システムソフトウェアと
オペレーティング・システム　67－25
（1994. 12. 9）

分散メモリ型並列計算機上の並列記号処理システム

福居 宏和，新實 治男，柴山 潔

京都工芸繊維大学 工芸学部 電子情報工学科

あらまし

高並列計算機 AP1000 上に Lisp の方言の一つである Scheme を核とした並列記号処理システムを作成した。各プロセッサに独立したメモリ空間を持つ Scheme を実装し、複数のプロセッサで式を同時に評価させることで並列性を引き出す。また、並列に評価する単位を手続きとして、その並列に評価される手続きを「第１級」のオブジェクトとして扱えるようにし、Scheme の手続きと同様の形式で他のプロセッサに存在する手続きを呼び出すことを可能にした。

和文キーワード　Scheme、並列記号処理、AP1000

A Parallel Symbol Processing System on a Distributed Memory Parallel Computer

Hirokazu Fukui, Haruo Niimi, Kiyoshi Shibayama

Dept. of Electronics and Information Science,
Faculty of Engineering and Design,
Kyoto Institute of Technology

Abstract

We propose a parallel symbol processing system, to execute a parallel program described in Scheme, on the highly parallel computer AP1000.

We implement a Scheme processor on each processor element with a local and independent memory space. Parallelism is extracted by evaluating expressions on multiprocessors concurrently.

All procedures, which are evaluated in parallel, are treated as the "first-class" objects. And any procedure which exists in other processors can be called in the same manner as a local procedure in Scheme.

英文 key words　Scheme, Parallel Symbol Processing, AP1000

1 はじめに

高並列計算機 AP1000 上に Lisp の方言の一つである Scheme[1] を核とした並列記号処理システムの作成を行う。種々の Lisp 方言の中から Scheme を選んだのは、仕様が簡潔に定義されており、言語仕様が小さいというのが最大の理由である。

一方、AP1000 は分散メモリ型の並列計算機で、各要素プロセッサが 2 次元トーラスのネットワークで接続されている。

我々はその各要素プロセッサに Scheme の処理系を実装し、AP1000 の通信用ライブラリを用いて、各要素プロセッサ上の処理系間で通信および協調処理を行えるように、Scheme の拡張を行う。

2 Scheme

Scheme は静的なスコープを持つプログラミング言語である。Scheme の実装では適切に**末尾再帰**を行うことが要求される。これによって、構文上は再帰的な手続きによる繰り返し計算を一定の空間内で行うことができる。また、Scheme の大きな言語機能として**継続**がある。これは、非局所的脱出、バックトラック、コルーチンなどを含むさまざまな高度な制御構成を実装するのに役に立つ。

Scheme では、生成されたオブジェクトは無制限の存在範囲を持つ。そのため、図 1に示すプログラムのように、let を用いて局所変数 x を持つ局所環境を作り出し、その環境のもとで lambda を用いて手続きを生成することで、手続き内でのみ参照可能な局所状態変数が作られる (図 2参照)。

```
(define accumulator
  (let ((x 0))
    (lambda (value)
      (set! x (+ x value))
      x)))
```

図 1: 局所状態変数のプログラム例

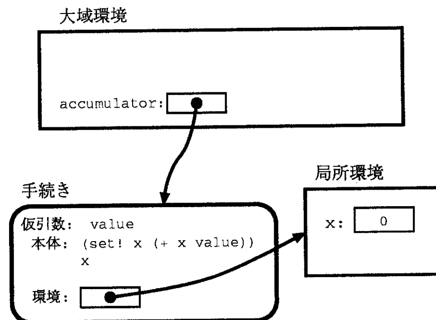

図 2: 手続き accumulator についての環境

3 Scheme の拡張による並列処理

3.1 手続きの並列実行

この Scheme の手続きを拡張し、他のプロセッサで実行できるようにした手続きを**拡張手続き**呼ぶことにする。また、呼び出された後、実行が進行中の拡張手続きを**プロセス**と呼ぶことにする。

拡張手続きは拡張した構文により、Scheme の手続きオブジェクトから生成する。Scheme の手続きオブジェクトには、手続きに渡す仮引数、手続きを呼び出したときに評価される式、そして、それが評価される時の環境についての情報が含まれている (図 2参照)。評価される時の環境とは、大域的な環境のことではなく、図 2の x の位置する局所環境のことである。プロセスの実行にはそれに加えて、どのプロセッサで実行するか、そして、呼び出し元のプロセスとどのように同期するかといったパラメータがある。

それらのパラメータをまとめ、プロセスの実行する直前の状態をオブジェクトとし、Scheme 上で扱えるようにする。これにより、同じプロセスを何度も呼び出す場合、プロセスの実行を始めるまでのオーバヘッドを軽減することができる。このプロセスを実行直前のオブジェクトのことをを**拡張手続きオブジェクト**と呼ぶことにする。もちろん、拡張手続きオブジェクトは「第 1 級」として扱われ、手続きやシンボルや

数値などと同様に変数に代入したり、関数の引数として渡したり、関数の返り値とすることができる。

拡張手続き呼び出し、すなわちプロセスの実行は、通常の手続き呼び出しと同じ形式で行うことができる。ここで、拡張手続きを呼び出した側のプロセスを親プロセス、実行させられるプロセスを子プロセスとする。引数は呼び出し側の環境、つまり親プロセスの環境で評価され、子プロセスを実行するプロセッサに渡される。そして、その引数を仮引数に束縛して、その環境のもとで子プロセスの実行がなされる。子プロセスの実行の最後の評価の値が親プロセスのプロセスの実行手続きの返り値になる。

3.2 親プロセスと子プロセスの同期

親プロセスと子プロセスの同期方式に次の3通りを用意する。

(a) 親プロセスは子プロセスの終了を待ち(wait)、その子プロセスの返り値を受け取る。これは遠隔手続き呼び出しにあたる(図3(a))。

(b) 親プロセスは子プロセスの終了を待たない(no-wait)。子プロセスの返り値は子プロセス側で捨てられる(図3(b))。

(c) 親プロセスは子プロセスの終了を待たない。親プロセス側には **future**[2] と呼ばれるオブジェクトが返される。future は特殊なデータ型で子プロセスの計算が終了したときに、その計算された値に置き換えられてしまう。もし、子プロセスが終了する前に future を参照しようとすると、子プロセスが終了するまでその参照はブロックされる(図3(c))。

3.3 拡張手続きオブジェクトの生成

新しくプロセスを生成し並列に実行するとき、共有メモリ型並列計算機の場合には、新しく実行させたいプロセッサにその評価式の先頭のアドレスを与えるだけで式の評価を始めることができる。しかし、分散メモリ型並列計算機の場合には、個々のアドレス空間が独立であるので、新しくプロセスを生成する時は評価すべき式を自プロセッサのアドレス空間に転送する必要がある。

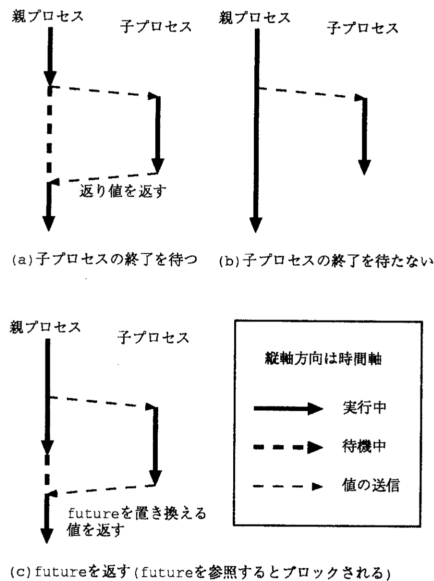

図 3: 親プロセスと子プロセスの同期

ネットワークトラフィック量の低減のため、評価すべき式をそのプロセスが実行されるプロセッサに送って格納しておき、それをプロセスの実行ごとに再利用する。このとき、評価すべき式を大域変数に格納すると、そのプロセッサで実行される他のプロセスからもその変数の値をアクセスすることが可能になる。ただし、処理系上のアプリケーションが大きくと、大域変数の名前の衝突が発生し易くなる。したがって、プロセスとして評価される式は実行されるプロセッサに存在しても、呼び出すもの以外からは隠蔽する必要がある。

拡張手続きオブジェクトは、プロセスで評価すべき式が全て送られてきて、引数が揃えば直ちに実行できる状態のものである。したがって、その拡張手続きオブジェクトを通さずにそのプロセスに関する操作を行えないようにした。つ

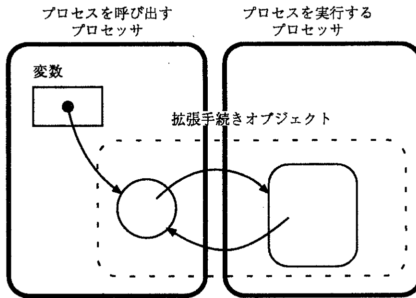

図 4: 拡張手続きオブジェクト

まり、拡張手続きオブジェクトを局所変数に束縛すれば、プロセスの存在を外からは完全に隠蔽することができる。

拡張手続きオブジェクトの生成には、先に述べた3通りの親プロセスと子プロセスの同期方式の各々に対応して、次の3つの手続きを用意した。

(1) (send-procedure-wait *processor proc*)

(2) (send-procedure-no-wait *processor proc*)

(3) (send-procedure-future *processor proc*)

processor はプロセスを実行するプロセッサ名を表し、シンボルで指定する。*proc* はプロセスとして評価される式であり、Scheme の手続きオブジェクトで指定する。この手続きでは *processor* で指定されるプロセッサに *proc* の複製を送り、プロセスの実行の準備をする。

上の3つの手続きの返り値は拡張手続きオブジェクトである。拡張手続きオブジェクトは図4のように、プロセスを呼び出す側とプロセスを実行する側のプロセッサにまたがって存在するオブジェクトである。

拡張手続きオブジェクトには次の情報が格納されている。

- プロセスを実行するプロセッサ名
- プロセスの同期方法 (wait,no-wait,future)
- プロセスを実行するプロセッサ側の拡張手続きオブジェクトへのポインタ

プロセスを実行するプロセッサ側に存在する拡張手続きオブジェクトは次の情報が格納されている。

- 親プロセスの存在しているプロセッサ名
- プロセスとして実行する手続き
- 返り値を返す場所へのポインタ

返り値を返す場所へのポインタは同期指定の方法が wait,future のときだけ有効な値をとる。

3.4 拡張手続きオブジェクトによるプロセスの実行

プロセスの実行は拡張手続きオブジェクト (*process-obj*) をリストの第一番目においてそのあとに引数 $(arg_1, arg_2, \cdots)$ が続くリストを評価することで行う。

$$(\textit{process-obj} \quad arg_1 \quad arg_2 \quad \cdots\ )$$

プロセス実行の手順は次のようになる。

step1 リストの第1番目が Scheme の予約語ではないので、リストの第1番目から最後まで評価する。

step2 リストの第1番目が拡張手続きオブジェクトであることが判り、step1 で評価した引数をプロセスを実行するプロセッサへ送る。これで、子プロセスの実行を始めることができる。

step3

(a) send-procedure-wait のとき
子プロセスが終了し、プロセスの返り値が返ってくるまで待ち、その値をプロセス実行の手続きの返り値とする。

(b) send-procedure-no-wait のとき

子プロセスの終了を待たず、プロセス実行の手続きは終了する。返り値は不定になる。

(c) send-procedure-future のとき
子プロセスの終了を待たず、プロセス

実行の手続きは終了する。子プロセスの返り値が後に入ることが約束されている future と呼ばれるオブジェクトが返り値として返される。

3.5 ブロードキャスト

ブロードキャストを行うためには、拡張手続きオブジェクトの生成時に、プロセスを実行するプロセッサ名にシンボル **all** を指定する。全てのプロセッサにプロセスの実行の準備をさせることができる。

send-procedure-wait で生成されたプロセスの返り値を待つ拡張手続きオブジェクトで呼び出した時、呼び出し側は全ての子プロセスが終了するのを待つ。それ以外の時は子プロセスの終了を待たない。そして、子プロセスの返り値も回収しない。

図 5に、all を使ってブロードキャストを行い全プロセッサで計算を行う例を示す。

```
(let ((foo
       (send-procedure-no-wait
        'all
        (let ((value 0))
          (lambda (message v)
            (case message
              ((add)
               (set! value
                     (+ value v)))
              ((set)
               (set! value v))))))))
  (foo 'set 4)
  (foo 'add 1))
```

図 5: ブロードキャストの例

プロセスは各プロセッサ上に独立に局所的な状態変数 value を持つ。`(foo 'set 4)` ですべてのプロセッサの状態変数 value に値 4 を設定し、`(foo 'add 1)` ですべてのプロセッサの状態変数 value に 1 を加算している。

4 分散メモリ型並列計算機での実装

4.1 共有変数

各プロセッサ上の Scheme の処理系での大域変数は、各々のプロセッサの中の環境で管理されている。そのため、全てのプロセッサで参照できる共有変数を定義するには、次のようにする。すなわち、変数に値を設定するごとに全てのプロセッサのトップレベルで define を実行するようなプロセス (以下に示す) を実行すればよい。

```
(define all-define
  (let ((def
          (make-process-wait
           'all
           (lambda (symbol value)
             (top-level-define
                symbol value)))))
    (make-process-wait
     'host
     (lambda (symbal value)
       (def symbol value)))))
```

しかし、このように all-define で逐一値を設定するのは面倒であるので、その変数に対して set! による代入が行なわれた時、自動的にその値が全てのプロセッサに反映するような機構を導入した。これを実現するために、define-common-variable という組み込み関数を処理系に追加する。

define-common-variable の構文は次のようになる。

(define-common-variable *variable expression*)

この構文を評価すると、*expression* の式がプロセッサの環境で評価され、以後、その値が *variable* と呼ばれる変数名で参照できるようになる。

4.2 通信

Scheme のオブジェクトがプロセッサ間で送受信されるのは、プロセスの実行開始時の引数

の引き渡しの時や、プロセス終了時に子プロセスが親プロセスに値を返すときである。

プロセッサ間でデータの共有は行わないため、オブジェクトの通信によって送信元のオブジェクトの実体の複製を受信先に作成することになる。例えば、リストを送信すると受信側に新しいリストが作られる。したがって、そのリストに対して set-car!や set-cdr!などの破壊的代入を行ってもリストを共有していないため、プロセッサ間をまたがって結果が反映されることはない。

また、同じプロセッサに向けて送信されたデータは、その送信した順序のとおりに受信されることを保証することにする。図 6のような例では、子プロセス内に存在する局所状態変数 value に、a,b,c の順で cons されていき、(foo 'cons 'c) で実行される子プロセスが終了した時点で、局所状態変数 value の値は (c b a) となることが保証される。

```
(let ((foo
       (send-procedure-no-wait
        'other-processor
        (let ((value ()))
          (lambda (message v)
            (case message
              ((get) value)
              ((cons)
               (set!
                 value
                 (cons v
                       value)))))))))
  (foo 'cons 'a)
  (foo 'cons 'b)
  (foo 'cons 'c)
  (write
   ((to-procedure-wait foo) 'get
                            ())))
```

図 6: 通信におけるオブジェクトの到着順序

4.3 相互排除

あるプロセッサでプロセスが実行されている間に、他のプロセッサからプロセスの実行開始のメッセージが到着した場合、そのメッセージはキューに入れられプロセスの実行開始の処理は後にまわされる。現在実行されているプロセスが終了するか、子プロセスの返り値の待ちなどの待機状態になった時に、キューに入ったメッセージが取り出され、新しいプロセスが実行される。すなわち、待機状態に入るような特殊な手続きを呼び出さない限り、プロセスの実行中は他のプロセスにコンテキストスイッチしないことになる。

拡張手続き内の局所状態変数は、プロセスの外からは隠蔽されて見えない。この変数にアクセスできるのは、その拡張手続きから呼び出されたプロセスだけである。このプロセスを実行している間、他のプロセスは実行できないので、局所状態変数に対する他のプロセスのアクセスを排除することができる。

4.4 同期

プロセスの実行中は他のプロセスは実行されることはないので、明示的に自プロセスの実行を一時中断し他のプロセスの実行を促す手続きを用意した。

(1) (get-current-process-id)
現在実行しているプロセスのプロセスを識別するためのオブジェクト *process-id* を返す。

(2) (stop-process *process-id*)
process-id で示されるプロセスを待機状態にする。

(3) (resume-process *process-id*)
stop-process で待機状態にした *process-id* で示されるプロセスの実行を再開する。実行を再開させるプロセスが同じプロセッサ上で実行されていたなら、実際にプロセスの実行を再開するのは現在実行しているプロセスが終了するか待機状態になってからである。

```
(define queue
  (send-procedure-wait
   'neighbor
   (let ((value-queue ())
         (read-proc-queue ()))
     (lambda (message . arg)
       (case message
         ((write)
          (set! value-queue (append value-queue (list (car arg))))
          (if (not (null? read-proc-queue))
              (begin
                (resume-process (car read-proc-queue))
                (set! read-proc-queue (cdr read-proc-queue)))))
         ((read)
          (if (null? value-queue)
              (let ((proc-id (get-current-process-id)))
                (set! read-proc-queue (append read-proc-queue (list proc-id)))
                (stop-process proc-id)))
          (let ((read-value (car value-queue)))
            (set! value-queue (cdr value-queue))
            read-value)))))))
```

図 7: キューのプログラム例

これらの手続きを用いて、同期を行い、キューを実現した例を図 7に示す。(queue 'write *value*) とするとキューに値 *value* を書き込み、(queue 'read) とするとキューから値を読み出すことができる。書き込みは常に成功し、キューに値が入る。読み出しはキューに値がなければ行うことができない。この例では、読み出しのときキューに値がなければ、キューに値が書き込まれるまで stop-process で待機する。stop-process の待機はスピンロックではなく、他に実行を待っているプロセスがあればそのプロセスが実行されるので、プロセッサの CPU 時間を無駄にしなくて済む。

4.5 ガーベージコレクション

プロセッサごとに実装した処理系のアドレス空間は独立しているので、逐次型の処理系と同じ手法で各処理系ごとに個別にガーベージコレクションを行うことができる。ただ、他のプロセッサからのみ参照される拡張手続きオブジェクトは、プロセッサ内に存在する拡張手続きオブジェクトのテーブルをみて、ガーベージコレクションのときゴミとして回収されないようにしなくてはならない。

5 おわりに

並列処理を手続きの単位で行う Scheme 処理系を作成した。その際、並列に実行可能な手続きを言語上で直接指定するようにし、なるべく容易に並列処理の記述ができることを目指した。

AP1000 への実装では、プロセッサごとにアドレス空間が独立しているので、処理系間を越えたオブジェクトの関係が複雑にならず、個別にガーベージコレクションを行うには便利である。しかし、プロセッサ間でオブジェクトの通信を行う場合、例えばシンボルを送信するとき、

プロセッサ間でアドレス空間が独立しているので、処理系内での複写のようにシンボルのポインタ値を送るだけではいけない。この場合、送信元でシンボルを文字列に変換し、受信先で文字列からシンボルを生成しなければならない。これは非常に無駄である。特に、記号処理においてシンボルは高速に処理したいので、プロセッサ間の通信量を減らす工夫が必要である。

また、実際に並列プログラムを実行させることによってシステムの評価を行う必要がある。

謝辞

本研究の一部は文部省科学研究費補助金・重点領域研究 (2)(04235103) の援助による。

参考文献

[1] W. Clinger, J. Rees (Editors), Revised^4 Report on the Algorithmic Language, LISP Pointers, Vol. IV, No. 3, July-Sep. (1991).

[2] R.M. Halstead Jr, Implementation of Multilisp: Lisp on a Multiprocessor, 1984 ACM Symp. on LISP and Functional Programming, pp.9-17 (1984).