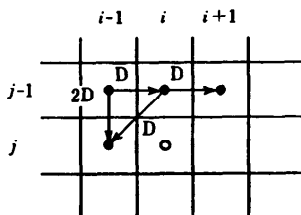


図-6 ラベル付けのタイミング
(D: 単位処理時間)

に拡張されたパイプラインと見なすことができ、各プロセッサでの演算タイミングを考慮してデータ列を入力しなければならない。すなわち、systolic array におけるプログラミングは、各プロセッサの処理記述とスケジューリングの両方を含んでいる。これに対して、wavefront array は要素プロセッサとしてデータ駆動型プロセッサを用いたもので、スケジューリングは不要である。プログラマは各プロセッサの局所的な処理内容を記述しさえすれば、ハードウェアが前述の computational wavefront を抽出しつつ実行する。

一樣処理のアルゴリズムを systolic array や wavefront array にマッピングする手続きは、一般的に次のとおりである⁶⁾。

- (1) Dependency Graph を作る。
- (2) Dependency Graph を一方向に射影して Signal Flow Graph を作る。
- (3) Signal Flow Graph の各ノードの処理を対応するプロセッサへマッピングする。

前記ラベリング処理の例で考えてみる。まず初めに、Dependency Graph は図-8 のようになる。プロセッサ数は n^2 (i : 画像サイズ) のオーダーなのに対して並列性は n のオーダーなので、このままでは効率が悪い。そこで図-8 を一方向に射影することを考える。射影

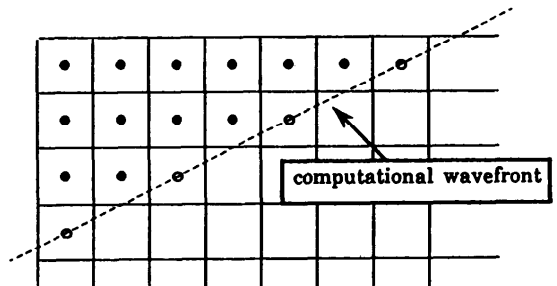


図-7 ラベリング処理における computational wavefront

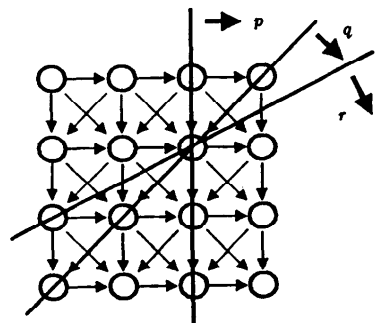


図-8 ラベリング処理の Dependency Graph

方向としては p, q, r などが考えられるが、ここでは p の方向に射影することにする。射影の結果、図-9 のような Signal Flow Graph が得られる。そのノードの処理内容は図-10 のようになり、隣接点との通信を入出力とする関数と見なせる。基本的に wavefront array はデータフロー計算機であり、そのプログラミング言語としては関数的性質をもつのがよい。このような言語としては、VAL⁷⁾、SISAL⁸⁾、MDFL⁹⁾ などがある。

3.4 非一様な処理の並列性

前節で、一様な処理の並列性と関数表現の関係について述べたが、複雑な画像処理を行おうとすると非一様な処理あるいは非定型処理が必要になる。このような場合には、並列性抽出の一般論 (3.2) で述べた4種類の並列性を利用する。詳しくは 4. の適用例でふれる。

3.5 画像配列

関数型言語の枠組みの中で配列をどう取り扱うかということは、関数型プログラミングを画像処理に適用する上で重要な問題である。配列の操作に関して処理

の正当性を保証するには

- (1) 配列から副作用を排除する
- (2) 配列の読み書きに同期機構を導入する

という二つの方向が考えられる。(1)の方法は配列に単一代入規則を適用し¹⁰⁾、配列要素への再代入をその要素のみが異なる新しい配列の生成と考えるものである。概念的には関数機構と適合性が良いが、特に画像配列のような大きな配列に対しては効率が悪い。また、複雑な画像処理を行おうとすると履歴依存処理は不可欠であり、副作用の利用が必要となってくる。(2)の方法は配列の副作用を認めてしまい、その代わり読み書きのタイミングを明確に表現できるようにするものである。この方法は効率は良いが、とするとプログラミングが逐次的になり、関数型プログラミングの特徴を生かせなくなってしまう。そこで一般的に最小限の副作用を許し、そのぶん同期機構の導入でカバーするという方法が取られる。たとえば、副作用のない変数と履歴変数や履歴配列を区別する、などの試みがある¹¹⁾。

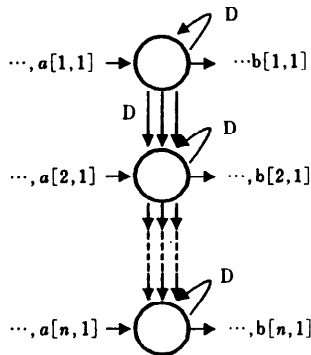


図-9 ラベリング処理の Signal Flow Graph (D: 単位処理時間)

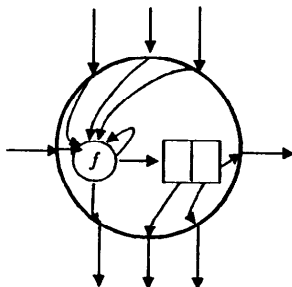


図-10 ラベリング処理における wavefront array の要素プロセッサの処理

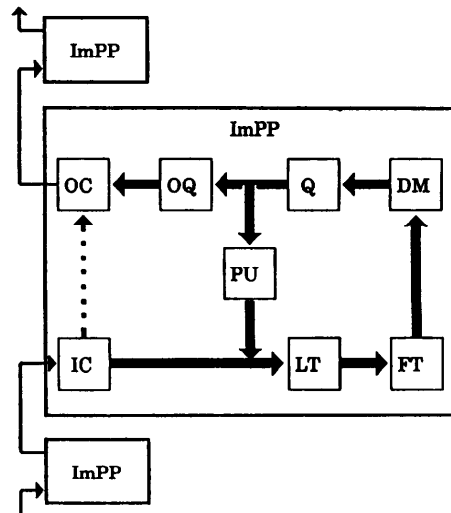


図-11 ImPP ブロック図

リング1周で1命令を実行

IC : Input Controller

OC : Output Controller

LT : Link Table

FT : Function Table

DM : Data Memory

Q : Queue

PU : Processing Unit

OQ : Output Queue

4. 適用例

前章では一様な画像処理を中心として、並列ハードウェアと関数記述の適合性について述べた。本章では非一様な場合も含めた一般の画像処理に関数型プログラミングを適用する具体例として、画像処理用データフロープロセッサ μ PD 7281 (Image Pipelined Processor: ImPP) のために筆者のところで開発している関数型言語 Stream の並列記述と問題点について説明する。

4.1 ImPP

ImPP はパイプライン処理を柔軟なものとするために、循環パイプラインとデータ駆動型の計算機構を採用している¹²⁾。すなわち、次の演算の処理情報をデータ自身が保持しながら（このデータと処理情報をまと

めてトークンと呼ぶ）循環パイプライン上を周回し、その間に所定の処理を受ける。図-11 には ImPP の内部ブロック図が、また図-12 には処理例とそのタイミングが示してある。ImPP は演算ユニット (PU) を一つしかもたないが、ImPP 自体をリング状に多段に結合して用いることにより MIMD 型の処理が行える。そうした意味で、ImPP システムはパイプライン方式とマルチプロセッサ方式の両面をもっている。

4.2 Stream 言語での並列記述

本節では Stream 言語の記述例を使って、その並列性の抽出について述べる。まず単独データの二値化関数 f を示すと

```
f(a, t: int) =
  if a > t then 1 else 0 end
end
```

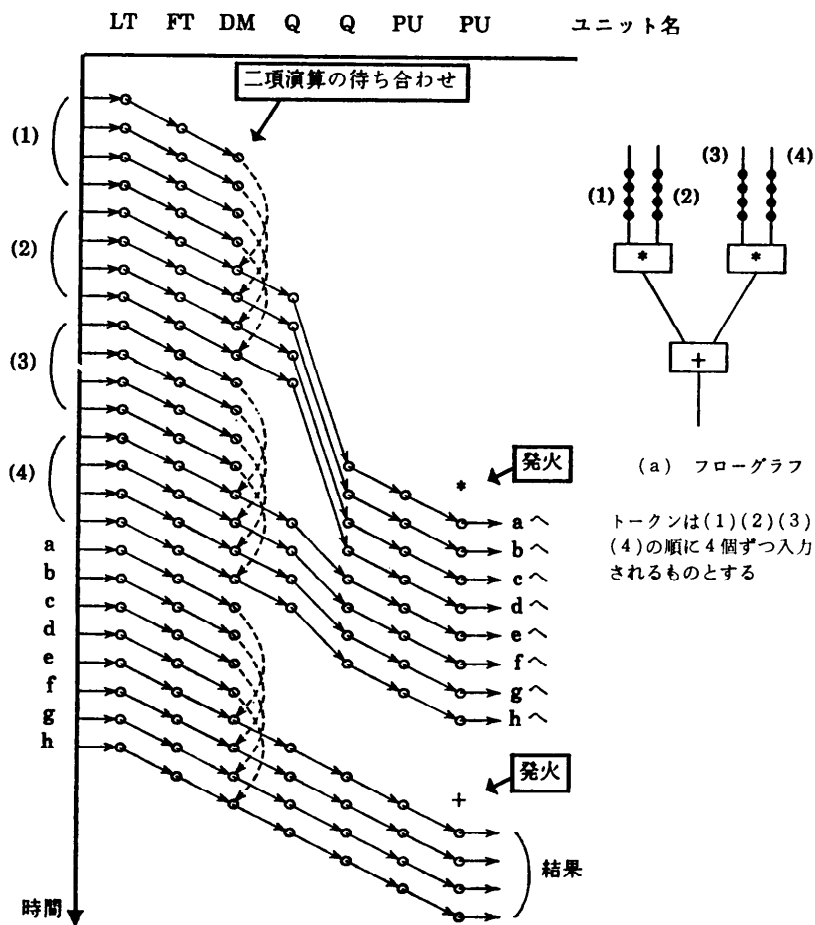


図-12 ImPP の処理例とタイミング

となる。ここで a は二値化するべきデータ、 i はしきい値であり、if は条件式である。次に画像 $A[i, j]$ のしきい値 i より大きい部分を残し、しきい値 i 以下の部分を 0 にする関数 g を示すと

```

g(i: int)=
  let X: int={ {p; M; 1} : p={0; N; 0} }
    Y: int={ {q; M; 0} : q={0; N; 1} }
  in
    for x, y: int in X, Y do
      A[x, y]*f(A[x, y], i)
    end
  end
end

```

となる。let 式はローカル名の定義とその参照を行うための制御構造で、let と in の間で定義されたローカル名 X, Y は in と end の間で参照される。 X と Y はそれぞれストリームデータと呼ばれる一次元的に並んだデータ列であり、上の例では配列 $A[i, j]$ のアクセスがラスタ走査となるように定義されている³⁾。for 式はストリームデータの先頭から要素を順次取り出して内側のブロックの評価を行う制御構造である。上の例では X, Y から要素 x, y を順次取り出し $A[x, y] * f(A[x, y], i)$ の評価を行っている。ここで、 f 及び g の記述から抽出される並列性をまとめると次のようになる。

- (1) 関数 f は副作用がないので画像配列 $A[x, y]$ へのアクセスと関数 f の適用は独立に行える。
- (2) for 式はストリームデータ X, Y についてストリーム並列に処理できる。
- (3) ストリームデータ X, Y の生成とその消費は並行して行うことができる。
- (4) ストリームデータ X と Y の生成は独立に行うことができる。これは let 式の約束である。

また、上の例でも分かるように、Stream 言語はストリームデータの生成と消費を基本的なプログラミングスタイルにしている。これは従来の逐次型言語ではループ変数に対応しているが、ストリームデータを用いることにより、自然にストリーム並列の可能性が生まれてくる。もちろんこのためにはストリームデータを消費する部分に副作用がないことが重要である。さらに、ImPP には複数のトークンを連続的に生成する命令があるが、ストリームデータの生成規則はこの命令に直接的に翻訳することができる。以上述べたように Stream 言語は ImPP の命令セットと親和性がある。

りかつ並列性の抽出が容易であるが、それに対して関数構造は重要な役割を演じている。

4.3 ノンストリクトな関数

ある関数 f があり、その引数 x の値が確定しなければ f の値も確定しないとき、 f は x に関してストリクトであるという¹³⁾。また、すべての引数に関してストリクトである関数をストリクト関数といい、そうでない関数をノンストリクト関数という。ノンストリクトな関数としては条件関数 (if) や論理和、論理積などが考えられる。

ImPP のようにデータ駆動によって関数型プログラムを並列実行しようとするとき、関数のストリクト性が問題となる。たとえば、

```

f(x, y: int)=
  let z: int=g(x, y)
  in
    h(x, z)
  end
end
g(p, q: int)=p+q end
h(r, s: int)=if r<0 then 0 else s end end

```

というプログラムにおいて、各関数のストリクト性は次のようになる。

```

f: {x, y} に関してストリクト
g: {p, q} に関してストリクト
h: {r} に関してストリクト
{s} に関してノンストリクト

```

3.2 で述べた関数表現に内在する並列性を適用すれば、 x, y にストリームデータが入力する場合、関数 g と関数 h は並列に評価できることになる。しかし、関数 h は引数 s に関してノンストリクトであるため、関数 g の評価結果が参照されない場合がある。データ駆動によってこのようなノンストリクトな関数を評価する場合には、参照されないデータはゴミデータとして残り、誤った結果を与えることがある。これに対処するには次の二つの方法が考えられる。

- (1) プログラム変換を用いる方法

前掲の例は次のように変換される。

(原形)

↓ …… ノンストリクトな関数の展開

```

f(x, y: int)=
  let z: int=g(x, y)
  in
    if x<0 then 0 else z end

```

```

end
end
      ↓……let の簡約化
f(x, y: int) =
  if x < 0 then 0 else g(x, y) end
end

```

このプログラム変換により、関数 g の評価結果は必ず参照されるようになった。一般に関数型言語のプログラム変換は、その参照の透明性をよりどころとしている。いまの例でいうと、関数 g はどこに置かれていても同じ結果を与えるということが前提となっている。

(2) バッファとダスト機構を用いる方法

前掲の例をそのまま並列評価するには図-13 のようにバッファを用いればよい。関数 g の評価結果はノンストリクトな関数 h の入口にあるバッファに格納される。関数 h では関数 g の評価結果の要不要が確定した時点でバッファからデータを取り出し、もし不要であれば消滅させる。このようにすれば関数 h の評価中も関数 g の評価を行うことができ、ストリーム並列を実現できる。

(1)の方法は関数 h の引数 s が真に必要なまで s の評価を行わないように変形したもので、プログラムの一部を要求駆動化したことになっている。しかし、ノンストリクトな関数が条件関数のときはこのように変換できるが、すべてのノンストリクト関数に適用できる方法ではない。一方、(2)の方法は汎用性はあるが必ず関数 g の評価を行うため、条件分岐の確率

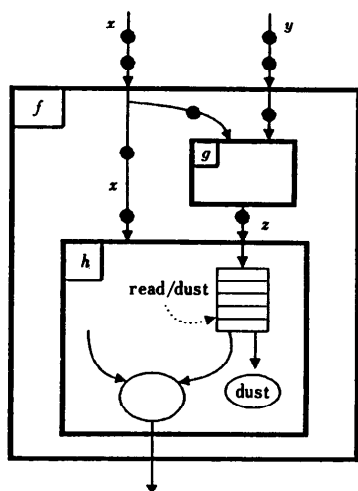


図-13 バッファを用いたノンストリクトな関数の実行
(黒丸はデータまたはトークンを表す)

や g の処理量、バッファリングとダスト機構の相対的なオーバーヘッド量によっては、かえって性能が低下してしまうことがある。たとえば、else 側の分岐確率が小さい場合には(2)の方法は不利である。実際上は、(2)の方法を主として使い、最適化の一つとして(1)を併用するのが良いであろう。

4.4 ボトム

ImPP は循環パイプラインを採用しているが、不要なトークンがこの中を周回していると性能低下をまねく。そこで ImPP ではこのようなトークンを消滅させる機能をもっている。Stream 言語はそれに対応するものとしてボトム ($\perp$) を導入している。すなわち、ボトムは値の消滅あるいは空を表している。たとえば、

```

let a,  $\perp$  : int = f(x)
in
  g(a)
end

```

において多値関数 f は二つの値を返すが、2 番目の値は捨てられる。また次のように等号の右辺にボトムが書かれた場合、その式自体の値がボトムとなる。たとえば

```
a : int = if x < 0 then  $\perp$  else 1 end
```

において、 a は x が負のときボトムに、それ以外のとき 1 になる。したがって容易に分かるように、ボトムは式の評価にともなって伝搬してゆくことになる。一般にデータ駆動によりこのボトムを正しく実現しようとするとコンパイル時にボトムの伝搬解析が必要である。そのことを次の簡単な例で説明する。

```
f: f() = if x < 0 then  $\perp$  else p(x) end
```

```
P1: let a : int = f()
in
```

```
  r(a)
```

```
end
```

```
P2: let a : int = f()
in
```

```
  if  $x * x < 1$  then q() else r(a) end
```

```
end
```

P_1 はそのままデータ駆動で実行してもかまわない。 a の値は $x < 0$ のときボトムとなり、関数 r のトリガがかからないため let 式自体も正しくボトムとなる。一方、 P_2 はそのままデータ駆動で実行することはできない。なぜなら、 $-1 < x < 1$ のとき関数 q は a の値のいかんにかかわらず評価可能であるのに、 a が

if 式のトリガになっているため、 a がボトムとなる区間 $-1 < x < 0$ で q のトリガがかからないからである。 P_2 を

P_3 : if $x * x < 1$ then $q()$ else $r(f())$ end

と変形すればこのような不都合はなくなる。そこで一般にはボトムの伝搬解析を行って、このような変換の必要性及び是非を判定しなければならない。

5. おわりに

画像処理は並列ハードウェアによって高速処理が期待されるが、関数型プログラミングは並列性抽出能力が高いことからその有力なプログラミングスタイルであると考えられる。しかし一方で、画像配列の効率的な取り扱いやデータの消滅に関連する問題など、具体的な並列ハードウェアに適用する際にはいろいろな問題が生じる。これらの問題は今後とも検討する必要があるが、それとともに並列ハードウェアの設計と言語の設計を十分密な関係のもとに行うことが、関数型プログラミングを効果的に画像処理に適用する上で重要であると考えられる。

参考文献

- 1) 前田：画像処理マシン，情報処理，Vol. 28, No. 1, pp. 19-26 (1987).
- 2) 太田他：ImPP 用高水準データフロー言語 Stream の設計，情報処理学会第 34 回全国大会論文集 1U-1, pp. 709-710 (1987).
- 3) 太田：ImPP 用高級言語 Stream，情報処理学会研究会資料，Vol. PL-14, No. 5, pp. 10 (1987).
- 4) 木戸出：画像処理における並列処理，コンピュータール，No. 19, pp. 66-73 (1987).
- 5) 横井：関数型言語，情報処理，Vol. 22, No. 1, pp. 583-587 (1981).
- 6) Kung, S. Y. et al.: Wavefront Array Processors-Concept to Implementation, Computer, pp. 18-33 (July 1987).
- 7) Ackerman, W. B. et al.: VAL-A value oriented algorithmic language: Preliminary reference manual. Tech. Rep. TR-218, Computation Structures Group, Laboratory for Computer Science, MIT., Cambridge, Mass. (Jun. 1979).
- 8) McGraw, J.: SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Lawrence Livermore National Laboratory (1985).
- 9) Kung, S. Y. et al.: Wavefront Array Processor: Language, Architecture, and Applications IEEE Trans. Comput., pp. 1054-1066 (Nov. 1982).
- 10) Ackerman, W. B.: Data Flow Languages, Computer, pp. 15-25 (Feb. 1982).
- 11) 尾崎他：データフロー方式の画像処理への応用に関する研究，信学技報，Vol. 87, No. 41, pp. 41-48 (1987).
- 12) Iwashita, et al.: Data Flow Chip ImPP and Its System for Image Processing, IEEE ICASP, 15-3, pp. 785-788 (1986).
- 13) Clack, C. et al.: Strictness analysis-a practical approach, Functional Programming and Computer Architecture, pp. 35-49, Springer-Verlag, Berlin (1985).

(昭和 63 年 4 月 25 日受付)