マルチマイクロプロセッサによる LISPマシン

LISP Machine Implementation on Multi-Microprocessor System

薄　隆，田丸喜一郎，所　真理雄
Takashi USUKI, Kiichiro TAMARU, Mario TOKORO
慶応義塾大学　工学部
Faculty of Engineering, Keio University

1. はじめに

近年LISPシステムのハードウエア化に関する研究が盛んになっているが，いずれも高速化を目指しており，ハードウエア・スタックを設けたり，ファームウエアを利用するなどの方法をとっている．LISPはその言語体系から，高速実行をはかるためには，再帰的手続き処理のためのスタック操作，ダイナミック・アロケーション処理のためのメモリー管理処理が効率よく行なえなければならない．しかし現在の計算機ではこれらを効率よく処理することは難かしく，そのため特別にハードウエアを設けることによりLISPマシンとして設計されているのが実状である．

我々は現在3台のプロセッサからなる共有メモリー型マルチプロセッサを持っている．このシステムを利用し，特別にハードウエアを設計することなくLISP 1.5レベルのプログラムが実行できるシステムを実装した．3台のプロセッサからなるマルチプロセッサに適合するように，LISP処理を機能的に3分割して実装している．

2. 設計方針

LISPの特徴を列挙すると次のようになる．

i) 再帰的関数が自由に定義できる．

ii) 動的記憶領域割付けを行なう．

以上のような特徴を効率よく実現するためには，以下の事項が考慮されなければならない．

i) スタック操作が容易に行なえる．

ii) メモリー管理が効率よく行なえる．

通常スタック操作を容易に行なうには，特別なハードウエア・スタックを設けるか，若しくは例えばPDP－11のように強力なメモリー・アドレシング機構を持つことが要求される．ここでは特別なハードウエアを付加しないという方針から，スタック処理の強化に主眼を置いたLISPマシンとはしないことにした．

次にメモリー管理に関して，LISPでは動的メモリー割付けのために，不要になったメモリー・セルを回収しなければならない．この方法として，リファレンス・カウンタを設ける方式や，ガーベージ・コレクションを行なう方式がある．ガーベージ・コレクションの方式に関しても，タグを用いる方式とビットテーブルを利用する方式がある．これらの方法を実現するためには特別なハードウエアは必要としない．このため，ここではメモリー管理を効率よく行なうことに主眼を置いてLISPマシンを設計実装することにした．

リスト構造のメモリー空間で不要になったメモリー・セルを回収する方法には前述のように，

i) リファレンス・カウンタを設ける，

ii) ガーベージコレクションを行なう，

がある．i)の方法では，カウンタのために特別なメモリー空間を必要とし，普通，ii)の方法よりもメモリー空間を必要とするが，全体的には処理時間をそれほど喰わず，また長時間処理が中断することもない．ii)の方法では，ビット・テーブルを用いたり，タグやフラグを設けたりするが，i)よりもメモリー要求量は少ない．しかしながらメモリーがなくなったときに行な

う一回のガーベージ・コレクションにかなりの時間がかかる．メモリー空間が大きいほどガーベージ・コレクションの回数は減るが，一回の処理時間が長くなる．そこで，ここではガーベージ・コレクションを並列に実行することによりLISPの処理の中断時間を最小にする方針とした．

3. 処理の分割

並列ガーベージ・コレクションに関しては，これまでに数多くのアルゴリズムが研究されている．基本的にはDijkstraのアルゴリズムを用いているものが多い．ガーベージ・コレクションは次の2つのフェーズからなる．

i) マーク・フェーズ，

ii) コレクション・フェーズ，

マーク・フェーズでは，ルートから参照されているリストをたどり，必要なセルであるという印を付ける．ガーベージ・コレクションがLISPインタプリテーションと並列に行なわれていたとすると，リストのつなぎかえなどにより参照されているリストにマークが付かなくなったり，一度つけたマークが消されたりしないように，インタプリテーションとガーベージ・コレクションで同期を取る必要がある．

具体的には，インタプリタがメモリーの内容を書換える動作と，ガーベージ・コレクタのマーク付け動作が同じメモリー・アドレスに対して行なわれた場合と，マークしている1つのリストをたどり終らないうちにリストのつなぎ替えが起こった場合に不合理が生じる可能性がある．前者に対しては何らかの方法でクリティカル・リージョンを設けなければならない．後者に関しては，インタプリタがリスト書換えに関する情報を残しておいて，ガーベージ・コレクタにマーク付けの指示をしなければならない．

以上の考察から，処理をインタプリテーションとガーベージ・コレクションというレベルで分割するよりも，インタプリテーションとメモリー管理という方向で分割した方が実装が容易であることが判る．メモリーの書換えを行なう関数は，CONS，RPLACA，RPLACD，SET，SETQの5種があるが，これらの関数をインタプリテーション・プロセスから取り除き，メモリー管理プロセスの中に組込むことによりこの考え方は実現できる．

次にLISP 1.5にはいくつかの入出力処理用の関数があるが，これらも1つのプロセスとして分離することにより，並列性を生かすことが可能である．

従って，次のように処理を機能分割し，それぞれを1台のプロセッサに割当てることにより，LISPの処理を並列処理させる方針を採ることにした．

i) インタプリテーション
リストの評価，実行．

ii) メモリー管理
ガーベージ・コレクション，
CONS，RPLACA，RPLACD の実行．
（SET，SETQは未実装）

iii) 入出力処理
リストの読込み，書出し．
ファイル処理．

4. ハードウエアシステムの構成

現在，図1に示すようなマルチプロセッサ・システムが実験用として構成されている．

i) PM/I

デバイス制御用に設計されたプロセッサで，ビット処理能力が優れている．内部に16個のレジスタ，16レベルのハードウエア・スタックを持ち，500 nsecのマイクロ命令サイクルで動作する．コントロール・メモリーは24 bit x 1 KWである．

ii) PM/II

汎用バイトスライス・マイクロプロセッサの試作機で，EX（Executor），SC（Sequence Controller），IOC（I/O Controller）の3モジュールからなっている．強力な演算命令を持ち，内部に24個の汎用レジスタ，8個の特殊レジスタ，16レベルのマイクロアドレス用ハードウエア・スタックを持つ．600 nsec のマイクロ命令サイク

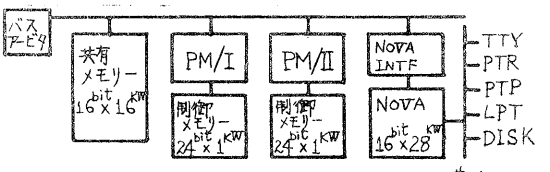

図1 実験用マルチプロセッサシステムの構成

ルで動作し，コントロール・メモリーは24bit×1KW で，ダイナミック・ライタブルである．

iii) NOVA

メイン・メモリー 28KW で，入出力機器としてディスク（2.4MW），ラインプリンタ，TTY，紙テープ・リーダ，パンチャーが付いている．基本命令サイクル 1.2μsec で，RDOSが稼動している．

iv) 共有メモリー

16KW×16bit で，メモリープロテクション機能などは持っていない．

v) バス・コントローラ

各プロセッサに非同期バスの使用権を与えたり，また存在しないアドレスへのアクセスがあった場合にはQUIT信号を発生する機能を持つ．

バス使用権の優先順位は，PM/I，PM/II，NOVAの順に与えられている．

vi) P-レジスタ

PM/I，PM/II およびNOVAインタフェースには1語分のレジスタと，このレジスタの状態を示すフラグ（PRF）がある．このレジスタはまとまってプロセッサ間交信に用いることができる．（図2） あるプロセッサが他のプロセッサのP-レジスタに1語のデータを書込もうとしたとき，相手プロセッサのPRFがオフなら書込みに成功するが，オンだと書込みが行なわれず，RJCT 信号が返され，書込みに失敗したことが知らされる．1回の転送には約 600 nsec 要する．

5．処理方式

ハードウエアの構成および設計方針より，各プロセッサへの処理の分担を次のようにした．

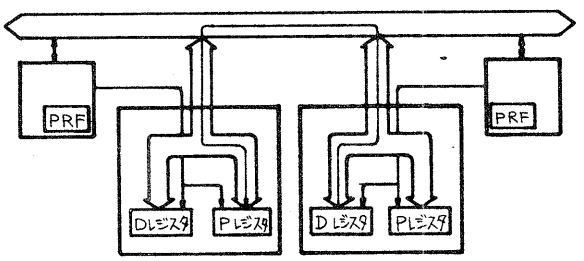

図2．P-レジスタとPRF

PM/II　－　インタプリテーション
PM/I　－　メモリー管理
NOVA　－　入出力処理

それぞれのプロセッサを，インタプリト・プロセッサ（IP），ストレージ・マネージメント・プロセッサ（SMP），入出力プロセッサ（IOP）と呼ぶ．

処理内容

i) IP

LISP処理全体のコントロールを行なう．すなわち，各プロセッサに起動をかけたり，関数の実行依頼を行うとともに，リストの操作を行ない，評価する．各プロセッサへの処理の依頼はPレジスタを用いて行なわれる．関数の評価形式はEVALQUOTEタイプである．また関数 read はIOPと並列して動作し，IOPが作成したプログラムソースの中間形をリストに変換する．

ii) SMP

共有メモリーに対する書込権を持っている．ガーベージ・コレクションの他にメモリーへの書込みを行なう関数 CONS，RPLACA，RPLACD を実行し，結果を各プロセッサに返す．またリストのルートであるスタックへのアクセス制御を行なう．常時ガーベージ・コレクション・サイクルを実行し，他のプロセッサから処理の依頼があると一時中断して依頼された処理を優先的に実行する．

iii) IOP

入出力の関数，ファイル処理を行なう．S式で記述されたプログラムを，左カッコ，右カッコ，ドット，アトム（数値を含む），に分類し，入力された順にそれぞれ中間形に変換したのち，IPに送るべく，共有エリアであるメッセージ・バッファに転送する．またこの時，文字アトムに対しては，そのフルワードを共有メモリー内には作成せず，実体をIOPのメモリー内に置く．これにより共有メモリーが有効に利用される．

処理手順

図3に3台のプロセッサが並列に動作する状態を示す．まずS式がIOPに入力される．このときIOPはS式を中間形に変換しつつ，数値変換，文字アトムの作成を行ないながら

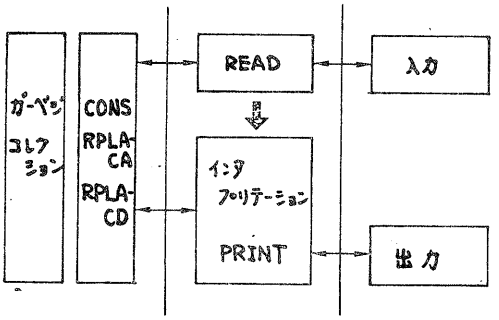

図3　並列動作

入力が終了するまで処理を続ける．これはIPからのREAD命令で動作を開始し，終了は左右カッコの数が一致した時点で判断される．またアトム作成中にCONSを実行する必要が生ずるとSMPに依頼し，実行結果を得る．この動作と同時にIPは中間形から実行可能なリストへ変換を行なう．この時もCONSの実行はSMPに依頼する．IPはREADが完了すると今作成したリストの評価にうつる．この時点でSMPはガーベージ・コレクションの実行サイクルにはいる．IPはEVAL-QUOTEで入力された$式の評価を行ない，入出力があればIOPに，またCONS，RPLACAなどのリスト構造変更の関数実行の必要があればSMPにそれぞれ処理を依頼する．SMPは，IPがリストの評価中常にガーベージ・コレクションを行なう．IPが評価を終えるとIOPにPRINTの実行を依頼して結果を出力し，次の入力に備えるために再びREAD命令の実行を依頼する．以上のサイクルが繰返される．このようにして，入出力処理，$式の評価，ガーベージ・コレクションが並列に実行される．

プロセッサ間交信

IPがIOPやSMPに関数の実行を依頼したり，結果を得るためにはプロセッサ間交信が必要となる．これにはPレジスタおよび共有メモリーの一部を使用して行なわれる．関数の実行依頼には関数の種類および引数の受渡しを行なう必要があるが，ここでは，引数を共有メモリーの交信領域にあらかじめセットしておき，次に関数の種類とプロセッサの種類を示すコマンドを相手プロセッサのPレジスタに転送する方式を採った．実行を依頼したプロセッサは，結果が自分のPレジスタにセットされるまで，その処理が待状態となる．

また2台のプロセッサが同時に同じプロセッサにアクセスした場合には，どちらか一方だけが処理され，他方にはその間RJCT信号が返り，RJCT信号が受けたプロセッサは要求が受付けられるまで再試行する．従ってその間，そのプロセッサの処理はホールド状態となる．

図4にコマンドのフォーマット，図5に交信パターンを示す．

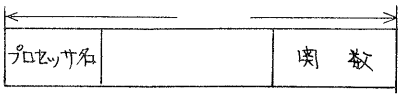

図4　プロセッサ間コマンドのフォーマット

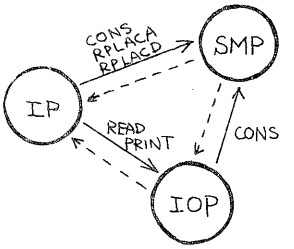

図5　プロセッサ間交信パターン

ガーベージ・コレクション

基本的にはDijkstraの並列ガーベージ・コレクション・アルゴリズムを採用している．メモリー管理を1台のプロセッサが行うのでマーキング・アルゴリズムを簡単化することができる．

並列ガーベージ・コレクションの基本となっているのは，インタプリタが評価中にリストの構造を更新したり新しくリストを作成した場合に，ガーベージ・コレクタがこの時もしマーク・フェーズを実行中であれば，リスト変更状態を示すスタックに変更に関する情報を格納し，後でマークを行なうことにより，少くとも必要とするリストには必ずマークが付くことを保証していることにある．

SMP

```
begin
  repeat
    while NEWFRAME do
      begin
RESTRT: disable;
        MARKFRAME POINTER := STACKTOP;
        NEWFRAME := false;
        enable;
        while MARKFRAME POINTER > 0 do
          begin for I := 1 to FRAMESIZE do
            if SKIPFRAME then begin
              send answer (INTERPRETER);
              goto RESTRT end
            else
              MARK;
            MARKFRAME POINTER := MARKFRAME POINTER
                                          - FRAMESIZE
          end
      end;
    GARBAGE COLLECTION
  forever
end
```

SMP when interrupted

```
begin
  repeat
    receive (INTERPRETER, NEWSP, OLDSP);
    NEWFRAME := true;
    if OLDSP ≧ MARKFRAME POINTER then
        send answer (INTERPRETER)
      else
        SKIPFRAME := true;
  forever
end
```

Procedure POP called by IP

```
procedure pop;
begin
  SP := SP - FRAMESIZE
end
```

Procedure PUSH called by IP

```
procedure push;
begin
  OLDSP := SP;
  NEWSP := SP + FRAMESIZE;
  send (MEMORY MANAGEMENT, NEWSP, OLDSP);
  receive answer (MEMORY MANAGEMENT);
  PUSH DATA;
  SP := NEWSP;
  STACKTOP := NEWSP;
end
```

図5　スタックの排他アクセス
アルゴリズム

図5にリストのルートとなるスタックの排他アクセスアルゴリズムを，図6にマーキングおよびコレクティングアルゴリズムを，さらに図7に関数 CONS, RPLACA, RPLACD の処理アルゴリズムを示す．

i) スタック・アクセス

スタックの状態はガーベージ・コレクションの動作とは無関係に増減を繰返している．このような状態で1関数のコール毎に作成されるフレーム内の変数をもとにしたマーク付けを始めるわけであるが，1フレーム内のマーク付けが完了しないうちにフレームが消滅してしまうと以後のマーク付けが正しく行なわれなくなってしまう．このためこのスタックへのアクセスは排他的に行なわれなければならない．図6のアルゴリズムはこの排他制御をできるだけ効率よく実現しようとしたものである．

インタプリタは常にガーベージ・コレクタにスタックのプッシュが可能かどうかを問うようになっている．すなわち一度プッシュした内容はスタックをポップしてもスタック・ポインタが変化するだけで，再びプッシュしなければメモリー上に履歴が残っている．従ってガーベージ・コレクタが1フレームを識別することが可能である．1フレームのマーク付けが完了したならば，次にマークすべきフレームは，もしスタックがポップされて減少したならばそのフレームからスタートすればよいし，スタックに変化がない，あるいはさらにプッシュされたならば今行なったフレームの次のフレームのマーク付けを開始すればよい．

ii) マーキングおよびコレクティング

リストへのマーク付けはルートから順に CAR をたどりながら各セルにマークを付け，CAR をたどり切った時点で CDR について同じ操作を行なえばよい．

コレクションについては，フリーリスト領域全体に渡り，不要マークとなっているセルをフリーリスト・セルのマークに変えながら現在のフリーリストに接続して行けばよい．

iii) CONS, RPLACA, RPLACD

これらの関数を実行するということは，新しいリストを作成する，または既存のリストを

変更するということであるから，これらの変更されたリストに対しても正しくマークが付くことを保証しなければならない．従って，ガーベージコレクタがマークフェーズの動作中にこれらの関数を実行した場合には，その引数および新しいセルをルートとしマーク付けが行なえるように一時スタックに保持して

```
procedure mark(ROOT);
begin
  if ROOT.TAG = USED then return
  else begin
         ROOT.TAG := USED;
         if ROOT = ATOM or
                   ROOT = NIL then return
       end;
  ROOTA := CAR(ROOT);
  mark (ROOTA);
  ROOTD := CDR(ROOT);
  if ROOTD = NIL or ROOTD = ATOM then
                                    return
  else mark(ROOTD);
end

procedure garbage collection;
begin
  A: I := FREE_LIST_BASE_ADDRESS;
  if I.TAG = NOT_USED then
  begin
    CDR(I) := FREE_LIST;
    I.TAG = FREE;
    FREE_LIST := I;
  end
  else if I   MAX_ADDRESS then return
       else I ;= I + 1; go to A;
end
```

図6　マーキングおよびコレクティングのアルゴリズム

```
procedure cons;
begin
  receive (INTERPRETER, X, Y);
  PTR := FREE_LIST;
  FREE_LIST := CDR(FREE_LIST);
  if FREE_LIST = NIL then ERROR;
  CAR(PTR) := X;
  CDR(PTR) := Y;
  if GC_MODE = ACTIVE then
                        PTR.TAG := NOT_USED
  else if GC_MODE = MARK_PHASE then
          PUSH_MARK_STACK(PTR, X, Y)
  else PTR.TAG := USED;
  send answer(INTERPRETER);
end

procedure rplaca, rplacd;
begin
  receive (INTERPRETER, X, Y);
  if GC_MODE = MARK PHASE then
             PUSH_MARK_STACK (X, Y);
  execution function;
  send answer (INTERPRETER);
end
```

図7　CONS，RPLACA，RPLACDの処理アルゴリズム

おく．この処理により並列にガーベージ・コレクションを行なっても必要なリストにマークが付くことが保証される．

メモリーマップ

図8に現在のシステムのメモリーマップを示す．スタックは共有メモリー内にインタプリタ用，ガーベージコレクタ用にそれぞれ1本づつのスタックが用意されている．さらにIOPのメモリー内に入出力用のスタックがある．

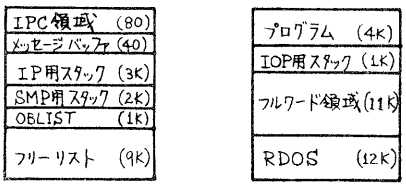

図8　メモリーマップ

6. システムの実装と考察

現在までに実装された関数は次のとおりである．

```
SUBR
apply      atom       attrib     car
cdr        cons       eq         eval
evalquote  evcon      evlis      get
maplist    nconc      null       pair
print      prog2      quote      read
rplaca     rplacd     sassoc

FSUBR
cond       list

ARITHMETICS
addl       difference             minus
minusp     numberp    subl       zerop
```

これらの関数はすべてマイクロプログラムでコーディングされている．IPで実行される関数だけで約1Kステップ，またガーベージ・コレクションとCONSなどの実行で約0.5Kステップのマイクロプログラムになっている．コントロールメモリーの容量の制約から，これ以上機能を増すことはできない状態である．現在のシステムはPROG形式を許していないが，コントロールメモリーの容量が増した時点で実装する予定である．数値は単精度整数型で取扱っているが，現在の機能でもLISP 1.5の基本的

な機能は備えており，ハノイの塔，アッカマン関数などの副題的なプログラムの実行は可能である．

LISPの処理をこのような機能的な分割により並列処理することにより，少なくとも長期間インタプリテーションが中断されることはなくなると予想される．しかしフリーリストを大量に消費しないような問題に対しては，プロセッサ間の交信のオーバヘッドの分だけシングルプロセッサによる処理時間より長くなるはずである．並列処理の効果が期待できるのはオーバヘッドの合計がガーベージ・コレクションに要する時間以内に収まる時であるが，通常の問題に関してはこれが満たされる．

インタプリテーション（CONS，RPLACA，RPLACDを含む）とガーベージ・コレクションという分割を行なって並列処理を行なう場合と比較すると，どちらもCONSを行なうときにはプロセッサ間の交信が必要である．しかしメモリー管理という観点から分割し，実装したためメモリーアクセスに対する相互排他のアルゴリズムの実現が容易である．

また今回の実装をもとに，より高速化するための改良点をいくつか述べる．すなわちLISPのように再帰呼出しの多い言語では，特にスタック操作が強力であることが望ましい．マイクロプログラムレベルでの容量の大きなハードウエアスタックを備えるとかなりの高速化が図れるであろう．また，マルチプロセッサ構成の場合プロセッサ間の交信が頻繁に生じるが，このとき交信媒体として共有メモリーを使用することはメモリーアクセスの回数の増加を招く．より早く応答するためには，直接相手のプロセッサに転送できる装置があるほうが有利である．

7. おわりに

現在本LISPマシンは，ハードウエアを含めてより高速化を図るために調整中である．また，新しいマルチプロセッサシステムへの移植を検討中であり，移植が実現すれば強力なLISPシステムとなるであろう．

謝辞

日頃御指導戴いている相磯秀夫教授，またユーザとして協力を戴いている研究室の方々に感謝する．

参考文献

1. M. Tokoro, et al, "PM/II -Multiprocessor Oriented Byte-Sliced LSI Processor Modules," Proc. NCC '77, June 1977.
2. E.W. Dijkstra, et al, "On-the-fly Garbage Collection: An Exercise in Cooperation," Lecture Note in C.S., No. 46, 1975.
3. H.T. Kung, S.W. Song, "An Efficient Parallel Garbage Collection System and its Correctness Proof," Proc. 18th Annual Symposium on Fundations of Compt. Sci., IEEE Compt. Society, Oct. 1977.