記号処理 37－4
（1986. 6. 20）

Ｌｉｓｐの並列実行

－処理系の設計・試作－

今　昭　　松山　隆司　　伊藤　貴康
（東北大学　工学部）

本報告では、Ｌｉｓｐの並列実行モデルに関する考察を行うとともに、我々の研究室で開発したマルチ・マイクロプロセッサ上でインプリメントされた並列Ｌｉｓｐインタープリタの性能について述べる。まず、Ｌｉｓｐの並列実行モデルに関しては、先の報告［５］において述べた多引数関数の引数の並列評価に加え、条件式の並列評価の有効性について考察する。次に、４台のＭＣ６８０００を持つマルチ・マイクロプロセッサ上でＳ式を並列に評価する並列Ｌｉｓｐインタープリタにおける並列処理方式を述べ、その性能評価を行う。さらに、Ｌｉｓｐに対する並列処理構文を導入し、その意義と効果について検討を加える。

PARALLEL EVALUATION OF LISP PROGRAMS

－DESIGN AND IMPLEMENTATION－

Akira KON, Takashi MATSUYAMA, Takayasu ITO

Department of Information Engineering
Tohoku University

This paper discusses models of parallel evaluation of Lisp programs, and describes design principles and implementation techniques of a parallel Lisp interpreter implemented on the experimental multi-microprocessor system developed in our laboratory. First we discuss the effectiveness of parallel evaluation of conditional expressions in Lisp; parallel evaluation of arguments of a function was discussed in our previous paper[5]. In the performance evaluation of the parallel Lisp interpreter, we performed various experiments to examine the effectiveness of our scheme of parallel evaluation. Then, we introduce a programming construct to specify parallelism in Lisp programs, and discuss its effectiveness with several experiments.

はじめに

人工知能ソフトウェアを作成するに当たってＬｉｓｐは欠かすことのできないプログラミング言語である。一般に人工知能研究ではソフトウェアが大規模化・複雑化する傾向が強く、言語処理系の高速化に対する要求が強い。このため長年に渡ってＬｉｓｐ処理系の高速化に対する研究がさまざまな角度から進められてきた。

高速化の方式を大別すると、

(1)処理系のインプリメンテーション技法の効率化。

(2)プロセッサ自身をＬｉｓｐ向きアーキテクチャにする。

(3)多数のプロセッサによる並列処理。

に分けられる。(1)に関しては線形スタックを用いたシャローバインディングやデータ構造に関する研究がなされている。(2)に関しては高機能なＬｉｓｐマシンがすでに商品化されその有効性が実証されている。一方(3)の並列処理に関してはいくつかの試みがなされているが[1][2][3] 今のところ決定的な方式が存在せず、実験段階に留まっている。

本稿では、Ｌｉｓｐの並列処理方式に関する考察を加えるとともに、我々の研究室で開発したマルチマイクロプロセッサシステム上に実現した並列Ｌｉｓｐインタープリタを用いた実験結果について報告する。以下１章では、先の報告[5]で述べた多引数関数の並列評価の理論的考察に引き続き，条件式の並列評価に関する検討を行う。２章では基本となる並列Ｌｉｓｐインタープリタの詳細を述べる。３章ではＬｉｓｐプログラムに対する並列構文の導入とその実現法について述べ、４章において試作した並列Ｌｉｓｐインタープリタの評価実験の結果を報告する。

第１章　条件式の並列評価

Ｌｉｓｐプログラムの並列評価を考える場合、プログラムに内在する並列性の抽出法とその有効性をまず検討する必要がある。前回の報告[5]では、Ｌｉｓｐのインタープリタ関数ＥＶＡＬに対する各種の並列処理方式を抽象的データフローモデルによって表し各方式の有効性を比較した。特に多引数関数の引数の並列評価に関してはプロセッサ台数やプロセッサ間の通信コストの影響についても詳細に解析した。本章では、Ｌｉｓｐの条件式の並列評価の有効性について考察する。

1.1　条件部の並列評価

条件式の並列評価としてまず、次のようなものを考える。Ｌｉｓｐにおけるｃｏｎｄ式、例えば、

$$(\mathrm{cond}\ (p_1\ b_1)\ (p_2\ b_2)\ \cdots\ (p_n\ b_n)) \qquad \cdots\cdots(1)$$

なる式はp_1～p_nの式を評価した結果がｎｉｌか非ｎｉｌかによってb_1～b_nのいずれかを評価する（あるいは何にも選ばれずにｎｉｌを返す）、といった条件分岐を行う式である。ここでp_1～p_nを条件部、b_1～b_nを実行部と呼ぶ。p_1～p_nを評価して分岐先を決定するのに要する時間が条件分岐のための処理時間である。ここではp_1～p_nを並列評価することによって条件分岐のための処理時間を短縮することを考える。

ｃｏｎｄ式(1)における条件分岐に要する時間を考えよう。各条件部p_iを評価する時間をt_iとする。次に各p_iが非ｎｉｌである確率をP_iとおく。ｃｏｎｄ式の評価の結果どの実行部が選ばれるかによって次のような表が作れる。

分岐先	選ばれる条件	選ばれる確率
b_1	p_1	P_1
b_2	$\neg p_1 \wedge p_2$	$P_2(1-P_1)$
・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・		
b_n	$\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_{n-1} \wedge p_n$	$P_n\prod_1^{n-1}(1-P_i)$
none	$\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_n$	$\prod_1^{n}(1-P_i)$

表１　ｃｏｎｄ式における分岐とその条件

以上のように実行部b_iが選択されるに当たって条件p_1～p_iすべてが関与する。確率B_iを、

$$B_1 = P_1$$
$$B_i = P_i\prod_1^{i-1}(1-P_j) \qquad (2 \leqq i \leqq n)$$
$$B_{none} = \prod_1^{i}(1-P_j)$$

とおき、どの条件が成り立つかよりもどこに分岐がなされるかに着目する。各実行部に分岐がなされるのに要する時間は逐次処理、並列処理の場合それぞれ次のようになる。

分岐先	逐次処理	並列処理	分岐の確率
b_1	t_1	t_1	B_1
b_2	t_1+t_2	$\max(t_1, t_2)$	B_2
・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・			
b_n	Σt_i	$\max\ t_i$	B_n
none	Σt_i	$\max\ t_i$	B_{none}

表２　分岐に要する時間

この表から逐次処理、並列処理におけるｃｏｎｄ式の条件分岐時間の期待値が求められ、それらは

$$\text{逐次処理}\quad \sum_1^n B_i \sum_1^i t_j + B_{none}\sum_1^n t_j$$

$$\text{並列処理}\quad \sum_1^n B_i \max_j t_j + B_{none}\max_j t_j$$

となる。ここで、各条件判定時間がほぼ同じで各実行部への分岐の確率が均等に行われると仮定する。そのとき、

$$t_1 \simeq t_2 \simeq \cdots \simeq t_n = t$$
$$B_1 \simeq B_2 \simeq \cdots \simeq B_n \simeq B_{none} = 1/(n+1)$$

のように考え、それぞれの期待値が次のように求まる。

逐次分岐時間の期待値　$tn(n+3)/2(n+1)$

並列分岐時間の期待値　t

以上より、

逐次処理による分岐時間／並列処理による分岐時間

$= n(n+3)/2(n+1) \geqq n/2$

と求められる。すなわち条件部をｎ個持つｃｏｎｄ式はおおむねｎ／２の並列性を持つと考えることができる。

ただし、この方式の並列処理をｃｏｎｄ式に適用するに当たって注意すべき点がある。それはｃｏｎｄ式の次のような二面性に起因する。

(1)　ｃａｓｅ文的なｃｏｎｄ式。

各条件部の条件成立が各々独立しているようなｃｏｎｄ式。このような場合は上で述べた並列処理を適用するに当たってなんら問題はない。

(2)　逐次実行によるセマンティクスに依存したｃｏｎｄ式。

この端的な例としては条件部でのｓｅｔｑ等による副作用を持つ式の評価が挙げられる。また多くの場合ｃｏｎｄ式は前の条件部の評価がｎｉｌを返したものとして次の条件部が書かれる。すなわち次のようなプログラムは暗に条件部の逐次実行を仮定している。

```
(cond ((atom x) ...)
      ((atom (car x)) ...)
      ((atom (caar x)) ...)
      ....    ))
```

プログラム3　逐次実行を仮定して書かれたｃｏｎｄ式

このようなｃｏｎｄ式は逐次処理ではエラーを起こさないが条件部を並列処理するとエラーを起こす場合がある。

以上から、条件部の並列処理を実現するに当たってはすべ

てのｃｏｎｄ式に対して並列処理を適用するのではなく、並列処理が可能なｃｏｎｄ式を抽出し、そのようなｃｏｎｄ式に対してのみ並列処理を行うようにする必要がある。

1.2　実行部の並列評価

ｃｏｎｄ式を並列評価するに当たって実行部までを並列に求めることも考えられる。[4] すなわち分岐先の決定以前に全ての実行部の評価を開始し、分岐が決定した時点で必要な結果を選択するというものである。このような処理を行　ことによって条件分岐に要する時間は理想的には０になる。

しかし実行部がｎあるとき(n-1)個の評価が無駄に終わること、現実にはプロセッサの数が有限であること、実行部に副作用を持つ式が現れると意味的に正しくなくなること等、実行部の並列評価の実現に関しては困難な問題がある。

第２章　マルチマイクロプロセッサを用いた並列Ｌｉｓｐ処理系

本章では我々の研究室で開発したマルチマイクロプロセッサ上で実現された並列Ｌｉｓｐ処理系について述べる。このシステムでは[5]で考察した多引数関数の引数の並列評価とともにガーベージコレクションを複数台のプロセッサが並列に実行する。

2.1　ハードウェアシステムの構成

図４にハードウェアシステムの構成を示す。システムは入出力、及びＳＰＵの管理を行うＭＰＵ(MC68000 8MHz)、リスト処理及びガーベージコレクションを分担して受け持つ４台のＳＰＵ(MC68000 12.5MHz)、そして、セル領域となる４つにバンク分けされた共有メモリ(CM 4MByte)を持つ。共有メモリとＳＰＵの結合はクロスバー方式による。また、共有メモリのバンク分けは、図５(b)のように大きく４分割するモードと(a)のように８Byte毎にバンクを振り分けるモードのいずれかを選択することができる。Ｌｉｓｐでは(a)の分割モードを選択することによってメモリアクセスの各バンクへの分散をはかっている。

各プロセッサは相互に割り込みをかけることができ、共有メモリ上に通信データを格納する領域（通信ポート）を用意し、相互に割り込みを行うことによってプロセッサ間のデータの通信が可能になっている。

2.2　キューを用いた並列Ｌｉｓｐインタープリタ

ここでは図４のハードウェアシステムを用いて実現した並列Ｌｉｓｐインタープリタの詳細について述べる。今回インプリメントした並列インタープリタでは各ＳＰＵが自身のローカルメモリ上にインタープリタプログラムを持ち相互に通信することによって入力されたＳ式を並列評価する。

並列インタープリタを実現するに当たって
(1)通信コストをできるだけ抑える。
(2)並列性を必要以上に抽出しない。
(3)特殊な構文を用いないで書いたプログラムでも並列に処理する。
といった、３つの方針を設定した。

2.2.1　並列処理の概要

まず並列処理がどのように成されるかを説明する。Ｌｉｓｐ言語は関数型言語であり、関数の組み合わせによってプログラムが構成される。関数の中には引数を複数持つものがある。関数を評価する場合まず関数の引数が評価され値が求められる。関数本体においてはその求められた値を用いて処理が進められる。今回インプリメントした並列インタープリタでは、多引数関数の引数の評価を並列に行う。

以下では１つのＳ式の評価を行う処理をプロセスと呼ぶ。多引数関数の引数を並列評価する場合、その関数式全体を評価している親プロセスから各引数を評価する複数の子プロセスが生成される。新たに生成されたプロセスは共有メモリ上のキューを介して他のプロセッサに割り当てられる。親プロセスを処理している親プロセッサはすべての子プロセスが結果を返すまで「待ち」状態に入る。しかし、「待ち」状態の間プロセッサが停止していてはプロセッサの数が有限であることから、有効な並列処理が実現できない。そこで親プロセッサはキューに積まれた他のプロセスの処理に入る。

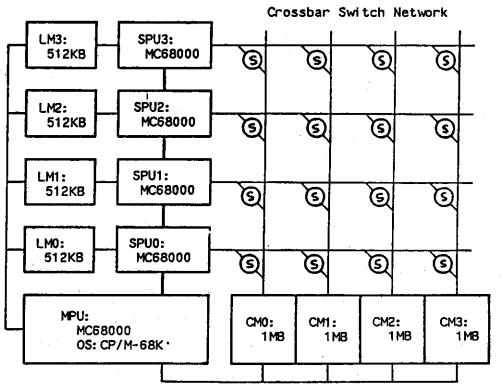

図４　ハードウェア構成

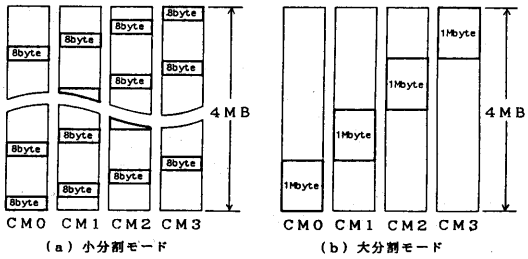

図５　メモリのバンク分けにおける２つのモード

並列インタープリタでは、プロセッサに対して(1)並列モード（Ｐモード）と(2)逐次モード（Ｓモード）の２つのモードを与える。プロセッサは初期状態としてはＰモードであるが、子プロセスを生成することによってＳモードに移行する。Ｓモードでは多引数関数の評価に際して子プロセスを生成せず、すべての引数の評価を自身で行う。ＳモードからＰモードへの移行は以前生成したすべての子プロセスが終了し結果が返されることによってなされる。

この２つのモードの導入には次のような利点がある。
(1)１つのプロセッサが生成する子プロセスは逐次モードの制限によって１世代のみになるのでプロセスの生成消去の処理が簡潔になる。すなわち１つのプロセッサが複数の世代の子プロセスを生成した場合、どの世代のプロセスが最も早く完了するか分からない。このため、すべての子プロセスが完了した場合、その世代の処理を円滑に継続するためにはプロセッサの機械語レベルの環境（例えばシステムスタック）のアロケーション等も面倒をみなくてはならない。したがって、プロセス生成のために大きなコストがかかる。しかし１つのプロセッサが１世代の子プロセスしか生成しないのであればそのような心配はない。
(2)逐次モードの導入によってプロセス生成の指数的爆発に歯止めをかけられる。生成されるプロセスの数が増大すると各

プロセスの大きさ（計算に要する時間）が小さくなり、通信コストやプロセス生成のオーバーヘッドが増加し、全体的な処理効率が低下する。したがってこのような歯止めは必要であると考えられる。具体的に生成されるプロセスの上限は、関数の引数の数がおおむねｋであるときプロセッサ台数ｎに対して最大でｎｋである。

2.2.2 インプリメンテーションの詳細

子プロセスの生成は評価すべきＳ式や評価の行われる環境等生成すべきプロセスに関する必要な情報を共有メモリ上にあるグローバルなキューに書き加えることによってなされる。プロセスのプロセッサへの割り当てはアイドル状態にあるプロセッサがキューをアクセスし、格納してあるプロセス情報を読み出すことによってなされる。アイドル状態は(1)初期状態(2)子プロセスを生成し結果待ちである状態、の２つが当たる。

キューのアクセスは排他的に行われるが、キュー上にプロセスが積まれていない状態ではアイドル状態のプロセッサがキューを頻繁にアクセスすることになる。すなわち、アイドルプロセッサのキューアクセスがメモリ・プロセッサ間のバスを頻繁に占有することになる。このことは他のプロセッサのメモリアクセス速度の低下をまねく。そこでアイドルプロセッサはキュー上に待ちプロセスが無いことを検知すると同じキュー上に自分のプロセッサＩＤを書き込み停止する。またプロセッサが子プロセスをキューに書き込む場合、待ちプロセッサがキュー上に存在していればそのプロセッサに割り込みをかけることによって停止していたプロセッサを起動し子プロセスの処理を開始させる。

以上のようにプロセスのプロセッサへの割り当ては個々のプロセッサがキューを読み書きし必要があれば相互に通信を行うといった方式によってなされる。キュー上に記録されているのは処理を待っているプロセスかあるいは処理待ちのプロセッサかのいずれかである。プロセスには処理待ちの他に実行中、子プロセスの結果待ちの状態、があるがこれら２つの状態にあるプロセスはすでにキュー上からは消されている。またプロセス全体を管理するプロセステーブルのようなものは作らなかった。

2.2.3 ｃｏｐｙの動作と処理時間

以上説明した並列Ｌｉｓｐインタープリタの下でｃｏｐｙを走らせることを考えよう。

```
(defun copy (x)
  (cond ((atom x) x)
        (t (cons (copy (car x)) (copy (cdr x))) ))
```

ﾌﾟﾛｸﾞﾗﾑ6　ＣＯＰＹ

ＣＯＰＹが(((a) b) (c) d))のようなバランストツリーを入力として実行されると、プロセス生成に制限を設けない場合は図７のようなプロセス生成の様子をしめす。しかし本インプリメンテーションにおいてはＰモード／Ｓモードによる制限があるのですべてのｃｏｎｓが並列評価されるわけではなく、図８のようなプロセス生成がなされる。図７において木のノードに当たる部分ではｃｏｐｙの引数ｘがアトムでないと判定され(copy (car x)), (copy (cdr x))の２つの式を評価し、返される２つの値をｃｏｎｓする。この処理の処理時間をＮとする。木のリーフに当たる部分では引数ｘがアトムであると判断されｘ自身を値として返す。この処理時間をＬとする。

ｃｏｎｓの部分において２つの引数 ((copy (car x)), (copy (cdr x))) は、プロセッサがＰモードである場合、子プロセスとして他のプロセッサにその評価が依頼される。プロセッサがＳモードであるなら自分で２つの引数を逐次評価する。プロセッサがＰモードであり子プロセスを他のプロセッサに処理させるためにはキューのアクセスやプロセッサ間通信が

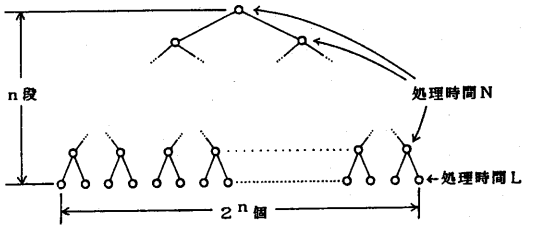

図７　プロセス生成に制限を加えない場合

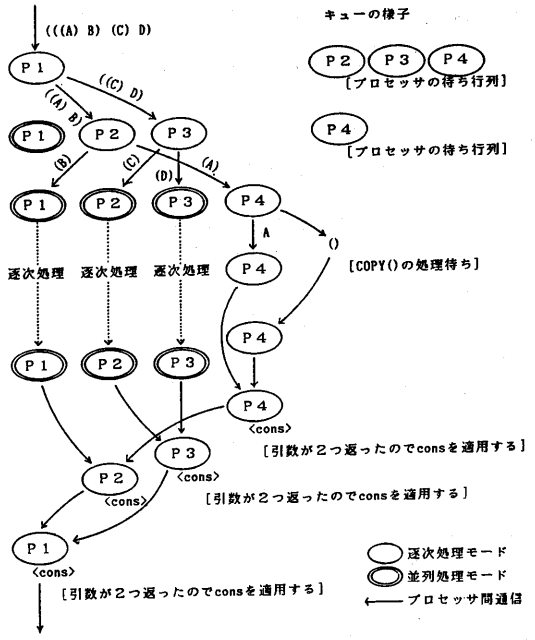

図８　並列インタープリタによって
(copy '(((A) B) (C) D))が処理される様子

入るので、ある程度の時間がかかる。それをＣとする。

ｃｏｐｙが深さｎのバランストツリーを処理するとき本並列インタープリタでは必ず図８のような形の処理過程が得られる。したがってその処理時間はＮ，Ｌ，Ｃを用いて次のように表せる。

$3(C+N)+2(C+2^{n-3}(N+L)-N)$
$=5C+N+2^{n-2}(N+L)$

2.2.4 基本インプリメンテーションでの問題点

上で述べた並列インタープリタの実現法は以後の基本となるので基本インプリメンテーションと呼ぶ。基本インプリメンテーションにおいては次のような問題点が考えられる。
(1)多引数関数に対して必ず並列処理を試みる。したがって負荷の配分が均等に行われない場合が考えられる。
(2)特にその顕著な例として１台のプロセッサのみが逐次モードで走り他は「待ち」であるという状態がありうる。

2.3 ガーベージコレクション（ＧＣ）

現在インプリメントされているＧＣは、ＳＰＵのいずれかがフリーセルの底つきを検知した時点で開始される、いわゆるＳｔｏｐ＆Ｍａｒｋ方式のものである。ＳＰＵがフリーセルの残りがないことを検出するとＭＰＵに割り込みをかけそれを知らせる。ＭＰＵはすべてのＳＰＵに割り込みをかけることによって並列ＧＣを起動する。並列ＧＣにおいてはＭＰ

Uと４台のＳＰＵが以下に述べるようにマーキングとスイーピングにおける処理を分担する。

2.3.1 マーキング

ＬｉｓｐのＧＣにおけるマーキングはまず有効なセルにマークを付けそのセルに格納されているポインタの指すセルに対してマーキングを行うといった再帰的手法で行われる。一番初めにマークを付ける位置をルートという。本Ｌｉｓｐ処理系ではルートとして
(1)パラメタスタック内のデータ
(2)ｏｂｌｉｓｔ
(3)キュー内のＬｉｓｐデータ
(4)通信ポート内のＬｉｓｐデータ。
がある。このうち(1)のパラメタスタックは各ＳＰＵのローカルメモリ上にあるので各ＳＰＵがそこをルートとしてマーキングをするのが自然である。そこで各ＳＰＵのパラメタスタック内のデータに関しては各ＳＰＵ自身がマーキングを行い(2)～(3)に関してはＭＰＵがマーキングを行うことにした。

2.3.2 スイーピング

スイーピングはデータ領域を走査して未使用セルがあればフリーリストにつなげるという単純な作業である。ここでは単にｃｏｎｓセル領域を４等分し、それぞれの処理をＳＰＵが分担するという方法で処理の分割を試みた。なおそれと並行してｓｙｍｂｏｌセル領域のスイーピングをＭＰＵが行う。

スイーピングの並列処理は通信を殆ど起こさずオーバーヘッドが小さいのでかなり並列処理による効果があがると考えられる。

第３章 並列構文の導入とその実現

２章で述べた基本インプリメンテーションでは通常のＬｉｓｐプログラムの並列評価法として、多引数関数の引数の並列評価について考察した。ここでは、基本インプリメンテーションを基に、Ｌｉｓｐに対する並列構文の導入及び、Ｌｉｓｐプログラムの新たな並列処理方式について考察し、その意義と効果について検討する。

並列構文は２つの目的で導入される。ひとつは負荷の分散を効率よく行うためであり、もうひとつは並列に動作している複数のプロセスに共有される資源（大域変数等）を保護するためである。

3.1 負荷分散のための並列構文

3.1.1 引数の並列処理 ｐｃａｌｌ[1]

基本インプリメンテーションにおいては多引数関数の引数はすべて並列評価されていた。ところが例えば、(＜ｘｙ) や (cons x (heavy-func z))のような式を並列評価するのは効率が悪い。前者においてはｘやｙの値を求める処理時間が、それらを他のプロセッサに行わせるために要する通信時間より小さく、結果的に並列処理は逆に処理時間を増大させる方向に働く。後者の式の評価に関しては第１引数ｘを評価する処理時間が第２引数を評価する時間に比べて極端に小さいため、プロセッサに与えられる負荷が不均質になる。

こうした問題を避ける方法として、ｐｃａｌｌ文による引数の並列評価の指定がある。

(fn x y z) は
(pcall fn x y z)

と書くことによって引数が並列に評価される。また

(pcall (lambda (x y) . BODY) X Y)を
(plet ((x X) (y Y)) . BODY)

のように書くと分かりやすいのでｐｌｅｔという構文も導入する。

ｐｃａｌｌ、ｐｌｅｔを適当に挿入することにより上で述べた２つの事態は回避される。

3.1.2 先行評価

関数を実行するに当たって、逐次処理系では引数の評価を終えて初めて関数本体の実行に入るが、関数本体の実行にあたっては引数が求まっていなくともある程度実行が進められる場合や、全部の引数の値が確定していなくとも本体の実行を終えられる場合がある。こうした関数本体の先行評価によって並列処理の効率が向上すると考えられる。

3.1.3 条件判定の並列処理 ｐｃｏｎｄ

第１章で検討したようにｃｏｎｄ式の条件部の並列評価に際してはｃｏｎｄ式の表す意味によって有効であるものとそうでないものがある。条件部の並列処理が有効であり並列処理によって効果があがりそうなｃｏｎｄ式に対しては並列処理の実行を指示するためのｐｃｏｎｄなる構文を与える。ｐｃｏｎｄはｃｏｎｄの代わりに用いられる。

ｐｃｏｎｄを用いた式

(pcond (p_1 b_1) (p_2 b_2) ⋯ (p_n b_n))

が

(plet ((x_1 p_1) (x_2 p_2) ... (x_n p_n))
  (cond (x_1 b_1) (x_2 b_2) ⋯ (x_n b_n)))

と異なるのはｐｃｏｎｄ式が一種の先行評価を行っている点である。すなわち分岐先の決定には必ずしもすべての条件部の評価が必要なわけではなく実際には一部の条件部が評価された時点で処理が先に進む。

3.2 排他的クロージャ

実用的なプログラムを書く場合大域的な状態に変化を与えたいことがある。しかしＬｉｓｐの大域変数はいわば裸の状態であり並列処理環境ではその更新には危険が伴う。

例えば変数ｘに初期値０が束縛されているときに次のような関数の処理を考える。

(pcall fn X Y)

ＸとＹは異なるプロセッサ上で並列に処理されうる。Ｘ，Ｙの中に(setq x (1+ x))のような式が含まれればｘの値の参照、更新が同時に行われる可能性がある。この例としては

(pcall list (setq x (1+ x)) (setq x (1+ x)))

がある。この式はｘの値が２だけインクリメントされることを期待して書かれているが評価のタイミングによっては１しかインクリメントされないこともある。

このような危険性を排除するために大域変数の値の更新に関してその受け口を限定することを考える。例えば今のようなカウンタ動作に関しては

```
(defun icounter ()
  (let ((x 0))
    #'(lambda (m)
        (cond ((eq m 'up) (setq x (1+ x)))
              ((eq m 'current-value) x)
              ((eq m 'clear) (setq x 0)) ))))
```

と定義しｘをクロージャ内に閉じ込めておき変数ｘに関するアクセスは関数ｉｃｏｕｎｔｅｒが生成するクロージャを介さないとできないようにしておく。このクロージャ自身にセマフォを置きクロージャへのアクセスを排他的に行うことによってカウントアップ時のｘの参照・計算・更新のサイクルが安全に行われる。我々の並列Ｌｉｓｐ処理系では、このような排他的クロージャを生成する関数の関数名をｆｕｎｃｔｉｏｎ＊として、通常のクロージャを生成する関数ｆｕｎｃｔｉｏｎとは区別し、略記も#'ではなく#‘を用いる。上

の定義で#'を#'とした時、次の

```
(setq counter (icounter))
(pcall list
  (funcall counter 'up)
  (funcall counter 'up))
```

は必ずxの値を2だけインクリメントする。

3.3 並列構文の実現

先に述べた並列構文のうちｐｃａｌｌ及び排他的クロージャに関してはインプリメントを終えている。先行評価および条件文の並列評価に関しては現在インプリメントを行っているところである。以下では、ｐｃａｌｌ，クロージャについては実現法、ｐｃｏｎｄ、先行評価については実現の際の注意点を述べる。

ｐｃａｌｌの実現は容易に行える。基本インプリメンテーションではインタープリタ関数ＥＶＡＬ内のＥＶＬＩＳ部分で引数が複数であれば並列処理を試みていた。この部分での並列処理を、すべての場合に適用するのではなくｐｃａｌｌ構文で指定された場合のみ行うようにすればよい。

クロージャはベクタを用いて実現されており、排他的クロージャはベクタ内にアクセス権のためのセマフォビットを置くことによって実現している。アクセス権の確保はＭＣ６８０００のＴＡＳ(Test And Set)命令によってなされる。

ｐｃｏｎｄと先行評価の実現においてはともに以下のような点を考慮する必要がある。

(1)未確定値の取り扱い。

(2)関数本体の実行完了後における未終了子プロセスの削除。

(1)に関してはタグフィールドにそのデータの値が確定しているかどうかのビットを設けることによってそのデータが未確定であるかどうかの判定を行う。プロセッサが未確定値に出会うとそこから先の処理が続けられなくなり、その処理は一端中断する。処理の続行をどのタイミングで行うかが通常問題になるが、基本インプリメンテーションにおいては先に挙げた並列処理の制限（Ｐモード／Ｓモード）が存在しているので、先行評価を行うプロセッサは子プロセスを生成したプロセッサしかありえない。子プロセスによって値が確定していくたびに親プロセッサに通信が行われるので、そのときに中断した処理の継続を判断すればよい。

(2)に関しては基本インプリメンテーションにおいてプロセステーブルを用いなかったので実現が一見困難であるように思える。しかし、これに関しても並列処理に対する制限を設けたために以下のようにプロセステーブルが無いことはあまり問題にならない。

ｎ台のプロセッサでのインプリメンテーションにおいて、

(a)実行状態のプロセスはたかだかｎである。

(b)子プロセスを待っている状態にあるプロセスはたかだかｎである。

(c)処理待ち状態のプロセスはすべてキュー上に書かれている。

(a)～(b)の状態にあるプロセスに関してはプロセスに関する情報（親プロセッサ識別子等）はすでにキュー上から削除されているが、それらのプロセスを処理しているプロセッサはもちろん自分が処理しているプロセスに関する情報を持っている。したがってキューを読むことと、各ＳＰＵに処理中のプロセスの情報を問い合わせることによってすべてのプロセスに関する情報を得ることができる。このように全プロセスに関する情報を比較的容易に得ることができるので、プロセステーブルを持たないことの損失は小さい。

あるプロセスを削除するときにはそのプロセスのすべての子孫プロセスも削除しなくてはならない。基本インプリメンテーションでは子孫がたかだかｎ代にしかならないのでその処理はさほど重たくはないと考えられる。

第４章　システムの評価

ここでは、２章３章で述べた各種の並列処理方式の有効性を実験結果によって示す。

4.1 通信コストの測定と逐次モードの有効性の検討

並列処理のオーバーヘッドの１つとしてプロセッサ間の通信コストがある。以下では２章で述べたｃｏｐｙを例にして実際の通信コストの測定と逐次モード導入の有効性について考察する。

４台のプロセッサを用いた基本インプリメンテーションによって、ｃｏｐｙが左右均等にバランスしている２進木を処理するときの処理時間は、

$$3(C+N)+2(C+2^{n-3}(N+L)-N) = 5C+N+2^{n-2}(N+L)$$

となる。逐次処理による時間を求めるために実験では４台のプロセッサのうち３台を強制的に停止させて処理時間を測定した。これは疑似的逐次処理とみなせ、その処理時間は

$$(C+N)+2(C+2^{n-1}(N+L)-N) = 3C-N+2^{n}(N+L)$$

と表せる。

これらの式と実測値をもとにＮ（ノードでの処理時間）、Ｌ（リーフでの処理時間）、Ｃ（通信コスト）を求めると、Ｎ≒０．７７、Ｌ≒０．４５、Ｃ≒０．２２　(msec)となった。

通信時間のリスト処理時間に対する比は当初の予想を下回り１／３であった。通信時間とリスト処理時間に関する検討[5]によると、逐次モードの導入によりプロセスの生成を制限したとき、不要な通信を削除したことによる処理速度の向上は１．５倍程度になる。

プロセスの生成に制限を加えない場合は負荷の配分が均等になりやすい。すなわち並列処理が可能な部分はすべてプロセスとしてキューに載せておくことによって各プロセッサが常に仕事が得られる状態にしておくことができる。したがって、通信時間がある程度以下であるならばかえって生成されるプロセスの数に制限を加えないほうが良いとも考えられる。ただし以上の議論は１回当りの通信時間が比較的小さいということに基づいており、２章で述べたように、本並列Ｌｉｓｐ処理系におけるような逐次モードの導入が１回当りの通信時間の減少させるように働いている（プロセス生成のためのオーバーヘッドが小さくなっている）という点にも十分注意しなくてはならない。

いずれにせよ、逐次モードを持たない場合の並列処理方式、あるいは別の形でプロセス生成に制限を加えた場合の並列処理方式については追ってインプリメンテーションを通して実験評価を行う予定である。

4.2 性能評価

試作システムを用いて実際にプログラムを走らせた結果が表９である。ここでは表をもとにリスト処理におけるプロセッサ台数の影響、並列構文の効果、並列構文のオーバーヘッドについて考察する。

＜測定方法＞

処理時間の測定は

(1)逐次処理の処理時間、

(2)並列構文指定無しの場合の処理時間、

(3)並列構文を用いた場合の処理時間、

の３種について行った。現在稼働している並列インタープリタはｐｃａｌｌを用いた構文指定による並列処理しか受け付けないため、具体的には次のように測定した。

(1)逐次処理

ｐｃａｌｌをまったくいれない。

(2)並列構文指定無し

基本インプリメンテーションにおいて並列処理をする部分す

べてにｐｃａｌｌを挿入する。
(3)並列構文指定有り
効果がありそうな部分にｐｃａｌｌを挿入する。
(2)に関してはｐｃａｌｌが入っている分だけ基本インプリメンテーションにおける処理よりもオーバーヘッドがかかると考えられるがこのことについては後に検討する。

4.2.1 プロセッサの台数効果

表９で扱っているプログラム(program10~13)は処理の並列化を行い易いものであるが、４台のプロセッサによる並列処理における処理速度の向上は平均して約２．６倍くらいであった。プロセッサ台数が４倍になったのに処理速度の向上が４倍にならないのは、
(1)常にすべてのプロセッサが動作しているわけではない。
(2)逐次処理においては不要である通信コストが含まれる。
(3)共有メモリのアクセスに競合が生じる。
の３つの理由が考えられる。しかし(2)については先の測定結果による計算から１回当りの通信時間はそれほど大きくないことがわかっている。ｓｒｅｖなどでは通信がかなりの回数で起きているが、それでも通信のための延べ時間は全体の処理時間の１０％にも満たない。したがってプロセッサ台数と同じの速度向上が得られないのは殆ど(1)か(3)の理由によるものであると考えられる。

4.2.2 構文による指定

３章で述べたように多引数関数の引数をすべて自動的に並列評価するのをやめ、並列構文を用いると次のような得失がある。
(1)引数が十分単純でその評価時間が通信時間に比べて小さい時には引数を並列評価しないほうが全体の処理時間が小さい。並列構文を用いることにより、そのような無駄な並列処理を削除できる。
(2)プロセッサの負荷が著しくかたよっていると、並列処理による効果が低くなる。特に今回のように並列／逐次・モードの切り替えがあるときは、全体の処理時間を増大させてしまう。並列構文によってある程度そのような事をふせげる。
(3)（欠点）ｐｃａｌｌ構文を導入することによってインタープリタにおける構文解釈のオーバーヘッドがかかる。
(1)に関してはｔａｋの(ｘｙｚ)を並列処理することを考える。この式を評価するのにｙ，ｚをキューに載せ他のプロセッサに評価させるよりは自分で逐次評価したほうが速い。例えばｔａｋの(ｘｙｚ)の部分だけにｐｃａｌｌを挿入すると通信は６３６０９回行われ処理時間は逐次処理の場合よりかえって２０秒増大する。
(2)に関してはｓｒｅｖ６の例が興味深い結果を与えている。すなわちＥＶＬＩＳで必ず並列処理を行う事をやめｐｃａｌｌを適当な部分に挿入する。プログラム上はｐｃａｌｌの挿入箇所を減らすことになるが、実際の処理では反対に通信回数が増え、処理時間が２割ほど減る。これは処理の配分がより均等に行われたことを示す。
(3)に関しては次のような方法でオーバーヘッドを調べた。まず、ＳＰＵのうち３台の動作を止める。１台しか動作していない状態でプログラム１２(b)を走らせる。プログラム１２(b)にはｐｃａｌｌによる並列処理の指定があるが実際には１台しか動作していないので逐次処理と実質的には同じである。したがってｐｃａｌｌの分だけプログラム１２(b)での処理においては時間がかかる。余計な時間をｐｃａｌｌの回数で割ったものが１回当りのｐｃａｌｌのオーバーヘッドである。このようにしてｐｃａｌｌのオーバーヘッドは２５μsecと求められた。この値は十分小さいものであり、(1)(2)の利点を考えるとｐｃａｌｌの存在価値は大きい。

		並列処理		逐次処理	tulisp/E15[8]
		並列構文での指定 有り	並列構文での指定 無し		
tak 18-12-6	処理時間	21,993	22,810	63,562	25,980
	通信回数	(6)	(29)	(0)	
	効率	２．８９	２．７４	1	
srev 5	処理時間	206	246	525	260
	通信回数	(53)	(65)	(0)	
	効率	２．５５	２．１３	1	
srev 6	処理時間	745	951	2,125	1,020
	通信回数	(140)	(107)	(0)	
	効率	２．８５	２．２３	1	
fib 19	処理時間	3,381	3,652*1	11,327	6,160
	通信回数	(4)	(15)	(0)	
	効率	３．３５	３．１０	1	
qsort 50	処理時間	426	586	625	280
	通信回数	(4)	(274)	(0)	
	効率	１．４６	１．０６	1	

表９ リスト処理における台数効果
注*1 この値は微妙なタイミングの違いにより4,783になることがある

```
(defun tak (x y z)
  (cond ((not (pcall < y x)) z)
        (t (pcall tak
                  (pcall tak (1- x) y z)
                  (pcall tak (1- y) z x)
                  (pcall tak (1- z) x y)))))
```
Program 10. TAK

```
(defun rev (x)
  (cond ((atom x) nil)
        (t (pcall app
                  (rev (cdr x))
                  (pcall cons (car x) nil)))))

(defun app (x y)
  (cond ((atom x) y)
        (t (pcall app
                  (rev (cdr (rev x)))
                  (pcall cons (car (rev x)) y)))))
```
Program 11. Slow REVerse

```
(defun fib (n)
  (cond ((pcall eq n 0) 1)
        ((pcall eq n 1) 1)
        (t (pcall + (fib (pcall - n 1))
                    (fib (pcall - n 2))))))
```
Program 12. FIBonacci

```
(defun qsort (l)
  (cond
    ((null l) nil)
    (t ((lambda (p)
          (pcall nconc
                 (qsort (car p))
                 (pcall cons
                        (car l)
                        (qsort (cdr p)))))
        (pcall partition
               (cdr l) (car l))))))

(defun partition (l x)
  (cond
    ((null l) (pcall cons nil nil))
    (t ((lambda (p)
          (cond
            ((pcall < (car l) x)
             (pcall cons
                    (pcall cons
                           (car l) (car p))
                    (cdr p)))
            (t (pcall cons
                      (car p)
                      (pcall cons
                             (car l)
                             (cdr p))))))
        (pcall partition (cdr l) x)))))
```
Program 13. Quick SORT

プログラムは構文指定無しの並列処理の場合のもの(a)、
構文指定有りの実験の際は、下線が引いてないｐｃａｌｌは取り除かれる(b)。
逐次処理の実験では、すべてのｐｃａｌｌが取り除かれる(c)。

4.2.3 ガーベージコレクションにおけるプロセッサ台数の効果

並列ＧＣにおけるプロセッサ台数の効果を調べるためＧＣを１台で行えるようにした。実験ではマーキング、スイーピングのすべてをＭＰＵ１台で行うという逐次ＧＣを用いた。また試作ハードウェアシステムにおいては共有メモリのアクセス競合を極力防ぐ目的で小分割モードを用いている。このメモリ分割モードの効果についてもあわせて実験を行った。
測定結果を表１４に示す。この結果を見る前に注意しなくてはならない点がある。まず逐次ＧＣは本来比較のためにはＳＰＵのうちの１台に行わせるべきである。ところがパラメタスタックがローカルメモリ上におかれるため、１台のＳＰＵからすべてのパラメタスタックをルートとしマーキングすることは不可能である。したがって逐次ＧＣにはＭＰＵを用いることにした。しかし、ＭＰＵのクロックは８ＭＨｚでありＳＰＵの１２．５ＭＨｚと比べると若干遅い。（メモリのアクセス時間も考えると１．２５倍程度）
表によると並列処理によりマーキング時間は約１．３倍ス

イーピング時間は約４．５倍向上する。スイーピング時間の向上が４倍を越えるのは今述べたクロックの違いと、並列ＧＣではＳＰＵの他にＭＰＵもスイーピングに参加することが理由として挙げられる。いずれにせよスイーピングにおいては予想通りプロセッサ台数と同程度の効果があげられた。しかし、マーキングは並列処理の効果があまりでておらず、その原因を詳細に分析する必要がある。

（インターリーブ）

メモリをインターリーブすること（図５の小分割モード）によりスイーピングにおいて約１．２倍の速度向上が得られる。しかしリスト処理においてはインターリーブは数％の効果を持つに過ぎない。スイーピングにおいてインターリーブが効果的なのはスイーピング処理のほとんどがメモリアクセスであるためである。以上のことから４台のプロセッサでの並列リスト処理においては共有メモリのアクセス競合がさほど起こらないと考えられる。

第５章　考察と今後の課題

本報告においては条件式の並列評価の検討と、並列Ｌｉｓｐインタープリタを用いたＬｉｓｐプログラムの並列実行の評価実験について報告を行った。今回の検討から次のようなことが言える。

(1)ｃｏｎｄ式の並列処理効果。

条件部がｎ個あるｃｏｎｄ式はおおよそn/2程度の並列性を持つ。

(2)４台のプロセッサを用いた並列Ｌｉｓｐインタープリタでは小規模なプログラムに対しては約２．５倍の処理速度の向上が得られる。

(3)並列構文の導入により、各プロセッサへの負荷分散の均一化、並列処理のオーバーヘッドの削減、大域変数に対する排他的アクセスが可能となる。

(4)並列処理すべき箇所を指定する構文ｐｃａｌｌを用いることにより、無制限な多引数関数の引数の並列評価に比べ、数％から２０％の処理速度の向上が得られる。

(5)Ｓｔｏｐ&Ｍａｒｋ方式のガーベージコレクションにおいてＭＰＵもあわせて５台のプロセッサで行うことによりスイーピングにおいてはプロセッサの台数倍近くの処理向上が得られた。

(6)共有メモリを８Ｂｙｔｅ毎にインターリーブすることによりスイーピング速度が約２割向上した。ただしリスト処理においてはインターリーブによる処理速度の向上は数％にとどまる。これらは共有メモリの８Ｂｙｔｅ毎のバンク分けがバス競合の回避にある程度効果的であることと、リスト処理がそれほどバス競合を起こしていないことを示している。

今後の課題として次のようなことが残された。

(1)プロセッサの負荷の均一化

今回の実験を通じて、並列処理の効率の向上のためにはプロセッサの負荷の均一化が重要な問題であることが分かった。今後は、並列処理のオーバーヘッドのより詳細な解析と負荷均一化のためのアルゴリズムを検討する必要がある。

(2)既存のプログラムから並列Ｌｉｓｐプログラムへの自動変換。

本報告では並列構文の有効性を示した。既存のプログラムを効率よく実行するには、プログラムを解析し並列構文を挿入する必要がある。こうした効果的な並列構文の自動挿入は今後の課題である。

(3)先行評価、並列条件分岐の実現。

今回はこれらの実現についての留意点を述べるに留まったが、今後インプリメントの詳細を詰めるとともにその効果について検討する予定である。プログラム１３のような形のクイックソートのプログラムは先行評価をうまく用いることによって処理速度がより向上する例であると考えている。

			並列処理	逐次処理
小分割	fib 19	処理時間	3,537 (165)	3,861 (483)
		ＧＣ回数	(3)	(3)
		ﾏｰｸ	３０	３７
		ｽｲｰﾌﾟ	２５	１２４
	srev 6	処理時間	796 (54)	905 (163)
		ＧＣ回数	(1)	(1)
		ﾏｰｸ	２８	４０
		ｽｲｰﾌﾟ	２６	１２３
大分割	fib 19	処理時間	3,665 (184)	3,950 (477)
		ＧＣ時間	(3)	(3)
		ﾏｰｸ	２８	３６
		ｽｲｰﾌﾟ	３３	１２３
	srev 6	処理時間	831 (61)	933 (160)
		ＧＣ時間	(1)	(1)
		ﾏｰｸ	２７	３７
		ｽｲｰﾌﾟ	３４	１２３

表１４　ＧＣにおける台数効果

（注）　カッコ内はＧＣのトータル時間である。
マーク、スイープ時間はＧＣ１回当たりの時間

(3)大規模なプログラムを用いた評価。

今回は評価用のプログラムとして比較的小規模なものばかりを使用したが、並列Ｌｉｓｐ処理系の有効性を示すには大規模なプログラムに対する処理効率の向上を明らかにする必要がある。先に述べたような並列Ｌｉｓｐプログラムへの自動変換ソフトウェアを用いれば既存のプログラムを並列Ｌｉｓｐプログラムとして動かすことが可能になろう。

最後に試作ハードウェアの製作にあたり多くの協力をしていただいた伊藤研究室の大友技官、沖電気工業株式会社，ハードウェアの設計製作を行った研究室の黒川氏に感謝いたします。なお本研究の一部は文部省科学研究費特定研究601100 3の助成によってなされた。

［参考文献］

[1]R. H. Halstead, Jr. : Implementation of Multilisp, ACM Symp. on Lisp And Functional Programming (1984)

[2]R. P. Gabriel, J. McCarthy: Queue-based Multi-processing Lisp, ACM Symp. on Lisp And Functional Programing (1984)

[3]安井ほか：LISPでの並列処理における動的特性とEVLISマシンの構成，情報処理学会記号処理研究会資料，10-4 (1979)

[4]伊藤，和田：LISP関数の並列実行モデルとその評価，情報処理学会記号処理研究会資料，26-5 (1983)

[5]黒川ほか：MC68000を用いた並列処理システムの試み，電子通信学会計算機アーキテクチャ研究会資料，EC85-44 (1985)

[6]今，伊藤：MC68020での高速なLisp処理系の設計と試作，情報処理学会第３２回全国大会，4G-8 (1986)

[7]H. Okuno et al: The Report of the Third LISP Contest and the First Prolog Contest, 情報処理学会記号処理研究会資料，33-4 (1985)