コンピュータビジョン　52—10
（1988. 1. 22）

リスプによる高速画像処理環境と処理システム［I］

Fast Image Processing and Processing System in a Lisp Environment [I]

城　和貴　　梅田　三千雄

Kazuki Joe　　Michio Umeda

ＡＴＲ視聴覚機構研究所

ATR Auditory and Visual Perception Research Laboratories

あらまし　一般に画像処理等の数値計算はリスプに向かないと言われているが、本稿ではリスプ環境下において Ｆｏｒｔｒａｎ と同程度のパフォーマンスを実現できる手法を提案する。この手法を用いて画像処理ルーチン・ＳＰＩＤＥＲの全ルーチンをＳｙｍｂｏｌｉｃｓ／３６７５・リスプ・マシンにコンバージョンを行い、ＶＡＸ８６５０／Ｆｏｒｔｒａｎ上のＳＰＩＤＥＲの約１９パーセントのパフォーマンスを達成した。ＶＡＸ８６５０のハード・ウェア的な計算能力がＳｙｍｂｏｌｉｃｓ／３６７５の約５倍であることを考えると、リスプ環境下での高速画像処理が実現されたことが示される。この結果、エキスパート・ビジョン、画像理解等の記号処理・信号処理の両方を必要とする研究をリスプ環境下で効率よく行うことが出来る。

Abstract

Lisp is usually considered to be inadequate for number crunching. This paper first presents a method that makes numerical computation for image processing in a Lisp environment as fast as in a Fortran environment. Then this method is applied to convert the SPIDER Image Processing Library (containing about 450 subroutines) to run on a Symbolics Lisp Machine. The result is that the Lisp SPIDER Library on a Symbolics/3675 works at about 19% of the speed of that of original Fortran Library on a VAX8650, while VAX is 5 times as fast as Symbolics from the view point of machine power. Thus the performance of the two programs are about equal.

１．はじめに

エキスパート・ビジョン、画像理解等の研究では、より高度なデータ構造、頻繁に行われる動的メモリ・アロケーション、再帰的処理等の記号処理が必要となる。これらはリスプ環境下では非常に整合性のあるものだが、その前の段階の信号処理では大量の数値計算を必要とするために、一般に数値計算に向かないと言われてきたリスプ下では、これまでに画像前処理はあまり行われていなかった。しかしながら、もし同一の環境下、すなわちリスプ環境下、において記号処理及び信号処理の両方を行うことが出来れば、全体として非常に効率の良いシステムが構築できるものと考えられる。[1]

そこで我々はリスプ・マシンのアーキテクチャを検討し、リスプ・マシンによる信号処理が実際の使用に耐えうる程に可能かどうかを調べてみた。その結果、システム・ルーチンの使用とプログラミング・レベルでのハンド・オプティマイズによりかなりのパフォーマンスが得られ、特に計算量の多いものを除けばインタラクティブな使用に耐えられることが判明した。

一方、画像前処理をはじめとする数値計算をリスプ下で行おうとする場合、利用できるライブラリは少ないので、リスプ環境下での画像処

理・理解のシステムを構築するには、先ずそのような基本的なルーチンのライブラリを構築しなければならない。現在、基本的な画像処理ライブラリとしては一般にＳＰＩＤＥＲが使われることが多いが、我々は前述したリスプによる数値計算のオプティマイズ法を用いてＳＰＩＤＥＲの全ルーチンをＳｙｍｂｏｌｉｃｓ・リスプ・マシンにコンバージョンを行い、ＳＣＬ－Ｓｐｉｄｅｒを作成した。

更に、一部のルーチンについてはＳｙｍｂｏｌｉｃｓに接続されるイメージ・コンピュータ、ＰＩＸＡＲを使うことにより、リスプ環境下においても高速の画像処理を可能とした。[1]

また、リスプ下での特徴としては前述したもの以外に、高度なインタラクティブ処理環境が上げられる。すなわち、これまでバッチ指向型のＦｏｒｔｒａｎの世界で行われてきた画像処理システムをインタラクティブな処理を前提とするリスプ・マシン環境下に置き換えることは、より高度なマン・マシン・インターフェイスを内包したシステムを構築できることを意味する。我々はこの思想をもとに、ＳＣＬ－Ｓｐｉｄｅｒの大部分のルーチンをＳｙｍｂｏｌｉｃｓ＋ＰＩＸＡＲで処理を行わせ、かつ、それらのほとんどの操作がマウスで行える画像処理メニュー・システム“Ｉｍａｇｅ　ＨｏｉＨｏｉ”を開発した。

本稿ではリスプ下で画像処理を行うためのリスプ・マシン・アーキテクチャ・レベルでの検討、ハンド・オプティマイズ法、及びＳＣＬ－Ｓｐｉｄｅｒの基本仕様、性能評価について報告する。

２．リスプ・マシン・アーキテクチャの検討

リスプ下での画像処理を検討するに当たって、パフォーマンスの問題は避けて通ることが出来ない。ただし、ここで言うパフォーマンスとは、純粋にどれくらい速いかということではなく、通常のＦｏｒｔｒａｎと較べて、同じ処理をどれくらいのマシン・サイクル・タイムで実行できるか、ということを意味している。

我々はこの考えをもとに、リスプ・マシンでの数値計算がどれくらいのパフォーマンスを出すことが出来るか、という検討を行った。理想的には、リスプのアーキテクチャにもとずいて設計されたリスプ・マシン上での全ての処理は、汎用機上のＦｏｒｔｒａｎと同等のパフォーマンスを出すはずであり、また、そのような目標を持つべきである。そして、リスプ・マシンとしては、画像処理で用いられる大量のデータに耐えうる仮想記憶空間を持つＳｙｍｂｏｌｉｃｓを用いることにした。

２．１．Ｓｙｍｂｏｌｉｃｓ・アーキテクチャ

Ｓｙｍｂｏｌｉｃｓ・リスプ・マシンは通常のレジスタ・マシンとは異なるスタック・オリエンテッド・マシンである。そのため、汎用レジスタ、アドレス修飾等の概念はない。アドレッシングはスタックが大部分を占め、コンスタントやスペシャル変数の場合のみインダイレクト・モードにより任意の仮想アドレスを指定できる[2]。ただし、このときのアクセス・タイムはスタックのアクセスに較べて２倍から１０倍かかる[3]。その理由は、スタックの一部が高速ＲＡＭ上に組み込まれていることにある。

Ｓｙｍｏｌｉｃｓの動作単位はプロセスであり、ユーザの使うプロセス以外にもシステムのプロセスが多数動いている。勿論、ユーザが複数のプロセスを使うことも出来る。各プロセスはスタック・グループと呼ばれるスタックの集合体に関係付られており、それぞれ独自のコントロール・スタック、バインディング・スタック、データ・スタックを持つ。コントロール・スタックは関数呼び出しの制御を行うものだが、リスプのアーキテクチャは関数呼び出しを基本としていることから、パフォーマンスに最も影響を与えるスタックが、このコントロール・スタックである[4]ことは想像に容易である。そして、このコントロール・スタックの先頭部分はスタック・バッファと呼ばれる高速ＲＡＭに自動的にマッピングされる。スタック・バッファは１Ｋワード（４Ｋバイト）からなる、他のマシンで言うキャッシュ・メモリのような役割をするものである。そしてこのキャッシュ・バッファ中のデータをアクセスするときには１マシン・サイクルタイムで実行される[4]。

Ｓｙｍｂｏｌｉｃｓのインストラクションは２種類に大別される。スタックのアクセスや分

岐命令、関数呼び出し等、基本的なものと、ファーム・ウェアによって組み込まれているマイクロ・コードである。基本的なインストラクションはその実行時間がマシン・サイクル・タイムと同じ２００ｎｓである[2]。そして、マイクロ・コードは当然のことながら、各コードごとに実行時間が異なる。たとえば、リスプの基本関数である car はマイクロ・コードとして組み込まれており、４サイクル・タイムで実行[4]される。また、足し算・かけ算はそれぞれ４、１６サイクル・タイムで実行される。

２．２．Ｆｏｒｔｒａｎマシンとの比較［Ｉ］

以上のようにＳｙｍｂｏｌｉｃｓがスタック・マシンであることを考慮に入れた上で、基本的な処理速度はＦｏｒｔｒａｎマシンとどれくらい違うであろうか。（表１）はＳｙｍｂｏｌｉｃｓ３６７５及びＶＡＸ８６５０における基本制御命令、基本演算子のベンチマーク・テストの結果である。除算や整数乗算のように差の大きいものもあるが、ＶＡＸ８６５０とＳｙｍｂｏｌｉｃｓとのＣＰＵの能力比が約５：１であることを考慮にいれると、少なくともこれら基本的な処理に関してリスプは決して遅くないと言える。

処理内容	Symbolic	ＶＡＸ	Symb/VAX
Add (int)	０．８０	０．３５	２．３
Add(float)	１．８２	０．２６	７．０
Sub (int)	０．７９	０．２７	２．９
Sub(float)	１．７８	０．４６	６．６
Mul (int)	３．４２	０．３８	９．０
Mul(float)	２．２６	０．４６	４．９
Div (int)	１２．５	１．０９	１１．５
Div(float)	１３．６	１．０７	１２．７
Loop	１．７５	０．６７	２．６
Condition	１．１８	０．３８	３．１

（表１）基本的な処理のベンチマーク・テスト
（単位：μ秒／１回の実行）
（注）本稿でのベンチマーク・テストはすべて以下の条件で行った。
ＶＡＸ８６５０／ＶＭＳ　Ｖ４．５
／ＶＡＸ－ＦｏｒｔｒａｎＶ４．６
メモリ３６ＭＢ
Ｓｙｍｂｏｌｉｃｓ３６７５
／Ｇｅｎｅｒａ　７．１
メモリ４ＭＷ（１６ＭＢ）
なお、ＶＡＸ８６５０はＶＡＸ７８０の６倍の６．５ＭＩＰＳと言われており、Ｓｙｍｂｏｌｉｃｓ３６７５は概算で１．３ＭＩＰＳと見積った。

また、（表２）はベンチマーク・テストそれぞれのソース・コード及び、インストラクションを示している。ＶＡＸには AOBLEQ のような高度なインストラクションもあるが、ＶＡＸのレジスタ・アクセスをＳｙｍｂｏｌｉｃｓのスタック・アクセスに置き換えると、ほぼ同等のコードを出力していることが分かる。２．１．で述べたように、スタック・アクセスの際にローカル変数はスタック・バッファにあるので、結局パフォーマンスの差はスタック・バッファ・アクセスとレジスタ・アクセスの違いに左右されることになる。

従って、ローカル変数がスタック・バッファに置かれるように注意すれば、３ステージのパイプ・ラインを持つＳｙｍｂｏｌｉｃｓ３６７５に関しては、ＶＡＸを始めとするＦｏｒｔｒａｎマシンとほぼ同等のパフォーマンスを出せると言えよう。

Ｓｙｍｂｏｌｉｃｓ		ＶＡＸ８６５０	
Source	Inst.	Source	Inst.
(setq i (+ j k))	push j + stack k pop i	I=J+K	ADD3L
(setq i (* j k))	push j * stack k pop i	I=J*K	MUL3L
(loop repeat n)	L1: push n +-p stack branch -false L2 1- n branch L1 L2:	DO I=1,N	L1: AOBLEQ N,I,L1

(cond
((= i
1))
(t))
	push i
	push 1
	- stack
	branch
	-false
		IF(I.EQ.
		1)THEN
		ELSE
		END IF
			CMPL 1,I
			BNEQ

（表２）ベンチマーク・テスト（表１）のソース・コードとインストラクション。
（注）Ｓｙｍｂｏｌｉｃｓの＊，－，1－，+-pはマイクロ・コード。また、push i はスタックにコール・フレーム中の変数 i の内容をプッシュすることを意味する。

２．３．Ｆｏｒｔｒａｎマシンとの比較［Ⅱ］

Ｆｏｒｔｒａｎを始めとするスタティック・データ中心の処理系では、データの型はコーディング時に決められる。一方、リスプにおいては、データの型自体もデータの一部であり、実行時になって初めてデータの型が決まるという場合すら有り得る。Ｓｙｍｂｏｌｉｃｓでは、データを処理する直前にそのデータの型をチェックする機能を持っているが、その機能がある

ためにかえって数値計算を遅くする場合もある。

一方、画像処理等の離散的データを扱う数値計算の場合、入出力データが整数値で、計算途中は小数値であることが多い。すなわち、integer と real との型変換が多く行われる。（表３）には、これらの型変換に要する時間が示されているが、これによるとＳｙｍｂｏｌｉｃｓはＶＡＸに較べて、Real -〉Integer で８６倍、Integer -〉Real で１７倍の実行時間を必要としている。これはパフォーマンスに重大な影響を与えるものと思われる。

Ｓｙｍｂｏｌｉｃｓ		ＶＡＸ８６５０	
Ｒｅａｌ → Ｉｎｔｅｇｅｒ			
truncate	４９．３	Ｉ＝Ａ	０．５０
round	４９．４	Ｉ＝Ａ+0.5	０．７５
ceiling	９０．９	--	--
zl:fix	４３．２	--	--
Ｉｎｔｅｇｅｒ → Ｒｅａｌ			
float	８．９４	float	０．５２
coerce	９．０９	--	--

（表３）型変換に要する時間
（単位：μ秒／１回の実行）

２．４．Ｆｏｒｔｒａｎマシンとの比較［Ⅲ］

２．２．、２．３．では基本的な処理のパフォーマンスについて検討したが、我々の目的がリスプ環境下における画像処理であるため、画像処理特有の要因についても検討しなければならない。

一般に画像処理においては、先に述べた基本的な処理以外に２次元配列のアクセスの問題がある[5]。Ｓｙｍｂｏｌｉｃｓでの配列アクセスは、その構造上、決してスタック・バッファを利用できないので(*)、かなり遅くなることが予想される。（表４）はＶＡＸ及びＳｙｍｂｏｌｉｃｓの２次元配列アクセスのベンチマーク・テストの結果である。

Ｓｙｍｂｏｌｉｃｓ		ＶＡＸ８６５０	
Normal Array	Array Register	No Optimize	Optimize
16.01	7.10	1.35	0.58

（表４）２次元配列アクセスのベンチマーク・テスト（単位：μ秒／一回のストア）

ここで使われているアレイ・レジスタとはＳｙｍｂｏｌｉｃｓ・システム・ルーチンの一つで、アレイの高速アクセスを目的としたものである。ところが、そのアレイ・レジスタを使ってもＶＡＸの１２倍、使わなければ実に２７倍の実行時間となっており、このままでは（画像）配列のアクセスを多く必要とする画像処理にリスプを使うと、パフォーマンスの点で問題があると思われる。

（＊）コントロール・スタック上にリストを作ってサブ試みたが、満足のいく結果は得られなかった。

２．５．ハンド・オプティマイズ［Ⅰ］

以上の考察をもとに、プログラム・レベルでパフォーマンスを上げる手法がいくつか考えられる。

２．２．より、基本的な処理に関しては局所化に心がければ、それだけでかなりの改善がなされる。実際、Ｓｙｍｂｏｌｉｃｓ・コンパイラのオプティマイザーも Peep Hole の手法を使っている[4]ことから明かである。また、ループ中での一定値による除算はループの先頭で逆数を計算しておいて乗算に置き換えることが出来る。

２．３．で述べた型変換の問題は、Ｓｙｍｂｏｌｉｃｓが『型変換という処理を行うために、型のチェックを行う』ことに原因があると思われる。従って、『型のチェックを行わない』型変換ルーチンを使って型変換を行うことにより、２．３．の問題のオプティマイズが出来る。具体的には、truncate ,round の代わりに sys:%convert-single-to-fixnumを、float の代わりに zl:float を使えば良い。前者は %convert-single-to-fixnum 後者は float のマイクロ・コードを直接呼ぶ関数である。（表５）にその結果を示す。real -〉integer についてはまだ十分な結果とは言えないが、型変換についての一応のオプティマイズは出来たと言えよう。

処理内容	Symbolic	ＶＡＸ	Symb/VAX
real-〉int	６．２０	０．５０	１２．４
int -〉real	２．７５	０．５２	５．３

（表５）型変換に要する時間（オプティマイズ）
（単位：μ秒／１回の実行）

２．６．ハンド・オプティマイズ［Ⅱ］

パフォーマンスに最も影響を与えるのは２．４．で述べた２次元配列のアクセスの問題点である。以下にこの問題点の改善法を順を追って説明する。

式ａ）は２次元配列をアクセスする際の通常の方法であり、式ｂ）はアレイ・レジスタを用いた方法である。式ｂ）において、すぐに解決できるところは（＊ｙｎ）のｘループからの脱出である。式ｃ）がその結果である。更に、（表１）の結果より、かけ算よりは足し算を用いたほうが速いので、式ｃ）は式ｄ）のように変形される。loop のフォームの中でのローカル変数の最適化を考慮に入れると、結局、式ｅ）の形になり、オプティマイズは終了する。また、式ｆ）は配列アクセスの row,column の順番を変えた場合である。（表６）は各式の実行時間を示している。結局、式ｅ）を使うことによりＶＡＸに較べて約６．６倍の実行時間となり、ＣＰＵのパワーの差から考えても、かなりの最適化が出来たと言えよう。

```
式ａ）
 (loop for y from 0 below n do
     (loop for x from 0 below n do
         (setf (aref ip y x) 1)))
式ｂ）
 (loop for y from 0 below n do
     (loop for x from 0 below n do
         (setf (sys:%1d-aref ip
                 (+ x (* y n))) 1)))
式ｃ）
 (loop for y from 0 below n do
     (loop for x from 0 below n
         with tmp = (* y n) do
       (setf (sys:%1d-aref ip
                 (+ x tmp)) 1)))
式ｄ）
 (loop for y from 0 below n
       for id-y from 0 by n do
     (loop for x from 0 below n do
         (setf (sys:%1d-aref ip
                 (+ x id-y)) 1)))
式ｅ）
 (loop for id-y from 0 below (* n n)
                by n do
     (loop for id from id-y
                 below (+ id-y n) do
       (setf (sys:%1d-aref ip id) 1)))
式ｆ）
 (loop for x from 0 below n
       with tmp = (* n n) do
     (loop for id from x below (- tmp x)
                 by n    do
       (setf (sys:%1d-aref ip id) 1)))
```

式ａ	式ｂ	式ｃ	式ｄ	式ｅ	式ｆ
16.01	7.10	4.16	4.04	3.83	4.35

（表６）式ａ－式ｆのベンチマーク・テストの結果
（単位：μ秒／一回のストア）

２．７．ハンド・オプティマイズ【Ⅲ】

基本的なオプティマイズは２．５．、２．６．で上げたとおりだが、それ以外にもページングやガベージを減らすためのシステム・ルーチンがいくつかあるのでそれらを紹介する。

・ｄｅｆｒｅｓｏｕｒｃｅ
・ｕｓｉｎｇ－ｒｅｓｏｕｒｃｅ

大きい配列に利用する。具体的には画像の配列などが該当する。ＳＣＬ－Ｓｐｉｄｅｒを使う際の入出力画像配列はこれらのルーチンを使って定義する方が良い。

・ｗｉｔｈ－ｄａｔａ－ｓｔａｃｋ
・ｍａｋｅ－ｓｔａｃｋ－ａｒｒａｙ
・ｍａｋｅ－ｒａｓｔｅｒ－ｓｔａｃｋ－ａｒｒａｙ

小さい配列に利用する。具体的にはマスクやヒストグラムの配列などが該当する。ＳＣＬ－Ｓｐｉｄｅｒの中のテンポラリ・アレイにはこれらのルーチンが使われている。

３．リスプによる画像処理ライブラリの構築

リスプによる画像処理が十分使用に耐えうる可能性は示されたが、リスプでの画像処理を行うには現実問題として画像処理ライブラリが必要である。しかるに、そのような基本的なライブラリはほとんどないのが現状である。そこで、我々は前述したオプティマイズ法をもとに、現在一般に使われている画像処理ライブラリ、ＳＰＩＤＥＲの全ルーチンをＳｙｍｂｏｌｉｃｓ・Ｃｏｍｍｏｎ・Ｌｉｓｐ（ＳＣＬ）にコンバージョンした。

３．１．ＳＣＬ－Ｓｐｉｄｅｒの基本仕様

コンバージョンするにあたって、Ｆｏｒｔｒａｎを Ｌｉｓｐに変更するわけであるから、言語仕様の違いなど問題も多いと思われたので、次のような基本仕様をもとにした。

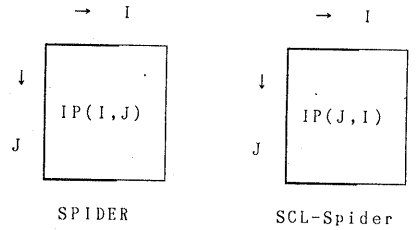
・配列の添字は０から始まる。

Ｆｏｒｔｒａｎの添字は１から始まるが、コモン・リスプでは０から始まる。ＳＣＬ－Ｓｐｉｄｅｒではコモン・リスプの言語仕様に合わせた。そのため、配列添字をデータとして持つような配列の内容は、Ｆｏｒｔｒａｎのそれと当然異なる。

・２次元配列の第１添字と第２添字を入れ換える。

Ｆｏｒｔｒａｎでは第１添字に沿ってメモリに展開されるが、コモン・リスプの仕様では第２添字に沿ってメモリに展開される。その

ため、ＳＣＬ－Ｓｐｉｄｅｒでは（図１）のような変更を行った。



（図１）２次元配列の表し方

・変更される引数は配列以外は関数の戻り値とする。

ＳＰＩＤＥＲの大部分のルーチンはサブルーチン形式となっており、戻り値は引数を変更することによって渡されている。コモン・リスプの言語仕様でこれと全く同じ形の引数の受渡しを実現すると複雑になってしまう。一方、コモン・リスプでは複数の戻り値を扱うことができるので、配列以外の変更される引数はその関数の戻り値として扱うようにした。例えば、与えられたデータの基本統計量を求めるルーチンＳＴＡＳ１は、Ｆｏｒｔｒａｎでは次のような引数指定になっている。

```
CALL STAS1(IP,ISXY,HIST,N,XLB,D,W,XMAX,
           XMIN,E,V,SIGMA,RANGE)
     IP(ISXY),XLB,D,W          : INPUT
     HIST(N),XMAX,XMIN,
       E,V,SIGMA,RANGE         : OUTPUT
```

これがＳＣＬ－Ｓｐｉｄｅｒでは次のような引数指定になる。

```
(multiple-value-setq
      (xmax xmin e v sigma range)
  (stas1 ip isxy hist n xlb d w))
```

・EQUIVALENCE には原則として Displaced-Array を使う

配列どうしの EQUIVALENCE はコモン・リスプの Displaced-Array を使えば問題ない。ところが、配列と変数の EQUIVALENCE ではコモン・リスプでインプリメントする方法はないと思われる。このような時には、もしその部分がパフォーマンスに影響を与える所なら変数で、パフォーマンスにあまり関係ない所なら配列で、それぞれ表し、EQUIVALENCE をあきらめるものとした。パフォーマンスに影響を与えるのは例えば次の例の様な場合である。

```
Fortran
      DIMENSION MASK(3,3)
      EQUIVALENCE (M11,MASK(1,1)),
     1            (M12,MASK(1,2)),
     1            (M33,MASK(3,3))
      EQUIVALENCE (K1,KK(1)),....,(K9,KK(9))

      DO 100,I1=1,N1
        CALL SET_MASK_ROUTINE(MASK)
        DO 101,I=1,ISY
          DO 102,J=1,ISX
             K1 = IP(J-1,I-1) * M11 + ....
             K2 = IP(J-1,I) * M11 + ......
              :                  :
             K9 = IP(J+1,I+1) * M11 +
             JP(I,J) = SOME_FUNCTION(KK,I1)
102       CONTINUE
101     CONTINUE
100   CONTINUE

Lisp
(loop for i1 from 1 to n1
      with k1 and k2 and .... and k9  do
   (multiple-value-bind
       (m11 m12 m13 m21 m22 m23 m31 m32 m33)
     (set-mask-routine)
    (loop for i from 0 below isy do
      (loop for j from 0 below isx do
         (setq k1 (+ (aref ip (1- i) (1- j))
                     m11....)

               k2 (+ (aref ip (1- i) j
                :       m11....)
                :          :
               k9 (+ (aref ip (1+ i) (1+ j))
                     m11 ....))
         (setf (aref jp i j)
               (some-function k1.... k9 i1))
    ))))
```

ＳＣＬ－Ｓｐｉｄｅｒで、パフォーマンスに影響を及ぼすために上の例の様な変更を行ったものにはＥＧＰＲなどがある。

・各サブルーチンの制御構造は原則としてそのまま使う。

各サブルーチンの制御構造までも最適化の候補として変更して行くと、アルゴリズム自体を変更してしまう恐れがあるので、原則としてそのような変更はしない。ただし、明らかにパフォーマンスの悪い構造の場合には全面的に変更したものもある。あるいは、パフォーマンスには関係なくとも、ＧＯＴＯ文の乱用のため制御構造が読み取りにくいものや、リスプに落としたときの局所性が大幅に妨げられるものについても全面的な変更を行った。

・エラー処理はＳＰＩＤＥＲに準拠する。

リスプ環境下では、エラー時にはデバッガーが自動的に起動され、インタラクティブにデータあるいは関数の修正を施した後、処理を再開させるのが普通である。ＳＣＬ－Ｓｐｉｄｅｒでは現在のところ、エラー処理に対する系統立った検討はなされていないので、ＳＰＩＤＥＲ同様、エラー・コードを返す形の処理を取ることにしている。そのため、引数のデータ型のチェックも行っていない。

3．2．ＳＣＬ－Ｓｐｉｄｅｒの性能

（表７）にＳＰＩＤＥＲ及びＳＣＬ－Ｓｐｉｄｅｒの一部のルーチンをそれぞれＶＡＸ８６５０、Ｓｙｍｂｏｌｉｃｓ３６７５で実行させたときの実行時間の結果を示す。（表８）は更にＦＡＣＯＭ　Ｍ－１６０ＡＤの結果も合わせて示している。

ルーチン名	Symbolics	VAX8650	Symb/VAX
ＡＦＩＮ１	１９．９	３．９	５．１
ＮＯＬＭ１	１１．９	１．７	７．０
ＨＧＥＱ１	１２．０	１．９	６．８
ＨＧＥＱ２	３０．３	４．３	７．０
ＨＧＨＹ１	１２．３	２．０	６．２
ＥＧＰＲ	１５５．２	２８．９	５．４
ＩＴＥＮ１	２２３．４	３０．３	７．４
ＡＳＭＴ	１４６．２	３１．５	４．６
ＨＹＳＭ１	１２．５	３．９	３．２
ＭＥＤＩ	６７．４	１２．７	５．３
ＥＧＲＢ	１０．５	１．６	６．５
ＥＧＳＢ１	１５．５	２．７	５．７
ＥＧＰＷ１	１４．９	２．３	６．５
ＥＧＰＷ３	２７１．２	４６．６	５．８
ＥＧＫＳ１	２７１．２	４６．５	５．８
ＥＧＲＳ１	５７．９	５．８	９．９
ＥＧＬＰ	２８．４	６．７	４．２
ＳＴＡＳ１	２０．９	２．４	８．７
ＳＴＡＳ２	４４．２	５．９	７．５
ＳＬＴＨ２	２．０	１．３	１．５
ＣＬＩＰ	２．０	１．３	１．５
ＦＨＳＴ２	２５．０	２．４	１０．４
ＦＬＷＬ１	６１．７	１４．２	４．３
ＦＬＷＬ２	１２．１	３．５	３．５
ＮＯＲＭ１	１０．６	２．０	５．３
ＡＤＤＰ１	２．３	１．６	１．４
ＭＴＲＸ	１７５５	３８６	４．５
ＩＮＲＬ	３．０	１．２	２．５

（表７）主なサブルーチンの実行時間
（単位：秒）
（注）データの配列の大きさは（５１２、５１２）濃度値は２５６

3．3．ＳＣＬ－Ｓｐｉｄｅｒの性能評価

（表７）及び（表８）からＳｙｍｂｏｌｉｃｓのＶＡＸに対するパフォーマンスを算出すると約１９パーセントとなり、マシンのハードウェア的な能力比がおよそ１：５であることから、リスプによる数値計算がＦｏｒｔｒａｎと同等のパフォーマンスで実行されていることが分かる。しかしながら、各ルーチンの結果を個別に見て行くとパフォーマンスの良いものと悪いものとの差が大きい。これはもちろん完全に最適化出来ていないということもあるが、それよりはむしろＳｙｍｂｏｌｉｃｓの初等関数がマイクロ・コード化されていないことに起因するものが多い。（表９）はマイクロ・コード化されるべき主な基本関数の現状の速さを示している。これらがマイクロ・コード化され、汎用のマシン並に速くなれば、そして除算及び整数乗算のロジックが改善されれば、パフォーマンスの点から見たリスプによる数値計算の問題点はほとんどなくなると言って良いだろう。従って、こ

	M-160AD		VAX8650		Symbolics	
	1024	2048	1024	2048	1024	2048
FFTS1	0.52	1.11	0.1	0.24	0.73	1.53
WHT1 H	0.09	0.20	0.01	0.04	0.09	0.20
WHT1 W	0.25	0.53	0.04	0.10	0.28	0.61
HAR1	0.35	0.75	0.05	0.16	0.36	0.81
SLT1	0.28	0.61	0.05	0.12	0.33	0.68
DCB1	0.73	1.52	0.12	0.32	0.97	2.05
DCF1	1.27	2.66	0.22	0.51	1.78	3.71
	128*128	256*256	128*128	256*256	128*128	256*256
FFTS2	5.54	24.8	1.11	9.20	6.19	27.1
FFTA2	81.6	-	29.5	132.	95.9	411.
WHT2 H	1.96	8.64	0.33	1.70	2.17	9.64
WHT2 W	2.16	9.39	0.36	1.87	2.69	11.5
HAR2	0.76	3.01	0.25	1.49	1.10	4.39
SLT2	2.10	9.34	0.58	3.51	2.90	12.7
DCB2	8.68	36.9	1.33	6.05	12.9	54.5
DCF2	13.6	-	3.93	18.2	14.0	60.5

（表８）直交変換ルーチンの実行時間
（単位：秒）
（注）Ｍ－１６０ＡＤのデータはＳＰＩＤＥＲマニュアルより転載

	Symbolics	VAX8650	Symb/VAX
ｓｉｎ	１２１．５	１２．７	９．６
ｃｏｓ	１３６．４	１１．３	１２．０
ｔａｎ	１６６．２	２３．３	７．１
ｅｘｐ	１２２．６	１０．６	１１．６
expt , int	６０．５	６．５	９．３
expt,float	２９３．５	１９．５	１５．１
ａｂｓ	９．９	１．１	９．０

（表９）マイクロ・コード化されるべき初等関数と型変換関数のＶＡＸとの比較。
（単位：μ秒／１回の実行）

れらの問題点が解決されれば、ＳＣＬ－ＳｐｉｄｅｒはＳｙｍｂｏｌｉｃｓ３６７５上でＦｏｒｔｒａｎ／ＶＡＸ８６５０の約４倍から５倍の実行時間で動くことになるであろう。

４．リスプ環境下における高速画像処理システム

このようにして作成された画像処理ライブラリＳＣＬ－Ｓｐｉｄｅｒで実際の処理を行う場合、Ｓｙｍｂｏｌｉｃｓ自体の遅さ故にインタラクティブな処理が出来ない場合も考えられる。汎用機ではアレイ・プロセッサ、画像処理専用機などを接続できるが、Ｓｙｍｂｏｌｉｃｓにその様な形で接続できるマシンは少ない。そこで我々はＳｙｍｂｏｌｉｃｓに接続するバック・エンド・プロセッサとしてＰＩＸＡＲを検討した。その結果、ＰＩＸＡＲは本来画像生成専用のマシンであるが、アーキテクチャとしては汎用の命令セットを持っているため、画像処理専用マシンとして使うことにより、リスプ環境下での高速画像処理を可能とすることが判明した[1]。そして、ＳＣＬ－Ｓｐｉｄｅｒの中で特に計算時間の多くかかるものについてはＰＩＸＡＲへのコンバージョンを行った。

このようにして、リスプ環境下でのインタラクティブな画像処理はＳＣＬ－Ｓｐｉｄｅｒの個々のルーチンとしては可能になったわけであるが、我々の最終目標であるエキスパート・ビジョン、画像理解等の研究を行うためには、これらの基本的処理を簡単に操作性良く行える環境が必要である。また、その様な環境で使ってこそリスプ環境下での画像処理の意味があるものと思われる。我々はこの思想をもとにＳｙｍｂｏｌｉｃｓ＋ＰＩＸＡＲでＳＣＬ－Ｓｐｉｄｅｒをマウスを使ってインタラクティブに処理させる画像処理メニュー・システム、”Ｉｍａｇｅ　ＨｏｉＨｏｉ”を構築した。

ＳＣＬ－ＳｐｉｄｅｒのＰＩＸＡＲへのコンバージョン及び”Ｉｍａｇｅ　ＨｏｉＨｏｉ”の詳細については別稿で発表の予定である。

５．おわりに

本稿ではリスプ・マシン環境下における画像処理が、システム・ルーチンとハンド・オプティマイズの利用により、速度の面において可能であることを示し、画像処理ライブラリＳＰＩＤＥＲをＳｙｍｂｏｌｉｃｓ・リスプ・マシンにコンバージョンしたＳＣＬ－Ｓｐｉｄｅｒの基本仕様と性能評価について報告した。

このようにして記号処理と信号処理の両方を自由に扱うことの出来るリスプ環境が構築されたわけだが、この環境下でのより高度な画像処理・理解の研究を行うことが今後の課題である。

謝辞

ＳＰＩＤＥＲコンバージョンの実際のコーディングを担当してくれたソフトウェア・コンサルタント社の若きハッカー、山上雅之君に深く感謝します。

参考文献

[１] 城、梅田：”リスプ環境下における高速画像処理”、昭和６２年電子情報通信学会情報・システム部門全国大会

[２] 3600 Technical Summary, Symbolics Inc, 1983.

[３] Richard Zippel,"Lisp Machines, Texas Instruments and Symbolics"

[４] David Moon, "Architecture of the Symbolics 3600",12th Int'l Symp. Computer Architecture,1985, pp.76-83

[５] 城、梅田：”ＡＣＢ（アレイ・キャッシュ・バッファ）による高速アレイ・アクセス”、情報処理学会第３５回全国大会