FILOMAT (Niš) 9:2 (1995), 367–376

Filomat '94, Niš, October 22-24, 1994
Geometry. Computer Sciences

ONE METHOD OF IMPLEMENTATION OF LISP INTERPRETER TO TRANSPUTERS

Jozef Kratica

ABSTRACT. *The paper describes one method of implementation of LISP interpreter to transputers. Developed interpreter contains standard functions common for almost all LISP versions. Architecture is binary tree message passing. Implementation was developed on transputer parallel C language (ANSI C with procedures for interprocessor communications and synchronization). Part intended for evaluation of functions (expressions) was parallelized, but I/O operation and parsing were sequential. This is caused by the technical limitations of transputer systems, because I/O operations can executed only by first transputer, and interprocessor communication is slow. Maxima increase in speed equals 6.5 times, on transputer system with 17 transputers T800, by as compared to single transputer T800. That increase in speed is obtained for recursive problems demanding much computing. Small increase in speed is obtained for problems with more I/O operations.*

1. Implementation method

In LISP implementation on uniprocessor machines ([2], [3]), the basic part for parallelizing is part for evaluating expressions (functions). Provided that only first transputer can perform I/O operations, these operations (I/O) must be executed sequentially. Parsing functions are also executed on the first transputer, because interprocessor communication is slow. First transputer sends function definitions to other transputers when they need them (when other transputers evaluate functions).

Technical limitations of transputer systems are ([7]):

a) Every transputer have 4 links to other transputers;

b) Every transputer must be reset (one of its 4 links) by other transputer. Only first transputer is reset by the host.

c) Every transputer can reset maximally another 2 transputers, one by system, and the other by subsystem reset link.

Graph theory defines precisely technical limitations by term RS complete graph maximal degree 4. [3]

Binary tree architecture satisfies technical constrains of transputers (RS complete graph maximal degree 4) [3]. Binary tree architecture is applied in this paper.

Transputers can be grouped in 3 categories:

a) The first transputer;

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

c) Transputers which have no successor (other 9 transputers).

File with NIF extension describes architecture (configuration) of the transputer system. Example of NIF file for our implementation, which contains 17 transputers T800 is shown below:

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 the transputer (the first transputer must be connected to the host by link 0);

b) Name of a program that will be executed on that transputer;

c) R or S (system or subsystem reset), and number of the transputer which will reset him;

d) Number of the transputer which is connected by link 0;

e) link 1;

f) link 2;

g) link 3;

Free connection by that link marking empty place.

Example: 5. Transputer execute LISPTR2, reset by subsystem link of 2. Transputer (S2). Link 0 connects to transputer number 10, link 1 to transputer 11, link 2 to transputer number 2. Link 3 is free.

Configuration of the transputer system given in previous NIF file is binary tree (Figure 1). More about a configuration of a transputer in a network is presented [3] and [7].

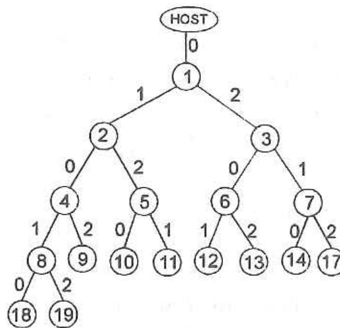

FIGURE 1. Architecture of the transputer system

1.1 Work done by the first transputer.

First transputer performs following operations:

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 the execution of the functions (whether to execute function itself, or to send it to "successors").

1.1.1 Calling of user-defined function.

If the first transputer evaluates user defined function, two cases can arise:

a) If the function contains only calls of built-in functions, the first transputer itself evaluates all parts of the function, because in many cases this evaluation is short.

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

1.1.2 Calling of built-in function.

In this case, the first transputer performs all computing alone, because the evaluation of calls of built-in functions is short.

1.2 Work performed by the transputers that have "successors".

Every transputer that has "successors" (in this configuration of 17 transputers, these are transputers Nos. 2-8), waits for the message "COMPUTE" from "parent" transputer.

After receiving the message "COMPUTE", it receives the following data:

a) Expression (function) that it will evaluate;

b) Names and values of variables in that expression;

c) Definitions of functions, that the expression (function) needs for the evaluation;

d) Contents of argument stack in that moment.

After that, the transputer evaluates function calls, in the same way that the first transputer does. After the end of the evaluation of that function call, the transputer sends the "FREE" command to the "parent", and saves a result to its communication stack.

In the moment in which the "parent" needs this result, the transputer loads this value from his communication stack and sends it to the "parent".

1.3 Work performed by transputers that have no "successors".

Every transputer that has no "successors" (in this configuration of 17 transputers, these are transputers Nos. 9-19), waits the message "COMPUTE" from the "parent" transputer.

After the receival of the message "COMPUTE", ite receives the following data:

a) Expression (function) which it will evaluate;

b) Names and values of variables in that expression;

c) Definitions of functions, which the expression (function) needs for evaluation;

d) Contents of argument stack in that moment. After that, it itself evaluates the function call (because it has no "successors").

After the end of the evaluation of the function call, a transputer sends the "FREE" command to a "parent", and saves the result to its communication stack. In the moment in which the "parent" needs this result, the transputer loads this value from its communication stack and sends it to the "parent".

2. Realization

The implementation of LISP interpreter for transputers (multiprocessors), was based upon the corresponding implementation for uniprocessor machines

[3]. Changes in parts of implementation for uniprocessor machines are minor. The implementation for multiprocessors (transputers) contains two new parts:

1. Argument passing and
2. Control part

In this implementation there are two segments of the program:

a) the segment which will be executed on the first transputer;

b) the segment that will be executed on the other transputers. This segment does not contain the procedures which other transputers cannot execute (I/O operations, parsing, ...).

2.1 The segment for the first transputer.

2.1.1 Argument passing.

The parallel C contains only procedures intended for passing of integers or characters to (from) communication channels. In the program the complex and powerful data structures (pointers, linked lists, ...) and procedures necessary for passing those data structures to (from) communication channels were used. This part of the program contains procedures that enable those possibilities.

2.1.2 The control part.

This is the most important part of the program.

It performs following operations:

a) receives messages from input channels, and performs their commands;

b) takes note of transputers which ended theis previous evaluation, and now are free;

c) when it evaluates function calls, it analyses following cases: if the transputer has "successors", if its "successors" are free, and if expression is user-defined function, then it sends a function to be evaluatet to the first free "successor". In other case it itself evaluates a function call;

d) it sends the message "GIVE ME" to a "successor", demanding the value it computed. Then it waits until it receives the value.

2.1.3 The segment intended for other transputers.

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

In the part Argument passing new procedures are procedures intended for the communication stack (not necessary for the first transputer).

In the part Control parts there are several operations to be performed:

a) receival of function intended for evaluation (and all necessary data) from the "parent".

b) receival of the message "GIVE ME", from the "parent";

c) receival of the message "END" from the "parent". This message means the end of the interpreter work. In that moment, execution of program ends, and the user exits from the interpreter to the operating system.

3. The efficiency of the implementation

This implementation is efficient, in case of the great number of operations, and recursive oriented solutions. However, increase in speed depends upon the nature of a problem.

The testing was performed using few test examples. The increase in speedup was notable only for problems with a small number of I/O operations, and a great number of computing operations. In the alternate case (a great number of I/O operations) the increase in speed is small, because the communication time for one datum is 4 times greater than the time needed for the arithmetic operation on that datum.

In Tables 1-3 all times are given in ms. The maximal error of measurement equals 5ms.

The results for different arguments are given in different rows of each table.

In each row are given:

a) arguments of functions;

b) the execution time on 1 transputer;

c) the execution time on configuration with 3 transputers, and increase in speedup in comparison to time on 1 transputer;

d) the execution time on configuration with 7 transputers, and increase in speed compared to the time on 1 transputer.

e) the execution time on configuration with 15 transputers, and increase in speed compared to the time on 1 transputer;

f) the execution time on configuration with 17 transputers, and increase in speed compared to the time on 1 transputer;

Example 1: The function with 2 recursive calls:

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

The method of evaluation of (t2 17) is given in Fig. 2.

Example 2: The recursive search of Fibonacci numbers:

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

The method of evaluation of (fib 24) is presented in Fig. 3.

TABLE 1. Times for the Example 1

X	1 tr.	3 tr.	spe.	7 tr.	spe.	15	spe.	17	spe.
10	741	428	1.73	232	3.19	125	5.92	125	5.92
11	1481	850	1.74	457	3.24	237	6.24	237	6.24
12	2961	1695	1.74	906	3.26	461	6.42	461	6.42
13	5921	3384	1.75	1804	3.28	911	6.49	910	6.50
14	11841	6764	1.75	3541	3.34	1809	6.54	1808	6.54
15	23681	13522	1.75	7073	3.34	3605	6.56	3604	6.5
16	47362	27039	1.75	14138	3.35	7197	6.58	7197	6.58
17	94722	54073	1.75	28267	3.35	14382	6.58	14382	6.58

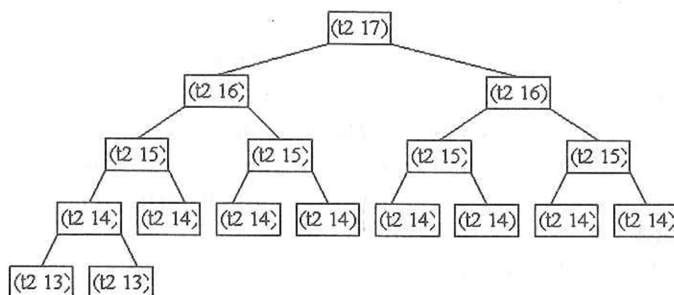

FIGURE 2. Scheme of evaluation for example 1

TABLE 2. Times for the Example 2

X	1 tr.	3 tr.	spe.	7 tr.	spe.	15	spe.	17	spe.
10	65	49	1.32	36	1.80	26	2.5	22	2.95
15	710	502	1.41	337	2.10	201	3.53	138	5.14
16	1148	809	1.41	543	2.11	317	3.62	216	5.31
17	1857	1306	1.42	874	2.12	506	3.67	343	5.41
18	3004	2111	1.42	1411	2.12	826	3.63	548	5.48
19	4860	3412	1.42	2279	2.13	1310	3.71	879	5.52
20	7862	5519	1.42	3684	2.13	2115	3.71	1416	5.55
21	12721	8927	1.42	5957	2.13	3418	3.72	2284	5.57
22	20582	14433	1.42	9476	2.17	5523	3.72	3688	5.58
23	33301	23353	1.42	15311	2.17	8932	3.72	5961	5.58
24	53881	37758	1.42	24761	2.17	14453	3.72	9639	5.59

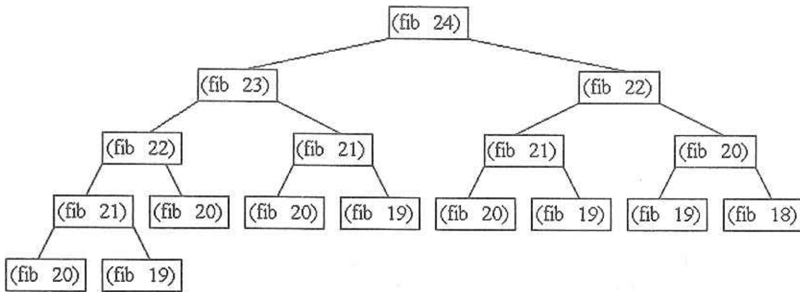

FIGURE 3. Scheme of evaluation for example 2

Example 3: Some problems have a great number of I/O operations, a lot of communication, or their execution is slow due to certain technical limitations of transputers. In the evaluation of those problems a small increase in speed is obtained, or, conversely, more time is needed than on one transputer. The example of such a problem is the program which forms a "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 '())
```

TABLE 3. Times for the Example 3

X	1	3	spe.	7	spe.	15	spe.	17	spe.
2	53	63	0.84	63	0.84	63	0.84	63	0.84
3	293	344	0.85	344	0.85	344	0.85	344	0.85
4	1461	1721	0.84	1721	0.84	1721	0.84	1721	0.84
5	7338	8633	0.85	8633	0.85	8633	0.85	8633	0.85

Remaining methods of implementation are extensively described in [1]. Some of them are:

1. Translation of a program code into metalanguage, that is more suitable for evaluation [6];

2. Division of the problem into subproblems (divide and conquer approach) [5];

3. Translation of the program into a code that does not the requirements of speed, and after that, during run-time, automatic improvement of its performances [4].

4. Conclusions

Most implicit parallel languages implement functional programming languages. Reasons for using functional paradigm (insted of the procedural one) are:

1. Smaller kernel of language;
2. A precise grammar, and, consequently, uniform constructions;
3. No side effects;
4. Easy writing of recursive functions;
5. No explicit sequence of execution.

Because of that, parallel implicit programming languages are most popular.

In this paper the interpreter for LISP that implicitly solves problems of communication and synchronization between processors was developed. This method is the most general one, but it does not, in the same time, produce the fastest code. The code is equal to the code used for uniprocessor machines, and all programs written in sequential LISP can operate on those machines as well. But a programmer can manually write the fastest code (in explicit parallel programming languages, like Parallel C or Occam).

Architecture is binary tree. This means easier control of processors (communication and synchronization), but it also means the unnecessary waiting of some processors (transputers). A more complex architecture (than the tree) can reduce waiting of processors, but it will also increase a communication.

The methods of improving this implementation are:

1. the implementation of new built-in functions in accordance with the Common LISP standard. It should be noted that there are few thousand of built-in functions in Common LISP;

2. using more complex architecture of the transputer system. New generation of transputers has more interprocessor channels (16) than this generation (4). This means that architecture can be more complex, and the increase in speed can be greater.

References

[1] Ashcroft E.A., Faustini A.A., Jagannathan R., *An Intensional Language for Parallel Aplication Programming,* Parallel Functional Languages and Compilers, ACM Press, 1991, pp. 11 – 50.

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

[3] Kratica J., *Paralelization of functional programming languages and implementation to transputer systems,* Mag. thesis, University of Belgrade, Faculty of Mathematics, 1994.

[4] Leung S., Zahorjan J., *Improving the Performance of Runtime Paralelization,* Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (1993), ACM Press, 208–217.

[5] Mou Z.G., *A Formal Model for Divide-and-Conquer and its Parallel Realization,* Ph.D. thesis, Yale University, Department of Computer Science, 1990.

[6] Skedzielevski S.K., Glauert J., *IF1 - An intermediate form for aplicative languages,* Manual M-170, Lawrence Livermore National Labaratory, 1985.

[7] *Transputer Toolset,* Inmos corp., 1989.

27. Marta 80, 11000 Belgrade