BUCHBERGER

Computer trees: a concept for parallel processing

B Buchberger, J Fegerl* and F Lichtenberger report on a multimicrocomputer system currently being investigated for special purpose parallel processing

This paper describes the multimicrocomputer concept called 'computer tree' which is currently being investigated at the institute. The first part summarizes the global idea and characteristic features of the concept and complexity considerations concerning the main application of computer trees: parallel execution of recursive algorithms. The paper then goes on to describe a simple hardware implementation of the concept, which has been undertaken in a first stage of the project, and a proposal for a future hardware implementation that goes beyond the capabilities of the first in that a flexible connection of processor modules and, thereby, an adaptation of the hardware structure to the structure of the algorithm during execution time is provided.

BASIC FEATURES

The idea for the concept of *computer trees*[1-3] derives from the observation that the execution of recursive procedures on present day computers (of the Von Neumann type) has to be effected in an unnatural way: instead of delegating different procedure calls to different processors the flow of procedure calls is artificially put into sequence. The parallelism inherent in many recursively formulated algorithms is thus destroyed. The concept of computer trees aims at exploiting this parallelism to reduce the computation time of a wide class of algorithms.

The exploitation of various forms of parallelism in algorithms by multi processor systems for the reduction of computation time has been investigated extensively during the last decade. A compilation and assessment of the literature on the subject is beyond the scope of this paper (see, for instance, references 4–6 and the proceedings of the various conferences on the subject, for instance reference 7). In comparison with other approaches, the concept described here is characterized by the following:

- decentralized control of the (microcomputer) modules
- tree-shape connection between the modules
- main application: recursive algorithms

Of course, one or the other of these aspects is incorporated in other proposals[8-10]. However, until now these aspects have been considered separately whereas they are intimately connected in the present concept, the final aspect being the starting point and the other being natural consequences.

Instut für Mathematik, Johannes Kepler Universität, A-4040 Linz, Austria *Fa. ICA, A-1160 Wein, Austria

In more detail, recursive algorithms are considered of the following and related types

$$F(x) = \textit{if } p(x) \textit{ then } f(x) \textit{ else } h(F(g_1(x)), F(g_2(x)),$$

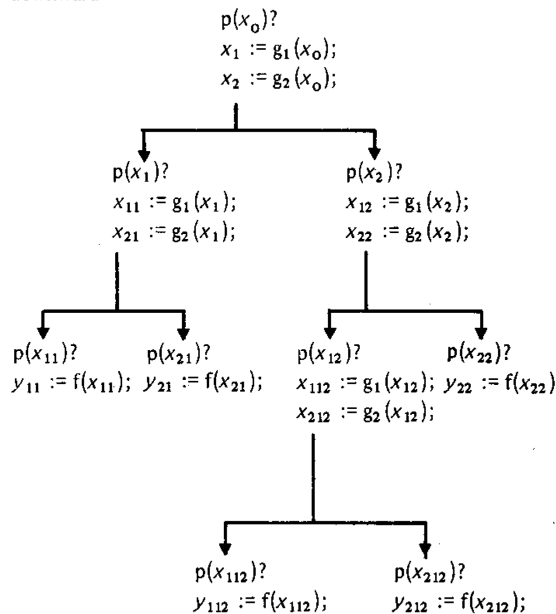
with two or more (parallel) recursive calls of F in the procedure body of F. Many algorithms are of this type (e.g. evaluation of terms, merge-sort, searching in trees; more general 'divide and conquer' algorithms[11]; algorithms for 'hard' problems such as NP- complete problems[11]; or nondeterministic algorithms that may be transformed into recursive form).

Assume that for a certain input value x_0 for the above procedure F

$$\neg p(x_0),$$
$$\neg p(g_1(x_0)), \neg p(g_2(x_0)),$$
$$p(g_1(g_1(x_0))), p(g_2(g_1(x_0))), \neg p(g_1(g_2(x_0))), p(g_2(g_2(x_0))),$$
$$p(g_1(g_1(g_2(x_0)))), p(g_2(g_1(g_2(x_0)))).$$

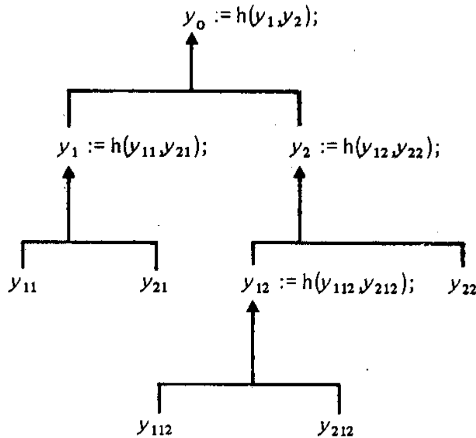
Then the following computation has to be carried out, which divides into a 'downward' computation involving the computation of g_1 and g_2 on several arguments and an 'upward' computation involving the computation of h:

downward:



 0141–9331/79/060244–05 $02.00 © 1979 IPC Business Press

upward:



On a computer of the Von Neumann type the auxiliary values $x_1, x_2, x_{11}, x_{21}, \ldots, x_{212}, y_{212}, \ldots, y_2, y_1, y_0$ have to be computed sequentially. This is because only one central processing unit is available. If several CPUs were available the computation of the values x_{11} and x_{12}, for instance, could take place in parallel. Roughly, instead of nine time units (one time unit corresponding to the execution of the operations at one node of the above computation graph) only four time units would be needed for the above computation if the parallelism inherent in the above computation could optimally be exploited.

The concept of computer trees solves the problem of an optimal exploitation of the above parallelism in a very natural way. A computer tree consists of a large pool of identical (micro-) computers ('modules') that may be connected arbitrarily in order to form tree structures. One module consists of

- processing unit (CPU)
- program memory (PM)
- data memory (DM)

Every module has access to its own data memory and to the data memory of its sons (not to the data memory of its father).

In addition, every module may transmit and receive sensor information in order to control locally the interplay between the modules. More accurately, a computer tree may appear as in Figure 1.

The machine language of the modules must be enriched by two features: an *address modification rule* and *sensor instructions*. These two features are described in an ALGOL like notation because the basic principle of the approach is language independent.

Address modification rule

Every variable (address) x is available in several issues $x, x', x'', \ldots$ with the following semantic interpretation: a module C in the tree, by means of variables (addresses) of the types $x, x', x'', \ldots$, has access to its own data memory, to the data memory of its first son C′, second son C″, . . . , respectively. Furthermore, if module C addresses a storage region of its first son C′ by variable x' then module C′ may address the same region by variable x, and analogously for C and its second, third, . . . son C″, C‴, ...

Sensor instructions

if S then . . . ;
if T_1 then . . . ;
if T_2 then . . . ;

The semantics of these instructions is straightforward.

U := *true;* U := *false*;

By these instructions a module C may set and reset the sensor bit T_i of its father, if C is the *i*th son of its father.

V_1 := *true*; V_1 := *false;*
V_2 := *true;* V_2 := *false*;

These set and reset the sensor bit S in the first, second, . . . son, respectively.

The above recursive algorithm, then, may be executed on a computer tree by loading the following program to the program memory of all modules in a binary computer tree.

```
1: if ¬S then goto 1;
   if p(x) then y := f(x);
             U := true;
             stop;
   if ¬p(x) then x' := g1(x) ; x'' := g2(x);
             V1 := V2 := true;
         2:  if ¬(T1 ∧ T2) then goto 2;
             y := h(y', y'');
             U := true;
             stop;
```

In order to initiate the computation, store the input value x_0 into the storage region x of the data memory of the top module and set the sensor bit S of the top module to *true*. A computational 'wave' is then generated in the tree, which first operates downward and then upward in the tree and thus naturally corresponds to the computation graph shown above.

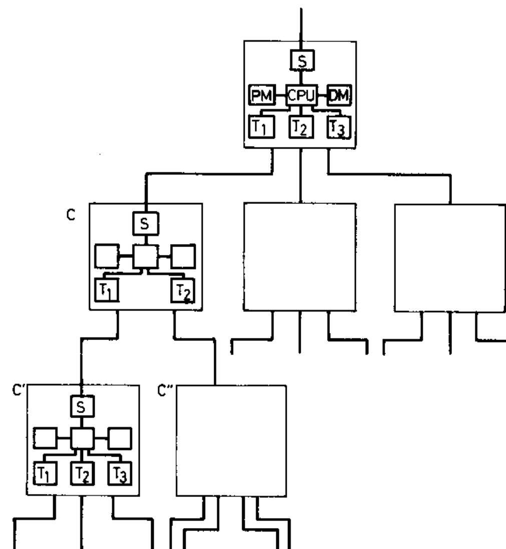


Figure 1. Computer tree. S, T_1, T_2, . . . are sensor bits that may be read by the module in which they are situated, and may be set by the father, first son, second son, . . . respectively

Various examples of programs for computer trees with a detailed explanation of their operation including examples of programs that are not derived from recursive algorithms may be found elsewhere[1-3].

COMPLEXITY

It is easy to conclude from the example in the previous section that a drastic speed-up of algorithms may be achieved on computer trees. For example, if a computation yields a balanced binary computation graph with 2^k-1 nodes (k levels) we need 2^k time units on an ordinary computer of the Von Neumann type but only k time units on a computer tree. According to how the number of levels in the computation graph (k) depends on the problem size n, a typical speed up of $0(2^n) \rightarrow 0(n)$ or $0(n. \log n) \rightarrow 0(n)$ may be achieved[1,3].

Of course, the gain in speed must be purchased at the cost of hardware complexity. It is only through the advent of very cheap microprocessors that such an approach may be relevant practically.

Here one example only is pursued in more detail: the rucksack problem, which is defined as follows:

Given n+1 natural numbers $a_1, a_2, \ldots, a_n$ and b.
Find a set $I \subset (1, 2, \ldots, n)$ such that $\sum_{i \in I} a_i = b$.

This problem belongs to the class of NP-complete problems; every known algorithm for solving the problem on an ordinary computer (see for example reference 12, p 158) has exponentially dependent computation time in the worst case. A program for solving the problem on a binary computer tree with infinitely many modules has been given[3]. The algorithm for solving the rucksack problem essentially is of the type of the recursive function F considered above, and therefore needs only linear time on a computer tree.

In order to estimate the average speed-up which could be achieved by implementing the algorithm on a computer tree, a simulation program has been written. For every problem size $n \leqslant 24$ about 100 tests with randomly generated input data have been performed. The results of the simulation are shown in Table 1, which also lists the number of modules needed for an optimal exploitation of parallelism for every n.

Table 1 shows that in the case of large problem sizes more modules are needed than can be expected in a practical realization of a computer tree. Therefore programs for a computer tree should include the possibility of switching to sequential computation when all modules are exhausted. In this case the speed-up is essentially the number of bottom modules of the computer tree.

Table 1. Results of computer tree simulation

Problem size	Speed-up factor Average	Speed-up factor Maximum	Modules needed Average	Modules needed Maximum
10	7	23	26	102
12	15	45	75	250
14	33	141	192	911
16	103	404	704	3214
18	266	1374	2095	10556
20	819	4942	6830	42535
22	2759	8755	26102	89190
24	8754	40501	87540	388101

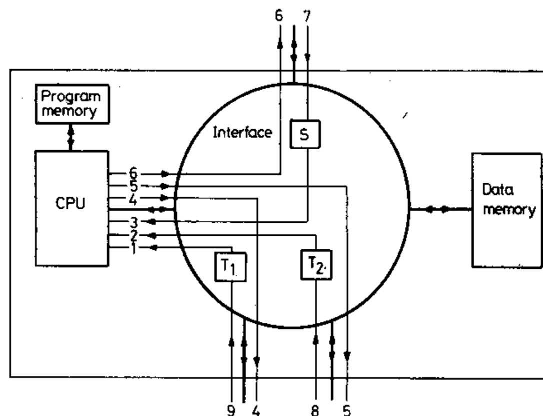


Figure 2. Structure of module with two sons. Lines 1,2,3 enable the CPU to read T_1, T_2, S; lines 4,5,6 enable the CPU in the module to set and reset S in the right and left hand son and T_1 (or T_2) in the father; lines 7,8,9 enable the father and the right and left sons to set and reset S, T_1, T_2 respectively

HARDWARE IMPLEMENTATION

First version

In a first stage of the project a hardware implementation of a module which may serve as a node in arbitrarily large computer trees was undertaken. This module is designed to have exactly two sons, and its structure is as in Figure 2.

The interface organizes access to the data memory, realizes the address modification rule and contains the sensor bits S, T_1, T_2.

From these modules arbitrarily large binary trees may be composed without any need of an overall control system. The restriction to only two sons, theoretically, is no serious limitation, because one can show [1] how an arbitrary number of sons may be simulated in a binary tree by software.

The drawback of this implementation is that the interconnections between the modules must be established prior to the execution of the programs. Thus the bottom level of the tree may be reached in one branch of the computation although a large number of modules is still available in some other branch of the tree. It is the goal of the next implementation to overcome this limitation.

The implementation of the basic address modification rule in the present modules is as follows. The set A of addresses that are not needed for addressing the program memory or for other special purposes is divided into three disjoint subsets A_0, A_1 and A_2 of equal size and two bijective address mappings are defined:

$f_1 : A_1 \rightarrow A_0, \quad f_2 : A_2 \rightarrow A_0$

For example,

$A := (0, \ldots, 3071), A_0 := (0, \ldots, 1023),$
$A_1 := (1024, \ldots, 2047), A_2 := (2048, \ldots, 3071)$
$f_1(a) := a - 1024, f_2(a) := a - 2048$

The interface must realize the following function:

- an address a produced by the CPU is analysed in the interface. Accesses to the data memory of its own module or of the left and right son are, then, carried out according to the following rule:

if $a \epsilon A_0$, then location a of the data memory of its own module is accessed;
if $a \epsilon A_1$, then location $f_1(a)$ of the data memory of the left son is accessed;
if $a \epsilon A_2$, then location $f_2(a)$ of the data memory of the right son is accessed

- an address a arriving at the interface on the connection line to the father remains unchanged. The father, then, may access location a in the data memory.

A test module of the type described above has been implemented by the second author using the KIM-1 microprocessor. In this concrete implementation the different design objectives have been realized as indicated below.

Components

- Microprocessor : KIM-1
- Program memory : 1k RAM
- Data memory : 1k RAM
- Interface : specially designed component (TTL logic)

Basic address modification rule

$A := K1 \cup K2 \cup K3$, $A_0 := K3$, $A_1 := K1$, $A_2 := K2$
$f_1(a) := a + 2048$, $f_2(a) := a + 1024$

The remaining addresses in the address space of the KIM-1 are used as follows:

K0 : program memory
K4 : special purposes
K5–K7 : KIM-monitor

Sensor bits

Bits 5, 6 and 7 of location 1100 are used for realizing the sensor bits T_1, T_2 and S respectively. By reading these bits the status of T_1, T_2 and S may be determined. Correspondingly, by writing into these bits information may be sent to the sensor bits S in the left and right son and the sensor bit T_1 (or T_2) of the father respectively.

In this concrete realization, then, the interface has the logical structure shown in Figure 3.

Second version

In the next stage of the project it is planned to implement a hardware system that is capable of adaptively composing arbitrary tree structures during execution time according to the actual structure of the computation graph of the algorithm for the respective input. In this paper only a rough idea of the new system can be given.

Figure 3. Logical structure of the interface (SRL is storage reservation logic which is not described here as it does not affect the essential features of the design)

The basic structure of this system will be as in Figure 4 (shown for only four processors and four memory blocks). Figure 5 gives more detail of the microcomputers, which contain private memory for programs and special purposes.

An address m in an instruction of the processor with number k is decomposed into two parts as shown in Figure 6. n determines a location in the storage mapping of the kth processor ($n = 0$ means 'the own data memory of the kth processor'). The content n' of this location is the actual number of the processor that serves as the nth son of the kth processor. Incidentally n' is the number of the memory block serving as data memory for the n'th processor. Then, (n', r) is the address leaving the kth processor. This address is analysed in the addressing unit and determines the location to be addressed in the data memory bank.

The address modification is realized as follows: if the nth son of the kth processor has the number n', then n' must be stored in location n of the private memory of the kth

Figure 4. Planned second version hardware implementation

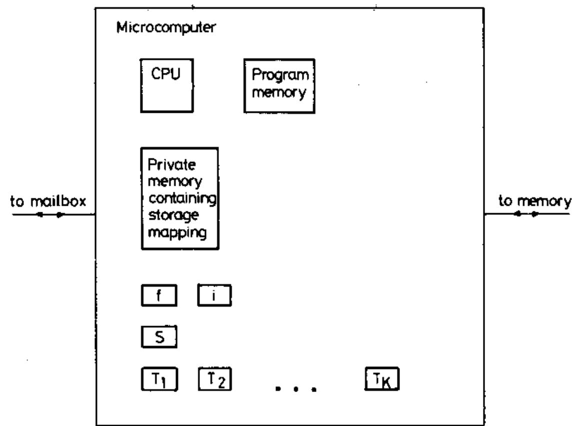


Figure 5. Microcomputer structure. S, $T_1, \ldots, T_K$ are sensor bits, and f and i are locations containing respectively the number of the father and j, a number indicating that the processor is the jth son of its father

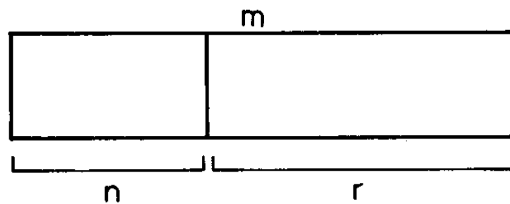


Figure 6. Instruction address m.n is the number of the son and r an address in the memory block of the nth son

processor and in location 0 of that of the n'th processor (location 0 of every processor should contain its own number from the outset).

The appropriate updating of the private memories in the processors is realized during execution time by the control unit. Whenever the kth processor realizes during the execution of an instruction that involves its nth son that location, n of its storage mapping is not yet defined, it interrupts the execution of this instruction temporarily and puts a corresponding request into its mailbox.

The control unit constantly checks the mailboxes of all processors and eventually answers a request using the storage containing the numbers of free processors.

Sensor instructions, too, are executed by putting a corresponding request into the mailbox (for executing U := *true*, and U := *false* the information stored in f and i is necessary).

Of course, the control unit is the bottle neck of the system. In the worst case, if all processors need its assistance at the same moment, the execution of a sensor instruction or of an instruction that is interrupted because of updating operations may need the time pt, where p is the number of processors in the system and t is the time the control unit needs for handling one request.

In normal applications the worst case is very unlikely to occur, because the instructions involving the control unit are scarce. (Note that normal accesses to the data memory do not slow each other down). In any case, the main effort will be to make t as small as possible (by developing a special purpose processor as the core of the control unit and by analysing the parallelism inherent in the operation of the control unit).

Note also that at a given instant a memory block can be accessed by two processors only. This is because in the tree structures generated by this system the number of a memory block may appear in the storage mapping of two processors only. In normal applications, an appropriate use of the sensor instructions guarantees that, in fact, only one processor can access a memory block at a given instant.

CONCLUSIONS

A concept for a multimicroprocessor system was proposed by which the time complexity of a wide class of algorithms may be converted into hardware complexity. This may be an advantage taking into account the present and future development of the costs of hardware components, in particular, of microprocessors. Two versions of possible hardware implementations of the concept were described.

The first version is totally modular but uses fixed connections between the modules. Test modules for this version have already been implemented using the KIM-1 microprocessor. The second version aims at an adaptive generation of the connections between the modules at execution time. This has to be purchased by giving up the extremely modular hardware structure of the first version.

REFERENCES

1 **Buchberger, B** 'Computer trees and their programming' *Proc. Troisieme Colloque de Lille 'Les arbres en algebre et en programmation', February 1978* pp 1–18

2 **Buchberger, B and Fegerl, J** 'A universal module for the hardware implementation of recursion' *Univ. Linz Inst. Math. Bericht* nr 106 (1978)

3 **Lichtenberger, F** 'Speeding up algorithms on graphs by using computer trees' in **Nagl, M and Schneider, H-J** (eds) *Graphs, data structures, algorithms* Carl Hanser Verlag, München–Wein (1979)

4 **Buchberger, B** 'Implementation of an adaptive multi-microcomputer system with tree structure' *German research proposal* (February 1979)

5 **Enslow, P H** (ed) *Multiprocessors and parallel processing* Wiley, New York (1974)

6 **Toong, H D** 'Multimicroprocessor systems' in **Schwärzel, H** (ed) *Forschungs-Symposium 'Systeme mit Mikroprozessoren', Munchen, June 1978*, Siemens AG, München (1978) pp 9–20

7 **Syre, J C** (ed) *Proc. First European Conference on Parallel Distributed Processing, Toulouse, France, (1979)*

8 **Sullivan, H, Bashkow, T R and Klappholz, D** 'A large scale homogenous fully distributed parallel machine, I and II' *Proc. 4th Annual Symposium on Computer Architecture*

9 **Händler, W** 'Aspects of parallelism in computer architecture' in **Feilmeier, M** (ed) *Parallel Computers – parallel mathematics, Proc. IMACS (AICA) Symposium, Munich, March 1977* North-Holland (1977) pp 1–8

10 **Glushkow, V M, Ignatyev, M B, Myasnikov, V A and Torgashev, V A** 'Recursive machines and computing technology' *Proc. IFIP Congress, 1974* North-Holland (1974) pp 65–70

11 **Aho, A V, Hopcroft, J E and Ullman, J D** *The design and analysis of computer algorithms* Addison-Wesley (1974)

12 **Noltemeier, H** (ed) *Graphen, Algorithmen, Datenstrukuren* Hanser Verlag, München–Wein (1976)