AP1000 上の UtiLisp/C

和田英一[*] 田中哲朗[†]

概要

Implementation of UtiLisp/C, which is a newly coded version of UtiLisp for Sparc workstations, on AP1000 has been attempted. The goal is to reform UtiLisp for AP1000 with a very few C coded lines and a small number of special functions for distributed memory machines so that the basic feeling of parallel programming will be obtained in a relatively short time. No new fashionable functions often inplemented in parallel Lisps, e.g., future and so forth are not yet included; only a set of inter cell streams was added to execute top level eval loops in parallel. The set of the original UtiLisp functions proved well suited for use on MIMD machine.

1 概要

Lisp の Multiprocessor での実現はこれまでも行なわれてきたが, それらは大体が共有メモリーの計算機に対するものであった [1][2]. 最近, 処理装置の数が増大するにつれ, 共有メモリーから段々と分散メモリーが普通になりつつあるように思われる. AP1000 はそのような分散メモリーの multiprocessor であり, いろいろなプログラム言語処理系がその上に開発されている [3][4].

われわれは, たまたま AP1000 の cell で使っている処理系の sparc で動く lisp の interpreter[5] で C 言語で記述してある (従って UtiLisp/C という) のを所有していたので, AP1000 のような分散メモリーの MIMD 型計算機に lisp をなるべく簡単な手法で載せてみて, UtiLisp の分散環境との親和性, また分散環境での lisp のプログラミングの標準手法を用意したいと思い, 移植作業を始めた.

大体の方針は UtiLisp はすべておなじで, cell で動かす. はじめは host にも UtiLisp をのせる案もあったが, host は大勢で使っているため, load が大きかったりすると, 実行時間の測定が不正確になることもありそうだとか, その他の理由でやめにした. host の仕事は結局

1. object file を cell に load する.
2. lispsys.l を file system から読み込み, 各 cell に broad する.
3. user との I/O を担当する.
4. file system の file の読み書きをする.

程度のことしかしない. cell の方は host 及び他の cell から S 式を読み込み, それを評価して結果の値を, S 式を送ってきた host か cell に返す, いわゆる eval server として働く. host–cell 間, cell–cell 間は AP1000 の通信ライブラリに lisp の皮をかぶせた入出力関数 `read`, `print` で行なう.
各 cell の記憶装置が独立しているため, garbage collection については, なにも考える必要はないのが分散メモリーのメリットである. 各 cell で heap の free cell が不足してくると, 勝手に gc が走るだけである.

[*]東京大学名誉教授, e-mail: wada@u-tokyo.ac.jp
[†]東京大学, e-mail: tanaka@ipl.t.u-tokyo.ac.jp

2 実装

主として変更したのは read/write (lisp の関数としては read/print) である．分散メモリーの cell は，お互いおよび，host との間は cell 間通信 (T-net または B-net) によらなければならない．UtiLisp には stream を引数にとる read/print と引数のない (stream を standard-input, standard-output の変数に bind されているものとし，通常は terminal-input, terminal-output が bind してある.)

stream は通常は filename に対して，(`stream` *filename*) でその filename をもつ stream を作り，`inopen`, `outopen` したときに file descriptor を割り当てる．UtiLisp/C にはこの他，fixnum を引数とする stream も呼べて，そのときはその fixnum が file descriptor になる．

unix の file descriptor は，0 が read，1 が write，2 が error write に割り当ててある． このほか AP1000 の os ではその他の file には 63 番までの file descriptor を割り当てるみたいなので，cell 間通信には 256 から 1279 までの file descriptor を使い，file descriptor - 256 を通信相手の cell 番号に対応させた． つまり，file 256 は cell0 との通信に，file 257 は cell1 との通信に，file 1279 は cell1023 との通信に使う．

その他 file 4096 を host に割り当て，また file 4095 は任意の cell からの read と，一斉放送の write の対応させた．

cell 0 の top loop は host からの入力を待ち，それ以外の各 cell の top loop は任意の cell または host からの入力を待っている．通常の top loop は S 式を読み込むとそれを評価し，入力した stream に結果を返す．この他一斉放送による S 式を読み込んだ時は評価はするが，結果は返さない．この時は結果の値より副作用の効果の方を期待しているのである．

3 実例

簡単な並列処理の例を fibonacci 数の関数で説明する．cell0 が (`fib` n) を計算しようとしているとする．

- それには (`fib` $n-$ `1`) と (`fib` $n-$ `2`) が必要になるが，
- (`fib` $n-$ `1`) は自分で計算し，(`fib` $n-$ `2`) の計算は cell 1 に依頼する．
- cell 0 は (`fib` $n-$ `1`) を計算するのに (`fib` $n-$ `2`) は自分で計算し，(`fib` $n-$ `3`) の計算は cell 2 に依頼する．
- 一方 cell 1 は (`fib` $n-$ `2`) の計算をするのに (`fib` $n-$ `3`) は自分で計算し，(`fib` $n-$ `4`) の計算は cell 3 に依頼する．
- つまり依頼する相手と自分との差は，1 段目は 1，2 段目は 2，3 段目は 4 . . . のように増えていく，

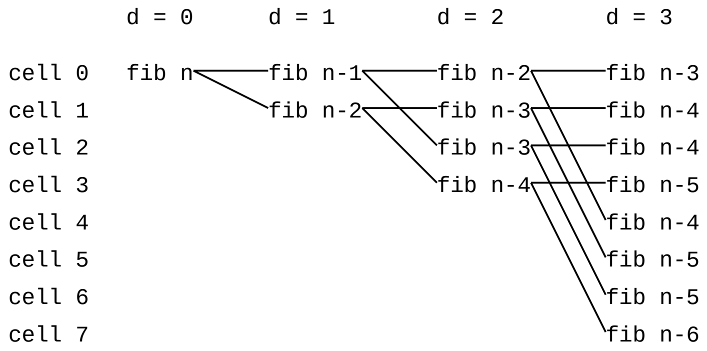

Figure 1

全体の cell 数を c とすれば, $\log_2 c$ 段まで進めることができる (Figure 1).

段数の制御のため, fib にもう一つの引数 d をつける.

```
(defun fib (n (d 0))
   (cond ((lessp n 2) 1)
         ((lessp d (getlcel))
          (lets ((mycid (getcid)) (child (plus mycid (expt 2 d))) (myret)
                 (childret) (cstream0 (outopen (stream (plus child 256))))
                 (cstream1 (inopen (stream (plus child 256)))))
           (print (list 'fib (/- n 2) (/1+ d)) cstream0)
           (setq myret (fib (/1- n) (/1+ d)))
           (setq childret (read cstream1))
           (plus myret childret)))
         (t (plus (fib (/1- n) (/1+ d)) (fib (/- n 2) (/1+ d))))))
```

fib の第 2 引数 `(d 0)` は default の値が 0 であることを示す.
`(getlcel)` は cell の全数の 2 を底とした対数を返す.
`(getcid)` は自分の cell 番号を返す.
`child` は下請けの cell 番号. `cstream0`, `cstream1` は下請け cell に対する入出力の stream である.
`(print (list 'fib (/- n 2) (/1+ d)) cstream0)` で下請けに `(fib` n $-$ `2)` の計算を依頼し,
`(read cstream1)` で下請けの計算結果を読み込む.

段数がだんだん増え, `(getlcel)` に等しいかそれ以上になると, 計算は自分で行なう.

quicksort もほぼ同様に書くことができる.

```
(defun qsort (l d)
   (cond ((not (greaterp (length l) 1)) l)
         ((lessp d (getlcel))
          (lets ((mycid (peek)) (myret) (childret)
                 (child (plus mycid (expt 2 d)))
                 (cstream0 (outopen (stream (plus child 256))))
                 (cstream1 (inopen (stream (plus child 256)))))
           (setq l (part l))
           (print (list 'qsort (list 'quote (caddr l)) (/1+ d)) cstream1)
           (setq myret (qsort (car l) (/1+ d)))
           (setq childret (read cstream0))
           (append myret (cadr l) childret)))
         (t (setq l (part l))
            (append (qsort (car l) (/1+ d))
                    (cadr l)
                    (qsort (caddr l) (/1+ d))))))
(defun part (l)
   (lets ((sm nil) (gr nil) (cr (car l)))
    (setq l (cdr l))
    (loop (cond ((null l) (exit)))
          (setq hd (car l) l (cdr l))
          (cond ((> hd cr) (setq gr (cons hd gr)))
```

```
                (t (setq sm (cons hd sm)))))
    (list sm (cons cr nil) gr)))
```

quicksort のテストに使う乱数の数列を作り, 例えば変数 `foo` に結合すれば, `(qsort foo 0)` のようにして quicksort を実行する.

4 計測

utilisp には評価の時間をはかる関数 (`time S` 式) があった. それと同様に AP1000 の utilisp にも time 関数を用意した. AP1000 用の time には default 値が 1000 の第 2 引数 `scale` があり, 1/`scale` 秒を単位として計時した結果を返す.

したがって, cell 台数を 64, 32, 16, 8, 4, 2, 1 とした時の `(fib 20)` の計算時間を得るには:

```
(defun foo ()
  (lets ((i 0))
     (loop (prin1 (expt 2 (- (getlcel) i)))
        (print (time '(fib 20 i)) 60) (cond ((= i 6) (exit)))
        (setq i (1+ i)))))
```

と定義し, `(foo)` を実行すればよい.
計測の実行結果は次の通り.

```
% host -n 64
[0]cell_main=0x62028
[0]> (exfile 'fib.l t)
[0]fib
[0]nil
[0]> (exfile 'time.l t)
[0]foo
[0]nil
[0]> (foo)
[0]64[0]27
[0]32[0]42
[0]16[0]67
[0]8[0]107
[0]4[0]173
[0]2[0]279
[0]1[0]450
[0]nil
[0]> (quit)
```

これは file システムから fib.l, time.l の file を読み込み (`exfile` による), 前述の `(foo)` で実行時間を計測したときの script である. `[0]` の表示はその次の出力が cell 0 のものであることを示す.

先頭の行の `host -n 64` は, cell 64 台で host を起動する. `(exfile 'fib.l t)` は, `fib.l` を読み込む. 同様に `time.l` も読み込む. `(foo)` の行の下が, cell の台数と 1/60 秒単位の実行時間である.

cell 台数対実行時間を Figure 2 に示す.

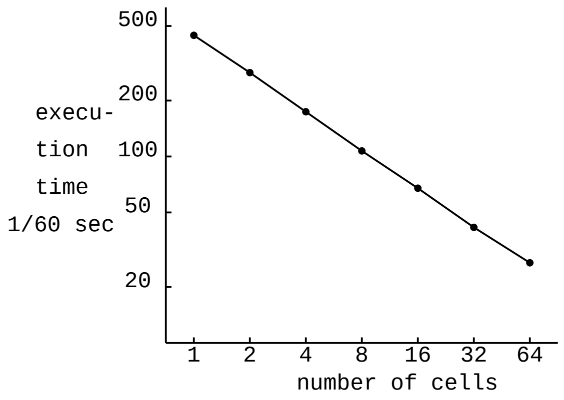

Figure 2

5 割込

ある cell が一心に計算中に, ほかの cell から S 式が送り付けられることがあるのは充分予想される. 一般的にはいま計算中の toplevel loop の eval の実行が済んでから, 読み込めば良いが, 中には別の cell からの緊急な要求が来ているかもしれない. たとえば, MUtiLIsp[1] では, 変数の bind している値は親のプロセスから継承するが, とりあえずは unbound の状態であり, 変数を参照にいって, unbound だとその時点で親のプロセスに聞きに行く. AP1000 の utilisp にも, このような機構をつけようとるると, 親の cell は自分の計算中に子の cell からの変数の値読みだし要求がきたら, 即刻対処しなければならず, そためには, 割り込みの機能が必要になる.

ところが, AP1000 の cell には割り込みの機能はついていない. したがって, 当面は eval の処理のたびに cprob によって, 受信待ち状態かどうかを知り, 受信まちの情報が溜っていれば, eval の実行の前に `break` を `funcall` することにした.

利用者は受信待ちの情報の処理の仕方を break の関数に定義しておけばよい. 現在 lispsys.l に組み込んだ break 関数は, 受信待ち情報を読み込み, 一応 stream の名前の待ち行列につなげるようになっている. これに対応して, `read` 関数もまず待ち行列を見に行き, 待っている情報があるとそれを行列からはずして, `read` の結果としてかえすようになっている. そのほか緊急を要する仕事は, `break` に追加すれば良い.

6 同期

並列計算を正しく実行するにはプロセスの進行を制御しなければならない時がある. 代表的な制御は同期である. 前述のように今回の移植は interpreter になるべく手を加えないという方針であったため, 最初に動いたシステムには同期を制御する関数がなかった. 同期制御が必要な時は cell 同士で情報を交換しながらやったのだが, これでは一斉射撃の問題を毎回解いているようなもので, 能率が悪く, Lisp 環境から AP1000 の cell ライブラリを呼べるようにした.

(`sync` *no stat*), (`cstat`), (`pstat` *stat*), (`gstat`) を用意し, 対応する同期用の手続きが使えるようになった.

7 作業

作業量については, 最初に述べたように UtiLisp/C のソースプログラムにはなるべく手を加えない方針だったので, C 言語によるプログラムの行数は次の程度に収まっている.

ファイル名	作業行数
`host.c` (hostmain)	134 行 (新設)
`machdep.c` (cell プログラム 入出力)	259 行 (新設)
`main.c` (cellmain)	39 行追加
`eval.c` (割り込み処理)	6 行追加
`sysfnmis.c` (雑処理関数群)	94 行追加

lisp で書いた部分は UtiLisp 起動時に読み込まれる lispsys.l に集約してある. lispsys.l の長さは 今のところ 247 行だが, これはどんどん長くなっている. ちなみに通常の UtiLisp の lispsys.l は 630 行程ある.

8 結論

この UtiLisp/C の AP1000 への移植では, C 言語のプログラムはなるべくしないというのを主目的にした. Lisp で書けるものは出来るだけ Lisp で書き, lispsys.l のファイルに入れて使う. その方が対話的に実験が出来るし, もともと UtiLisp のシステムも記述できる能力の証明にもなる. 実際デバッグや性能モニターが容易であった.

ここにはわずかなプログラム例しか示さなかったが, 別の例もやってみている. たとえば, 割り込みや一斉放送の機能を確かめるのに分散 Eratosthens のふるいも書いて見たりした. その結果, ある cell の計算中に別の cell が除算の依頼を出すのが期待通りに実行されるのが確認できた. もっともふるいとしては少しも早くは走らなず, まだまだ工夫の余地があることも判明した.

わずかながら試みたプログラムは, 台数効果をとりだすのがなみ大抵ではないことを示している. それに並列アルゴリズムにすると逐次実行の時にはあたりまえであった制御の流れの順に同期を明示しなければならない事態があることもわかってきた.

プログラムの処理時間解析には, 各部の詳細な時間データが必要であり, そのためには `time` 関数を修正する必要もある.

最後にシステムのサイズについて触れておく. interperter の binary text は 225KB. 作業領域のうち最大なのは heap で, default では 512KB がとられる. AP1000 の各 cell には 16MB のメモリーがあるので, まだかなり大きなプログラムを走らせることが可能である.

参考文献

[1] Halstead,R.H.: Multilisp: A Language for Concurrent Symbolic Computation, ACM Trans. Program. Lang. Syst., Vol.7, No.4 501-538(1985)

[2] 岩崎英哉: Lisp における並列動作の記述と実現 情報処理学会論文誌 Vol.28, No.5,465-470(1987)

[3] Baily,B., Newey,M.: Implementing ML on Distributed Memory Multiprocessors, Sigplan Notices, Vol.28, No.1,56-59(1993)

[4] Taura,K., Matsuoka,S., Yonezawa,A.: An Efficient Implementation of Concurrent Object-Oriented Languages on Stock Multiprocessors, Sigplan Notices, Vol.28, No.7 218-228(1993)

[5] 田中哲朗: SPARC の特徴を生かした UtiLisp/C の実現法 情報処理学会論文誌 Vol.32,No.5,684-690(1991)