One method for implementation LISP interpreter to transputers

Jozef Kratica
Faculty of Mechanical Engineering
University of Belgrade
27. Marta 80 (440), 11000 Belgrade

Abstract

This paper describing one way for implementation LISP interpreter to transputers. Realized interpreter contains standard functions common for almost all LISP versions.

Architecture is binary tree message passing. Implementation was realized on transputer parallel C language (ANSI C with procedures for interprocessor communications and synchronization). Part for evaluating functions (expressions) was parallelized, but I/O operation and parsing were sequential. Reasons are in technical limitations of transputer systems, because I/O operations can executing only first transputer, and slow interprocessor communication.

Speedup is maximum 6.5 times, on transputer system with 17 transputers T800, by alone transputer T800. That speedup is in recursive problems with much computing. Small speedup is in problems with more I/O operations.

1. Implementation method

In LISP implementation on uniprocessor machines (Š1Ć, Š2Ć) basic part for parallelizing is part for evaluating **expressions (functions)**. Providing that only first transputer can perform I/O operations, that operations (I/O) must executing sequentially. Parsing functions are also executing on first transputer, because interprocessor communication is slow. First transputer sends function definitions to other transputers, when they need it (when other transputers evaluate functions).

Technical limitations of tranputer systems are (Š3Ć):

a) Every transputer have 4 links to other transputers

b) Every transputer must be resetting (one of their 4 links) by other transputer. Only first transputer is resetting by host.

c) Every transputer can reset maximal 2 other transputers, one by system, one by subsystem reset link.

Graph theory define precisely technical limitations by term RS complete graph maximal degree 4. Š2Ć

Binary tree architecture satisfies technical constrains of transputers (RS complete graph maximal degree 4) Š2Ć. Binary tree architecture is applied in this paper..

Transputers can group in 3 categories:

a) First transputer

b) Transputers that have successors (transputer with numbers 2-8.)

c) Transputers which haven't successor (other 9 transputers).

File with NIF extension describe architecture (configuration) of transputer system. Example of NIF file for our implementation, which contains 17 transputers T800 is show bellow:

```
1,  lisptr1,   R0,      0  ,  2  ,  3  ,      ;
2,  lisptr2,   R1,      4  ,  1  ,  5  ,      ;
3,  lisptr2,   S1,      6  ,  7  ,  1  ,      ;
4,  lisptr2,   R2,      2  ,  8  ,  9  ,      ;
5,  lisptr2,   S2,      10 ,  11 ,  2  ,      ;
6,  lisptr2,   R3,      3  ,  12 ,  13 ,      ;
7,  lisptr2,   S3,   14 ,  3  ,  17 ,      ;
8,  lisptr2,   R4,   18 ,  4  ,  19 ,      ;
9,  lisptr2,   S4,         ,     ,  4  ,      ;
10,lisptr2,   R5,      5  ,     ,     ,      ;
11,lisptr2,   S5,         ,  5  ,     ,      ;
12,lisptr2,   R6,         ,  6  ,     ,      ;
13,lisptr2,   S6,         ,     ,  6  ,      ;
14,lisptr2,   R7,      7  ,     ,     ,      ;
17,lisptr2,   S7,         ,     ,  7  ,      ;
18,lisptr2,   R8,      8  ,     ,     ,      ;
19,lisptr2,   S8,         ,     ,  8  ,      ;
```

Every line contains:

a) Number of transputer (first transputer must be connect to host by link 0)
b) Name of program that will be executes on that transputer
c) R or S (system or subsystem reset), and number of transputer which will reset him
d) Number of transputer which connecting link 0
e) link 1
f) link 2
g) link 3

Free connection by that link marking empty place.

Example: 5. Transputer execute LISPTR2, resetting by subsystem link of 2. Transputer (SŽ). Link 0 connect to transputer number 10, link 1 to transputer 11, link 2 to transputer number 2. Link 3 is free.

Configuration of transputer system given in previous NIF file is binary tree (Figure 1). More about configuration of transputer network see Š2Ć and Š3Ć.

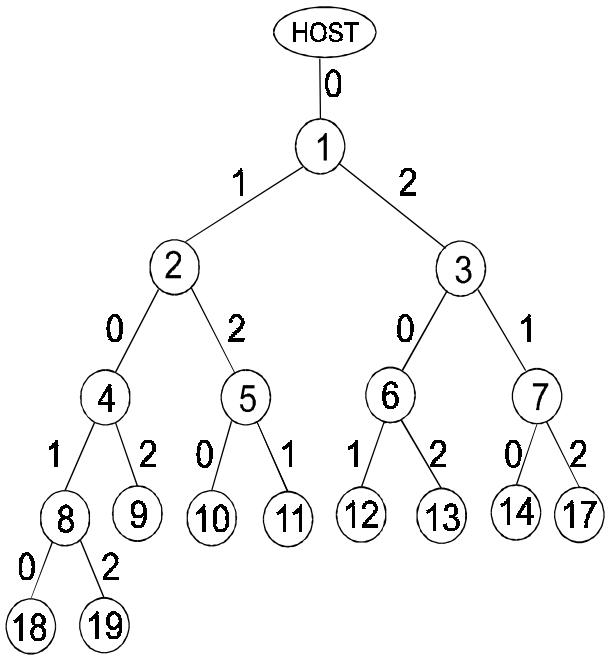

Figure 1.

1.1 Work of first transputer

First transputer does:

a) Loading input data

b) Parsing input data for definitions of user defined functions

c) Saving that definitions

d) Saving names of variables and functions

e) Parsing function calls from input data

f) Printing output results

g) Deciding about executing of functions (executing function alone or send it to "successors")

1.1.1 Call of user defined function

If the first transputer evaluating user defined function, may be two cases:

a) If that function contains only calls of built-in functions, firsts transputer evaluate all part of that function alone, because in many cases this evaluation is short.

b) In case that user defined function also contains calls of other user defined function (functions), we can expect much computing. In that case, if some "successors" are free, this transputer sends parts of this user defined function to free "successors" for evaluation. If all "successors" are busy, then this transputer evaluates all function calls alone.

1.1.2 Call of built-in function

In this case, first transputer performs all computing alone, because evaluating calls of built-in functions are short.

1.2 Work of transputers that have "successors"

Every transputer that has "successors" (in our configuration of 17 transputers, that are transputers no. 2.-8.), waiting message "COMPUTE" from "parent" transputer.

After receiving message "COMPUTE", he receiving following data:

a) Expression (function) which he will evaluating.

b) Names and values of variables in that expression.

c) Definitions of functions, which our expression (function) need for evaluating.

d) Contents of argument stack in that moment

After that, he evaluates function call, on equal way as first transputer.

After end of evaluating that function call, transputer send "FREE" command to "parent", and save result to his communication stack.

In moment in which "parent" need this result, transputer load this value from his communication stack and send that value to "parent".

1.3 Work of transputers that have not "successors"

Every transputer that has not "successors" (in our configuration of 17 transputers, that are transputers no. 9.-19.), waiting message "COMPUTE" from "parent" transputer.

After message "COMPUTE", he receiving following data:

a) Expression (function) which he will evaluating.

b) Names and values of variables in that expression.

c) Definitions of functions, which our expression (function) need for evaluating.

d) Contents of argument stack in that moment

After that, he evaluates function call alone (because he has not "successors")

After end of evaluating that function call, transputer send "FREE" command to "parent", and save result to his communication stack.

In moment in which "parent" need this result, transputer load this value from his communication stack and send that value to "parent".

2. Way of implementation

Implementation of LISP interpreter for transputers (multiprocessors), was based on implementation for uniprocessor machines [2]. Changes in parts of implementation for uniprocessor machines are minor. Implementation for multiprocessors (transputers) has two new parts:

1. Argument passing
2. Control part

In our implementation exist two segments of program:

a) Segment which will be executing on first transputer

b) Segment that will be executing on other transputers. This segment not contains procedures which other transputers cannot executing (I/O operations, parsing, ...).

2.1 Segment for first transputer

2.1.1 Argument passing

Parallel C contains only procedures for passing integers or characters to (from) communication channels. We using complex and powerful data structures (pointers,

linked lists, ...) and need procedures for passing those data structures to (from) communication channels. This part of program contains procedures that enable those possibilities.

2.1.2 Control part

This is most important part of program.

He doing:

a) Receiving messages from input channels, and perform his commands.

b) Note transputers which end previous evaluating, and now are free.

c) When he evaluating function calls, he analyzing cases: if transputer has "successors", and if his "successors" are free, and if expression is user defined function. In that case send function to evaluate to first free "successor". In other case evaluate function call alone.

d) Send message "GIVE ME" to "successor", for his computed value. Then wait for value, and then receive it.

2.2 Segment for other transputers

On other transputers some procedures are disposed as unnecessary. Some procedures are new.

In part **Argument passing** new procedures are procedures for communication stack (unnecessary for first transputer).

In part **Control parts** exist some extra cases:

a) Receiving function for evaluation (and all need data) from "parent".

b) Message "GIVE ME", from "parent".

c) Message "END" from "parent". This message means, that is end of interpreter work. In that moment, end executing program, and exit from interpreter to operating system.

3. Efficiency of implementation

This implementation is efficient, in case that number of operations is much, and recursive oriented solutions. However, speedup depends of nature of problem.

Testing was performed with few tests. Speedup is good only for problems with small I/O operations, and much computing operations. In other case (much I/O operations) speedup is small, because communication time for one datum is 4 time much to time for arithmetic operation on that datum.

In tables 1.-3. all given times are in ms. Error of measurement is most ±5ms.

In different rows of one table are results for different arguments.

In each row are given:

a) Arguments of functions.

b) Execution time on 1 transputer.

c) Execution time on configuration with 3 transputers, and speedup to time on 1 transputer.

d) Execution time on configuration with 7 transputers, and speedup to time on 1 transputer.

e) Execution time on configuration with 15 transputers, and speedup to time on 1 transputer.

f) Execution time on configuration with 17 transputers, and speedup to time on 1 transputer.

Example 1: Function with 2 recursive calls:

```
(defun t2 ( x )
    (if (= x 0)
        1
        ( + (t2 (- x 1)) (t2 (- x 1)))))
```

X	1 tr.	3 tr.	spe.	7 tr.	spe.	15 tr.	spe.	17 tr.	spe.
10	741	428	1.731	232	3.194	125	5.928	125	5.928
11	1481	850	1.742	457	3.241	237	6.249	237	6.249
12	2961	1695	1.747	906	3.268	461	6.423	461	6.423
13	5921	3384	1.75	1804	3.282	911	6.499	910	6.507
14	11841	6764	1.751	3541	3.344	1809	6.546	1808	6.549
15	23681	13522	1.751	7073	3.348	3605	6.569	3604	6.57
16	47362	27039	1.752	14138	3.35	7197	6.581	7197	6.581
17	94722	54073	1.752	28267	3.351	14382	6.586	14382	6.586

Table 1.

Evaluation of (t2 17) sees on Figure 2.

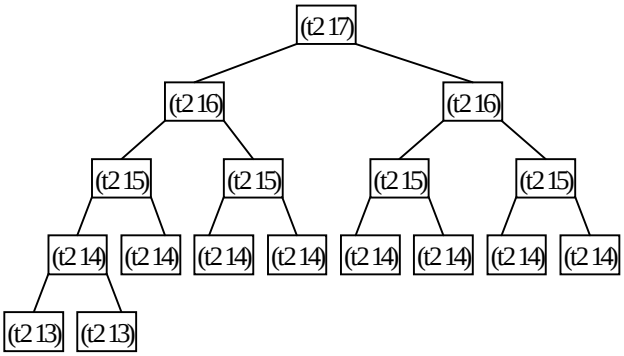

Figure 2.

Example 2: Recursive search of Fibonacci numbers:

```
(defun fib ( x )
    (if (< x 2)
        x
        (+ (fib (- x 1)) (fib (- x 2)))))
```

X	1 tr.	3 tr.	spe.	7 tr.	spe.	15 tr.	spe.	17 tr.	spe.
10	65	49	1.327	36	1.806	26	2.5	22	2.955
15	710	502	1.414	337	2.107	201	3.532	138	5.145

16	1148	809	1.419	543	2.114	317	3.621	216	5.315
17	1857	1306	1.422	874	2.125	506	3.67	343	5.414
18	3004	2111	1.423	1411	2.129	826	3.637	548	5.482
19	4860	3412	1.424	2279	2.132	1310	3.71	879	5.529
20	7862	5519	1.425	3684	2.134	2115	3.717	1416	5.552
21	12721	8927	1.425	5957	2.135	3418	3.722	2284	5.57
22	20582	14433	1.426	9476	2.172	5523	3.727	3688	5.58
23	33301	23353	1.426	15311	2.175	8932	3.728	5961	5.586
24	53881	37758	1.427	24761	2.176	14453	3.728	9639	5.59

Table 2.

Evaluation of (fib 24) sees on Figure 3.

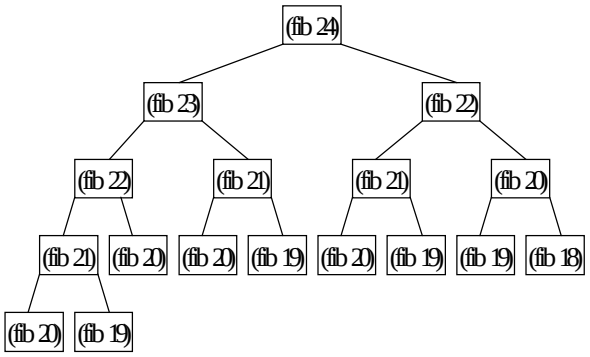

Figure 3.

Example 3: Some problems have many I/O operations, many communications, or some other technical limitation of transputers. Evaluation of that problems have small speedup, or work slowly than one transputer. Example is program which forming "big" list.

```
(define form (n)
( if (= n 0) (set list (cons '1 list))
    (begin
     (form (- n 1))
     (form (- n 1))
     (form (- n 1))
     (form (- n 1))
     (form (- n 1)))))
(set list '())
```

X	1 tr.	3 tr.	spe.	7 tr.	spe.	15 tr.	spe.	17 tr.	spe.
2	53	63	0.841	63	0.841	63	0.841	63	0.841
3	293	344	0.852	344	0.852	344	0.852	344	0.852

4	1461	1721	0.849	1721	0.849	1721	0.849	1721	0.849
5	7338	8633	0.85	8633	0.85	8633	0.85	8633	0.85

Table 3.

4. Conclusion

Most implicit parallel languages implementing functional programming languages. Reasons for using functional paradigm (not procedural) are:

1. Smaller kernel of language.
2. Precise grammar, and then construction are uniform.
3. No side effects.
4. Easy writing recursive function.
5. No explicit sequence of execution.

Because that, parallel implicit programming languages are most popular.

In this paper was constructed interpreter for LISP that implicitly solve problems of communication and synchronization between processors. This way is most general, but not produce fastest code. Code is equal as code for uniprocessor machines, and all programs written to sequential LISP work. But programmer manually can writing fastest code (on explicit parallel programming languages, like Parallel C or Occam).

Architecture is binary tree. This means easier control of processors (communication and synchronization), but also means unnecessary waiting of some processors (transputers). Complex architecture (than tree) can reducing waiting of processors, but also enlarge communication.

Ways for improving this implementation are:

1. Implementing new built-in functions to Common LISP standard. Problem is number of (few thousand) built-in functions in Common LISP.
2. Complex architecture of transputer system. New generation of transputers has more interprocessor channels (16) than this generation (4). This means that architecture can be more complex, and speedup can be greater.

5. References

[1] **Kamin N. S.**, "*Programming languages - An interpreter based approach*", Addison-Wesley, 1990.

[2] **Kratica J.,** *"Paralelization of functional programming languages and implementation to transputer systems"*, Mag. thesis, 1994.

[3] *"Transputer Toolset"*, Inmos corp., 1989.