UtiLisp for the AP1000

User's Guide
as of February 1995

1 The Directory

The files of UtiLisp/AP1000 reside on the directory `/pcrf/pub/regsfw/UtiLisp` which is hereafter represented by {REGD}.
Under this base directory, five subdirectories exit, i.e.:

{REGD}`/src/` includes source programs and a makefile.

{REGD}`/bin/` includes binary programs for the cell machine and `apilisp` for the host machine.

{REGD}`/lib/` contains `lispsys.l`, `apsys.l` and the directory `lisplib`.

{REGD}`/sample/` contains sample programs.

{REGD}`/doc/` holds documents: `guide.ps`(this document) and `utilisp.texi`

2 The Main Program

The main program is `apilisp` running on the host machine of the AP1000.

```
% apilisp
```

The options are:

`-N` *ncel* *ncel* specifies the number of cells to be organized in the cell array. Default is the total cells of the system.

`-x` *xcel* *xcel* specifies the number of cells to be organized in the two dimensional cell array.

`-y` *ycel* *ycel* specifies the number of cells to be organized in the two dimensional cell array.

Other parameters are the same as the original UtiLisp [1].

`-h` *size* This specifies that the heap area is to be *size* kilo bytes. The default heap size is 512 kilo bytes.

`-S` *size* This specifies that the symbol area is to be *size* kilo bytes. The default symbol size is 60 kilo bytes.

`-f` *size* This specifies that the size of the fixed heap area for compiled codes is to be *size* kilo bytes. The default fixed heap size is 64 kilo bytes. Compiler needs more than 300 kilo bytes of fixed heap area.

`-n` This specifies that UtiLisp should not read and evaluate the `$HOME/.utilisprc` file on starting up.

`-F` *filename* This specifies that UtiLisp should read and evaluate *filename* on starting up.

When `apilisp` is invoked on the host machine, the host machine broadcasts the utilisp interpreter to all the cells. Then it broadcasts the start up files "`lispsys.l`" and "`apsys.l`" consisting of a bunch of lisp forms which are read and evaluated one by one by the interpreters on the cells.

3 Read-Eval-Print Loop

At the end of reading "`apsys.l`", cell 0 enters into the nearly standard read-eval-print loop and waits for input from the host, i.e. terminal-input with the dafault prompt character "`>`". (To alart the user, the first prompt sign is preceded by a single beep.) Other cells wait for input sent from any cells or host.
The read-eval-print loop is somehow modified as follows:

1. Read function reads forms from standard-input, and remembers the sender. The evaluated value will be sent back to the original sender.

2. If the form read in is of the form "`(broad-massage form)`, i.e., the form is sent by broadcasting, the form is evaluated, but the value is not sent back; this is for obtaining the side effect.

3. At the entrance to eval function, the interpreter checks the existence of the incoming message and in case one exists, the function bound to `break` is funcalled. Inside the break function, the symbol `my-break` is examined if it is bound. If so, `my-break` is funcalled with the input data as an argument. You may simply setq as
   `(setq my-break (function (lambda (inputdata) ...)))`.

4 Numbered Streams

Stream is enhanced to include the integral numbered streams. Numbered streams correspond to the cells or the host. Streams form 256 to 1279 inclusive correspond to the cell 0 to cell 1023 respectively. Stream 4096 is for the host machine. Stream 4095 is used to indicate "any cell" or host for read and "broadcast send" for print. (see Figure 1)

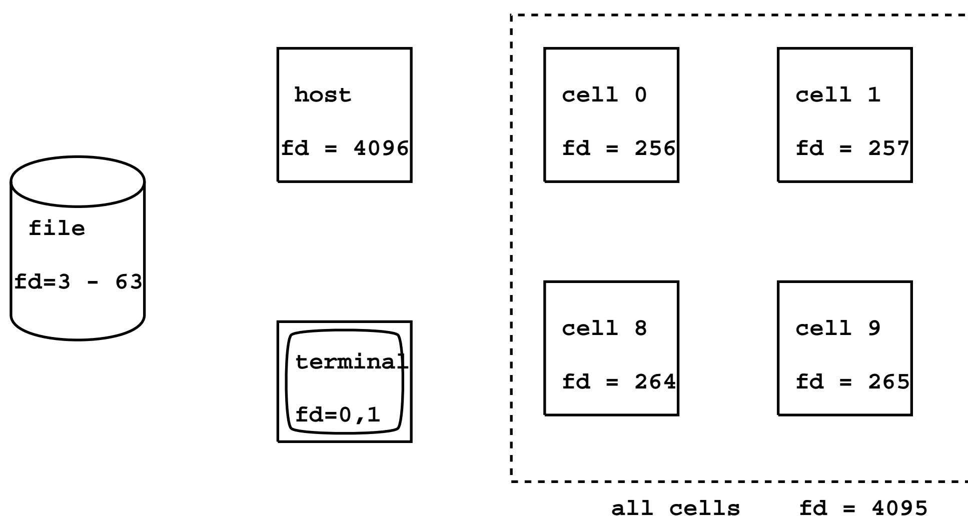

Figure 1

5 Additional Functions and System Variables

The following functions are defined for use with the AP1000 and included in the start up file "`apsys.l`".

`quit` (the original `quit` is overloaded)

`quit` stops all the cells and returns control to Unix running on the host.

`time` (*form*) (*scale*)

Mostly the same as the standard **`time`**. However, if *sacle* parameter is present, time for executing *form* is measured in the unit of 1/*scale* second. The default value of *scale* is 1000, i.e. measured in miliseconds.

`getcid`
`getcidx`
`getcidy`

`getcid`, `getcidx` and `getcidy` are the UtiLisp version of the corresponding functions `getcid`, `getcidx` and `getcidy` of the Cell Library.

`getncel`
`getncelx`
`getncely`

`getncel`, `getncelx` and `getncely` are the UtiLisp version of the corresponding functions `getncel`, `getncelx` and `getncely` of the Cell Library [2].

`getlcel`

`getlcel` returns $log_2 n$ where n is the number of cells in the system.

`sync` *no stat*
`cstat`
`pstat` *stat*
`gstat`

`sync`, `cstat`, `pstat` and `gstat` are the UtiLisp version of the corresponding funcitons in the Cell Library. For the values returned from each of the functions, refer to the Cell Library.

`debug` x

`debug` prints x with the cell number on the hostmachine. `debug` returns x.

`broadmessage` x

`broadmessage` broadcasts the list `(broad-message` x`)` to all the cells other than the own cell. `broadmessage` is exceptional in that the read eval print loop does not return the value.

For example, `quit` is defined as:

```
(putd 'ur-quit (getd 'quit))        ;the original quit is restored as ur-quit
(defun quit ()
  (broadmessage '(ur-quit))
  (ur-quit))
```

`cset` *cell var val* {Macro}

`cref` *cell var* {Macro}

Function `cset` is used to `setq` the variable *var* in the cell *cell* to the value *val*. Function `cref` reads the variable *var* in the cell *cell*.

`collectall` *op info* {Macro}

`collectall` gathers all the data named *info* in the cells applying the binary operation *op* and puts the resultant value in the *info* of the cell0.
For example:

```
(defun catest ()
  (broadmessage '(setq foo (ncons (getcid))))
  (setq foo (getcid))
  (broadmessage '(collectall append foo))
  (collectall append foo))
```

`foo` in the cell 0 is replaced by a list of all the cell numbers in the system.

`lambda` {Variable}

Variable `lambda` is used as an interrupt permission flag. When its value is 1, interruption is permitted.

`version` {Variable}

`version` remembers the stream number of the `standard-input` from which `read` function read data. `(print form)`, i.e., print out to the standard-output will send the print string to the stream which is the current value of the variable `version`.

6 Interruption

While a cell is busy by evaluating a form, it becomes sometimes necessary to interrupt to the cell to inform events. Since the AP1000 operating system does not support signalling mechnism, at the entrance of the eval function, the input buffer is examined. If there is a message and interruption is permitted, after resetting the interrupt permission, the break program is funcalled. The break program bound by the start-up program `apsys.l`", reads in the message and if symbol `my-break` is bound, it is funcalled with single parameter that is the input message.
So, as sees in the example of 8 queens, symbol `my-break` should be written as:

```
(setq my-break '(lambda (inputdata)
 (cond ((and (consp inputdata) (eq (car inputdata) 'tag))
        (process the inputdata)))
 (setq lambda 1)))
```

Here, the symbol `lambda` is used as an interrupt permission flag, one meaning permissible and zero prohibited.

7 Program Examples

Four examples are explained in this section. The corresponding program files are found under the directory {REGD}`/sample/`.

Fibonacci number	{REGD}`/sample/fib.l`
Execution time	`time.l`
Synchonization	`synctest.l`
Eight queens	`nqueens.l`
Load balance	`load.l`
Competing server Array	`strongprime.l`

7.1 Fibonacci number

The simplest example is a function to calculate the nth Fibonacci number.
Try to think that cell 0 needs `(fib` n`)`.

- Then `(fib` n $-$ `1)` and `(fib` n $-$ `2)` are needed.
- Cell 0 calculates `(fib` n $-$ `1)` itself and asks cell 1 to calculate `(fib` n $-$ `2)`.
- When cell 0 calculates `(fib` n $-$ `1)` itself, it calculates `(fib` n $-$ `2)` itself and asks cell 2 to calculate `(fib` n $-$ `3)`.
- On the other hand, when cell 1 calculates `(fib` n $-$ `2)` itself, it calculates `(fib` n $-$ `3)` by itself and asks cell 3 to calculate `(fib` n $-$ `4)`.
- The difference of cell numbers, asking and being asked, is 1 in the first step, 2 in the second step, 4 in the third step ....

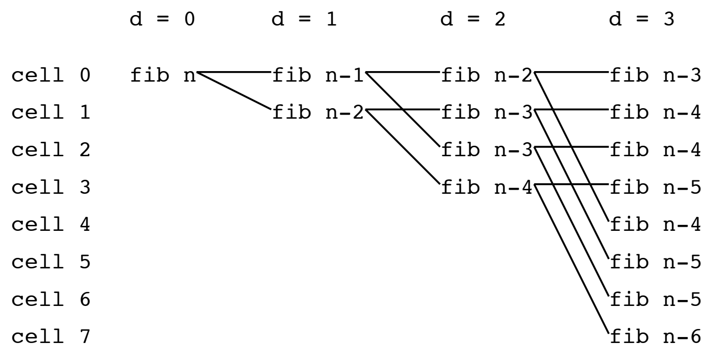

Figure 2

Lets c be the number of total cells, the distributed calculation may be performed until $\log_2 c$ steps. (see Figure 2)
Fibonacci program requires additional argument, `d`, to control steps.

```
(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))))))
```

The second argument `(d 0)` tells that unless the second argument is explicitly given, `d` will be bound to 0. Function call `(getlcel)` returns $\log_2 c$ where c is a total number of cells. Function call `(getcid)` returns the ordinal number of the current cell. Variable `child` is the ordinal number of the subprocessor cell. `cstream0` and `cstream1` are respectively input/output streams to communicate with child cell. By issuing `(print (list 'fib (/- n 2) (/1+ d)) cstream0)`, current cell asks the child cell calculation of `(fib` $n-$ `2)` and with `(read cstream1)`, it gets back the evaluated value.

If steps come to the upper limit, i.e `(getlcel)`, each cell proceeds calculation by itself.

Function `(time` *form*`)` of UtiLisp reports the time consumed in evaluation of the *form*. The equivalent function `(time` *form scale*`)` is prepared to measure the execution time of *form* with $1/scale$ as a unit. Here, the second argument *scale* is a default having the value 1000.

Changing the number of cells from 64 to 32, 16, 8, ..., 1, `(fib 20)` execution time in 1/60 sec was measured by provoking `(foo)` after defining function `foo` by:

```
(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)))))
```

The following script was obtained:

```
% apilisp -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)
```

The first line of the script, `apilisp -N 64` called command `apilisp` and 64 cell have been loaded with UtiLisp interpreter.

`[0]` in the above output indicates the print out following the brackets came from cell 0. `(exfile 'fib.l t)` and `(exfile 'time.l t)` read in the corresponding files. `fib` and `foo` next to `(exfile ...)` are

output of (defun ...). The last seven columns show a pair of number of cells (e.g. 64) and execution time (e.g. 27) in 1/60 second. Figure 3 indicates the number of cells vs execution time curve.

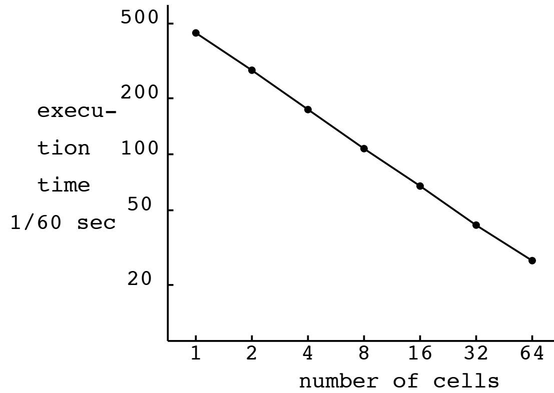

Figure 3

7.2 Synchronization

In the following program, four cells, 0 to 3, are to execute loop for its cid (cell number) times, i.e. 0 times for cell 0, 1 times for cell 1, etc. Until all the cells exited the loop, the already exited cells wait in the second loop. At the end of the program, all the cells print "end" with debug function. (see Figure 4)

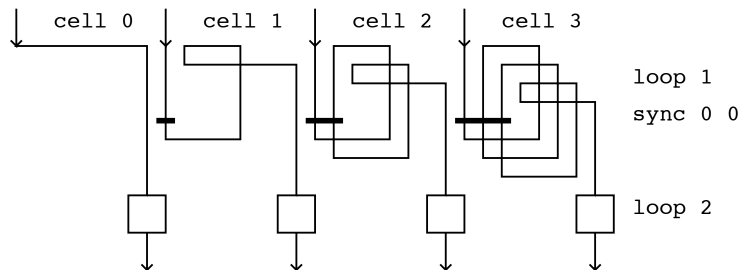

Figure 4

```
(defun synctest ()
 (pstat 1)                  ;set state 1
 (sync 0 0)                 ;synchronization to start all simultaneously
 (setq count (getcid))      ;initialize counter
 (loop (cond ((zerop count) (pstat 0) (exit)))   ;when counter becomes 0,
       (debug count)                             ;set state 0 and exit loop
       (setq count (sub1 count))                 ;decrease counter
       (sync 0 0))                               ;loops procede synchronously
 (loop (debug (list 'c (cstat)))          ;check logical or of state
       (cond ((zerop (cstat)) (exit)))    ;wait till all cell set state 0
       (sync 0 0))
 (debug (list 'end (cstat)))              ;print ''end'' with or-ed state
 (sync 0 0))

(cond ((0= (getcid))        ;driver program for cell 0
```

```
    (broadmessage '(synctest))
    (synctest)))
```

The script is shown below; the host program was invoked with a parameter `-n 4`.

```
[0]cell_main=0x64028
[0]> (exfile 'synctest.l t)
[0]synctest                      ;value of (defun synctest ()...
[0](c 16)                        ;cell 0 entered into the 2nd loop (1st)
[1]1                             ;cell 1 in the 1st loop (1st)
[2]2                             ;cell 2 in the 1st loop (1st)
[3]3                             ;cell 3 in the 1st loop (1st)
[2]1            ;sync 1          ;cell 2 in the 1st loop (2nd)
[0](c 16)                        ;cell 0 in the 2nd loop (2nd)
[1](c 16)                        ;cell 1 entered into the 2nd loop (1st)
[3]2                             ;cell 3 in the 1st loop (2nd)
[0](c 16)       ;sync 2          ;cell 0 in the 2nd loop (3rd)
[1](c 16)                        ;cell 1 in the 2nd loop (2nd)
[2](c 16)                        ;cell 2 entered into the 2nd loop (1st)
[3]1                             ;cell 3 in the 1st loop (3rd)
[0](c 16)       ;sync 3          ;cell 0 in the 2nd loop (4th)
[1](c 16)                        ;cell 1 in the 2nd loop (3rd)
[2](c 16)                        ;cell 2 in the 2nd loop (2nd)
[3](c 0)                         ;cell 3 entered into the 2nd loop (1st)
[0](end 0)                       ;cell 0 exited the 2nd loop
[1](end 0)                       ;cell 1 exited the 2nd loop
[2](end 0)                       ;cell 2 exited the 2nd loop
[3](end 0)                       ;cell 3 exited the 2nd loop
[0]0                             ;value of (sync 0 0)
[0]nil
[0]> (quit)
```

7.3 Eight queens

The eight queens program for the single processor is shown first[3]. Three bit arrays, `col`, `up` and `dn` keep the information about the free columns, free diagonals. (see Figure 5)

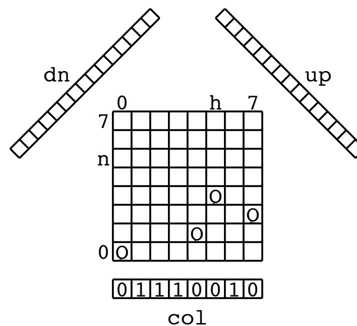

Figure 5

```
(defun generate (n x col up dn)
 (lets ((h))
  (cond ((= n 8) (print x))
        (t (do ((h 0 (/1+ h)))
               ((= h 8))
               (cond ((and (bref col h)
                           (bref up (/+ n (/- h) 7))
                           (bref dn (/+ n h)))
                            (generate
                            (/1+ n)
                            (cons h x)
                            (place col h)
                            (place up (/+ n (/- h) 7))
                            (place dn (/+ n h))))))))))

(defun place (str pos)
 (lets ((str1 (string-append str))) (bset str1 pos nil) str1))

(generate 0 nil (make-string 1 255) (make-string 2 255) (make-string 2 255))
```

The strategy for multiple CPU version is as follows:
The program in cell 0 keeps two lists, cell and work. In the beginning, cell has a list of all free cell machines, i.e. a list from 1 to 15 (in the case of 16 cells), and work is nil.
Cell 0 program dispatches a work to cell 1. Cell 1 generates new works and sends back to cell0. The message is the form of:

```
(nqueens newwork generate 1 ...)
```

Each message is received by `break` by setting a function in `my-break`. The break function for the AP1000 first reads in the incoming message in `inputdata` and then see whether a symbol `my-break` is bound or not. If it is bound, break function funcalls `my-break`. Accordingly, my break may be programmed as one in the example.
The main part of the cell 0 program is a loop where works in the list of work is sent to any free cell; the loop is exited if it finds that a work list is null and the cell list is full.
`my-break` sorts the type of the incoming messages by the second tag. (The first tag is always "`nqueens`". If the tag is `newwork`, the rest is consed to work list. If the tag is `free` it means the cell come to the end of the work, so it is registered to the cell list. If the tag is `result` the result value will be consed to

answer list. (see Figure 6)

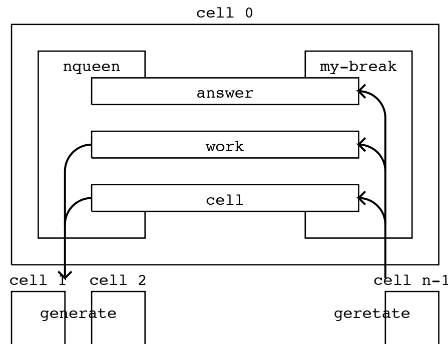

Figure 6

```
(setq qqq 8)                       ;i.e. 8 queens

(defun nqueen ()                   ;program running on cell 0
 (lets ((ncel (getncel)) (cell (genlist 1 ncel)) (work) (nextcell) (nextwork)
        (answer)
        (my-break                  ;my-break is bound only inside of nqueen
           '(lambda (inputdata)
             (cond ((and (consp inputdata) (eq (car inputdata) 'nqueens))
                    (selectq (cadr inputdata)
                       (newwork (setq work (cons (cddr inputdata) work)))
                       (result (setq answer (cons (caddr inputdata) answer))
                               (setq cell (nconc cell
                                            (list (cadddr inputdata)))))
                       (free (setq cell (nconc cell
                                          (list (caddr inputdata)))))))
                   (t nil))
             (setq lambda 1))))
  (setq nextcell (car cell) cell (cdr cell))
  (sendto nextcell
          (list 'generate
                0
                nil
                (make-string 1 255)
                (make-string 2 255)
                (make-string 2 255)))
  (loop (cond ((and (null work) (= (length cell) (/1- ncel))) (exit answer)))
        (lets ((lambda 0))
         (cond ((and work cell)
                (setq nextwork (car work)
                      work (cdr work)
                      nextcell (car cell)
```

```
                    cell (cdr cell))
               (sendto nextcell nextwork)))))))

(defun genlist (x y)           ;generate a list (x x+1 ... y-1)
 (cond ((= x y) nil) (t (cons x (genlist (1+ x) y)))))

(defun sendto (cell-to-send work-to-send)
 (lets ((s (outopen (stream (/+ cell-to-send 256)))))
  (print work-to-send s)
  (close s)))

(defun generate (n x col up dn) ;program running on cell n (n>0)
 (lets ((h) (mesg))
  (cond ((= n qqq) (list 'nqueens 'result x (getcid)))
        (t (do ((h 0 (/1+ h)))
               ((= h qqq) (list 'nqueens 'free (getcid)))
               (cond ((and (bref col h)
                           (bref up (/+ n (/- h) 7))
                           (bref dn (/+ n h)))
                      (setq mesg (list 'nqueens
                                       'newwork
                                       'generate
                                       (/1+ n)
                                       (list 'quote (cons h x))
                                       (place col h)
                                       (place up (/+ n (/- h) 7))
                                       (place dn (/+ n h))))
                      (sendto 0 mesg))))))))

(defun place (str pos)
 (lets ((str1 (string-append str))) (bset str1 pos nil) str1))

(and (= (getcid) 0) (nqueen))   ;start nqueen on cell 0
```

In the parallel processors, load palancing is of great concern. In the case of the eight queens program shown above, the work load of cell 0 is too heavy because it is constantly managing the jobs for other cells.

To see the rough load balancing, function `time` may be used. On entering to and on exiting from the main program, two functions `start` and `stop` are inserted which update the value of the `running-time`. The function `collect-time` collects all the values of `running-time`. In the following program, only the related parts are shown. (Update markers on the leftside of line indicate the modified lines.)

```
|(setq running-time 0)
|(defun start () (setq running-time (- running-time (time))))
|(defun stop () (setq running-time (+ running-time (time))))
|(defun collect-time ()
| (lets ((ncel (getncel)))
```

```
|  (print (list 0 running-time))
|  (do ((i 1 (/1+ i)))
|      ((= i ncel))
|      (lets ((s0 (outopen (stream (/+ i 256))))
|             (s1 (inopen (stream (/+ i 256)))))
|       (print (list 'sendback-time) s0)
|       (close s0)
|       (print (list i (read s1)))
|       (close s1)))))
|(defun sendback-time () running-time)

|(defun nqueen () (start)
  (lets ((ncel (getncel)) (cell (genlist 1 ncel)) (work) (nextcell) (nextwork)
         (answer)
         (my-break
            '(lambda (inputdata)
              (cond ((and (consp inputdata) (eq (car inputdata) 'nqueens))
                     (selectq (cadr inputdata)
                        (newwork (setq work (cons (cddr inputdata) work)))
                        (result (setq answer (cons (caddr inputdata) answer))
                                (setq cell (nconc cell
                                                  (list (cadddr inputdata)))))
                        (free (setq cell (nconc cell
                                                (list (caddr inputdata)))))))
                    (t nil))
              (setq lambda 1))))
   (setq nextcell (car cell) cell (cdr cell))
   (sendto nextcell
           (list 'generate
                 0
                 nil
                 (make-string 1 255)
                 (make-string 2 255)
                 (make-string 2 255)))
   (loop (cond ((and (null work) (= (length cell) (/1- ncel)))
|               (stop) (exit answer)))
         (lets ((lambda 0))
          (cond ((and work cell)
                 (setq nextwork (car work)
                       work (cdr work)
                       nextcell (car cell)
                       cell (cdr cell))
                 (sendto nextcell nextwork)))))))

|(defun generate (n x col up dn) (start)
  (lets ((h) (mesg))
```

```
|  (cond ((= n qqq) (prog1 (list 'nqueens 'result x (getcid)) (stop)))
        (t (do ((h 0 (/1+ h)))
|               ((= h qqq) (prog1 (list 'nqueens 'free (getcid)) (stop)))
               (cond ((and (bref col h)
                           (bref up (/+ n (/- h) 7))
                           (bref dn (/+ n h)))
                      (setq mesg (list 'nqueens
                                       'newwork
                                       'generate
                                       (/1+ n)
                                       (list 'quote (cons h x))
                                       (place col h)
                                       (place up (/+ n (/- h) 7))
                                       (place dn (/+ n h))))
                      (sendto 0 mesg)))))))

|(and (= (getcid) 0) (print (nqueen)) (collect-time)))
```

The load balance measured by the above technique is seen as:

```
[0](0 6657)
[0](1 454)
[0](2 412)
[0](3 442)
[0](4 424)
[0](5 500)
[0](6 496)
[0](7 421)
[0](8 512)
[0](9 382)
[0](10 399)
[0](11 611)
[0](12 495)
[0](13 418)
[0](14 428)
[0](15 404)
```

8 Competing Server Array

A competing server array is a system consisting of one master and a number of servers which are simultaneously finding a solution. When one of the servers came to find a solution, it reports the result to the master and then the master instructs the servers to terminate the searching job.
The following program is an implemetation of the FindingSimple algorithm of Jonathan Greenfield[4].
A skelton of the competing server array is a set of programs; master and server.

```
(defun master (x timelimit)
 (lets ((inputlist) (result))
```

```
  (broadmessage (list 'server x))
  (loop (cond ((filter inputlist '(lambda (x) (member 'result x)))
               (broadmessage (list 'complete))
               (exit))
              ((> (time) timelimit)
               (broadmessage (list 'complete)))))
  (cond ((setq result (filter inputlist '(lambda (x) (member 'result x))
         (cadar result)))))))

(defun server (x)
 (lets ((y) (z) (found) (inputlist))
  (initialize x y z)
  (loop (cond ((filter inputlist '(lambda (x) (member 'complete x)))
               (exit))
              ((null found)
               (cond ((setq y (test))
                      (sendto 0 (list 'result y))
                      (setq found t))
                     (t (update)))))))

(setq my-break (inputdata)
  (setq inputlist (nconc inputlist (ncons inputdata)))
  (setq lambda 1))
```

This program works as follows:
`master` starts with `x` and `timelimit` as parameters. After sending a list `(server x)` `x` being replaced by its value, `master` waits in the loop until some server sends a message `(result y)` or timelimit counts up.
In either cases, it broadcast the message `(complete)` to instruct servers to stop searching.
`server` is a program running on the cells other than 0. It first initialize the variables, and the start looping. Inside the loop, each time, it checks the incoming message `(complete)` and on finding it, it exits from the loop. Othewise, it tries to find the answer by calling `test` which returns answer if it finds one, or it returns nil. The `server` update variables and repeats the process.

9 Questions and Comments

Questions and comments shall be sent to either of the authers.

Eiiti Wada (Fujitsu Laboratory)	wada@u-tokyo.ac.jp
Tetsurou Tanaka (University of Tokyo)	tanaka@ipl.t.u-tokyo.ac.jjp

参考文献

[1] Wada Laboratory: “UtiLisp Manual Revision 2.0”, January 1988

[2] Fujitsu Parallel Computing Research Facilities: “AP1000 Library Manual (Cell) C Language Interface”, Fujitsu Laboratories Ltd.

[3] O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare: “Structured Programming”, Academic Press, 1972.

[4] Jonathan S. Greenfield: "Distributed Programming Paradigms with Cryptography Applications", Lecture Notes in Computer Science 870, Springer-Verlag, 1994.