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.

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

\[
F(x) = \begin{cases} \text{if } p(x) \text{ then } f(x) \\ \text{else } h(F(g_1(x)), F(g_2(x))) \end{cases}
\]

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\(^{13}\); algorithms for 'hard' problems such as NP-complete problems\(^{12}\); or nondeterministic algorithms that may be transformed into recursive form).

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

\[
\begin{align*}
\neg p(x_0) & , \\
\neg p(g_1(x_0)) & , \\
\neg p(g_2(x_0)) & , \\
p(g_1(g_1(x_0))) & , \\
p(g_2(g_2(x_0))) & , \\
p(g_1(g_2(x_0))) & , \\
p(g_2(g_1(x_0))) & , \\
p(g_2(g_2(x_0))) & .
\end{align*}
\]

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:

\[
\text{downward:}
\]

\[
\begin{align*}
p(x_0) & , \\
x_1 & := g_1(x_0); \\
x_2 & := g_2(x_0); \\
p(x_1) & , \\
x_11 & := g_1(x_1); \\
x_12 & := g_2(x_1); \\
p(x_2) & , \\
x_21 & := g_1(x_2); \\
x_22 & := g_2(x_2); \\
p(x_11) & , \\
x_{111} & := f(x_{11}); \\
x_{112} & := f(x_{112}); \\
p(x_12) & , \\
x_{121} & := f(x_{121}); \\
x_{122} & := f(x_{122}); \\
p(x_21) & , \\
x_{211} & := f(x_{211}); \\
x_{212} & := f(x_{212}); \\
p(x_{111}) & , \\
x_{1111} & := f(x_{111}); \\
x_{1112} & := f(x_{1112}); \\
p(x_{112}) & , \\
x_{1121} & := f(x_{1121}); \\
x_{1122} & := f(x_{1122}); \\
p(x_{121}) & , \\
x_{1211} & := f(x_{1211}); \\
x_{1212} & := f(x_{1212}); \\
p(x_{122}) & , \\
x_{1221} & := f(x_{1221}); \\
x_{1222} & := f(x_{1222}); \\
p(x_{211}) & , \\
x_{2111} & := f(x_{2111}); \\
x_{2112} & := f(x_{2112}); \\
p(x_{212}) & , \\
x_{2121} & := f(x_{2121}); \\
x_{2122} & := f(x_{2122}); \\
p(x_{221}) & , \\
x_{2211} & := f(x_{2211}); \\
x_{2212} & := f(x_{2212}); \\
p(x_{222}) & .
\end{align*}
\]

\[
\text{upward:}
\]

In contrast to the downward computation, the upward computation is driven by the evaluation of h, starting at the leaf nodes, where the actual computations are performed. The intermediate results are stored in memory, and the final result is obtained by combining the results of the upward pass with the results of the downward pass. This approach allows for efficient parallel processing, as the computations can be divided into smaller tasks that can be executed concurrently.
Sensor instructions

\[ \text{if } S \text{ then } \ldots ; \]
\[ \text{if } T_1 \text{ then } \ldots ; \]
\[ \text{if } T_2 \text{ then } \ldots ; \]

The semantics of these instructions is straightforward.

\[ U := \text{true}; \]
\[ U := \text{false}; \]

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.

\[ V_1 := \text{true}; \]
\[ V_1 := \text{false}; \]
\[ V_2 := \text{true}; \]
\[ V_2 := \text{false}; \]

These set and reset the sensor bit \( S \) in the first, second, \ldots, 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.

\[ 1: \text{if } \neg S \text{ then goto } 1; \]
\[ \text{if } p(x) \text{ then } y := f(x); \]
\[ U := \text{true}; \]
\[ \text{stop}; \]
\[ \text{if } \neg p(x) \text{ then } x' := g_1(x); \]
\[ x'' := g_2(x); \]

\[ \text{if } \neg (T_1 \land T_2) \text{ then goto } 2; \]
\[ y := h(y', y''); \]
\[ U := \text{true}; \]
\[ \text{stop}; \]

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 \text{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.

**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 \( C''' \ldots \), 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 \ldots, son \( C'', C''' \ldots \).
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.\textsuperscript{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 is $k$ depends on the problem size $n$, a
typical speed up of $O(2^n) \rightarrow O(n)$ or $O(n \log n) \rightarrow O(n)$ may
be achieved.\textsuperscript{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 $I \subseteq \{1, 2, \ldots, n\}$ 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.\textsuperscript{1} 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 $n \leq 24$ 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>
<thead>
<tr>
<th>Problem size</th>
<th>Speed-up</th>
<th>Modules needed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Factor</td>
<td>Average</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>23</td>
</tr>
<tr>
<td>12</td>
<td>15</td>
<td>45</td>
</tr>
<tr>
<td>14</td>
<td>33</td>
<td>141</td>
</tr>
<tr>
<td>16</td>
<td>103</td>
<td>404</td>
</tr>
<tr>
<td>18</td>
<td>266</td>
<td>1374</td>
</tr>
<tr>
<td>20</td>
<td>819</td>
<td>4942</td>
</tr>
<tr>
<td>22</td>
<td>2759</td>
<td>8755</td>
</tr>
<tr>
<td>24</td>
<td>8754</td>
<td>40501</td>
</tr>
</tbody>
</table>

**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 how an
arbitrary number of sons may be simulated in a binary tree
by software.

The drawback of this implementation is that the inter-
connections 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 $A_0, A_1, A_2$ of equal size and two
bijective address mappings are defined:

- $f_1 : A_1 \rightarrow A_0$
- $f_2 : A_2 \rightarrow A_0$

For example,

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

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

- an address \( \alpha \) arriving at the interface on the connection line to the father remains unchanged. The father, then, may access location \( \alpha \) 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

\[
A := K1 \cup K2 \cup K3, \quad A_0 := K3, \quad A_1 := K1, \quad A_2 := K2 \\
f_1(\alpha) := \alpha + 2048, \quad f_2(\alpha) := \alpha + 1024
\]

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

- K0: program memory
- K4: special purposes
- K5–K7: KIM-1 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.

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 \( m \) of an instruction of the processor with number \( k \) is decomposed into two parts as shown in Figure 6. \( k \) determines a location in the storage mapping of the \( k \)th processor \((n=0 \text{ means 'the own data memory of the } k \text{th processor'})\). The content \( n' \) of this location is the actual number of the processor that serves as the \( n \)th son of the \( k \)th 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 \( k \)th 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 \( n \)th son of the \( k \)th processor has the number \( n' \), then \( n' \) must be stored in location \( n \) of the private memory of the \( k \)th processor.

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)

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 $i$, a number indicating that the processor is the $j$th son of its father.

Figure 6. Instruction address $mn$ is the number of the son and $r$ an address in the memory block of the $n$th 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 $k$th processor realizes during the execution of an instruction that involves its $n$th son thst 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 bottleneck 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
3 Lichtenberger, F 'Speeding up algorithms on graphs by using computer trees' in Nagl, M and Schneider, HJ (eds) Graphs, data structures, algorithms Carl Hanser Verlag, Munich–Wein (1979)
4 Buchberger, B 'Implementation of an adaptive multiprocessor system with tree structure' German research proposal (February 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