Invited paper
AAECC Conference
Grenoble 1985

THE L-MACHINE: AN ATTEMPT AT PARALLEL HARDWARE FOR SYMBOLIC COMPUTATION

B. Buchberger
Johannes-Kepler-University
A4040 LINZ (Austria, Europe)

INTRODUCTION

This is a survey on the L-machine research project at the University of Linz. The L-machine is a parallel machine whose design objective is the execution of all types of parallel algorithms, in particular non-numerical (symbolic) algorithms. On the L-machine it should be possible to exploit the parallelism inherent in many algorithmic ideas in a natural way. As the main implication this requires that the interconnection topology of the processor modules of the machine should be easily adaptable to the parallel algorithm at hand. The L-machine is a highly modular structure whose building blocks are universal processor/memory modules, called L-modules, that can be interconnected in arbitrary ways. Since the partial processes realized in the L-modules can be asynchronous a flexible synchronization mechanism is necessary, which is realized by programmable sensor bits.

The L-machine project has been started in 1978, see [Buchberger 78]. Various extensions of the original concept are documented in the references. In this paper it is not possible to compare the L-machine concept with the various other parallel machine concepts for symbolic computation, in particular with other parallel inference machines. A comprehensive bibliography on parallel processing is [Bernutat-Buchmann, Rudolph, Schloßer 83], a bibliography on parallel machines for symbolic computation is [Bibel, Aspetsberger 85]. A comparison of the L-machine concept with other parallel machine concepts is given, for example, in [Fessler, Paepcke, Schröter 81]. Systematic descriptions of parallel machine concepts are, for example [Paker 83] and [Hwang, Briggs 84]. A comparison between the ALICE machine and the L-machine is given in [Aspetsberger 85a], where it is shown how the ALICE concept could be realized by the L-components.

We will first sketch some easy parallel algorithms with the intention to demonstrate that very different interconnection topologies are necessary in order to exploit the parallelism in the various examples. Then we summarize explicitly the basic design objectives for an ideal parallel machine for symbolic computation that follow from the consideration of the examples. Next we describe the L-machine concept that attempts to meet these design objectives. Finally we give a programming example for the L-machine that should show how, in the L-language, both the description of processes and the (recursive) description of interconnection topologies is possible.

SOME EXAMPLES OF PARALLEL ALGORITHMS

In this section we demonstrate by some examples that very different interconnection topologies between processor modules can arise naturally in parallel (symbolic) algorithms. As a byproduct we see that good parallelizations, when compared with the corresponding sequential algorithms, are able to preserve the (time x (number of processor modules)) product. It seems to be hard to design parallelizations that work cheaper than that. We do not know whether a corresponding general theorem could be proven.

Merge Sort Algorithm

We start with the merge-sort algorithm formulated for a uni-processor.

```
sort(x) := if length of x = 1
           then x
           else merge ( sort (left-hand part of x),
                        sort (right-hand part of x) )
```

Here, x is a sequence of items. The complexity of this algorithm is O(n.log n), where n is the length of x.

While in the sequential algorithm the recursive calls for sorting the left-(right-)hand part of x are executed, in a straightforward parallel algorithm two sons could be started simultaneously. By repeating this process an O(n) parallel algorithm using O(n) processor modules arranged in a binary tree would arise. The space needed in this algorithm is O(n.log n). For details see [Aspetsberger 80]. In [Buchberger 78] it is shown how this algorithm can be improved such that only O(n) space is needed.

In [Todd 78] a parallel merge-sort algorithm using processor modules arranged in a pipeline is presented. We present the basic idea of this algorithm by giving an example. The input sequence, for example (4 3 1 9 5 3 2 4), is split arbitrarily into two subsequences, say, (4 1 5 2) and (3 9 3 4) and stored into the first processor mudule P1. P1 merges the sequences (4) and (3) etc. yielding the sequence (3 4) etc. and stores these sequences into processor module P2.

P1:

4	1	5	2
3	9	3	4

P2:

3 4	3 5
1 9	2 4

The processor module P2 merges all the sequences of length 2 to sequences of length 4 and stores them to processor module P3.

P2:

3 4	3 5
1 9	2 4

P3:

1 3 4 9
2 3 4 5

Finally, processor module P3 merges the two sequences of length 4 to a sequence of length 8, which is the sorted version of the input sequence.

P3:

1 3 4 9
2 3 4 5

P4:

1 2 3 3 4 4 5 9

All these processes can be overlapped. Roughly, the (i+1)-th processor module can start merging the first two sequences when the i-th processor module has composed them. The time complexity of this parallel merge-sort algorithm on pipelines is O(n). The number of processor modules is O(log n). Thus the product (time x number of processor modules) is O(n.log n), which is the same as in the uni-processor version.

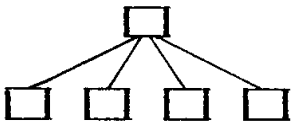
Parallelizations of Dijkstra's Single Source Shortest Path Algorithm

Below we present Dijkstra's $O(n^2)$ sequential algorithm for the single source shortest path problem, where n is the number of nodes in the graph, see [Aho, Hopcroft, Ullman 74]. At the right-hand side a condensed version of this algorithm is shown that reflects the relevant structure.

```
S := {v0}; D[v0] := 0;
for v ∈ V do                                  Initialization
    D[v] := l(v0,v)

while  S ≠ V  do                              while  S ≠ V  do
  w such that w ∈ V-S and                       minimum of n elements        *)
           D[w] is minimal;
  S := S ∪ {w};                                 S := S ∪ {w}
  for  v ∈ V-S  do                              for  v ∈ V-S  do
    D[v] := min(D[v], D[w]+l(w,v)).               minimum of two elements.   **)
```

A transformation of this algorithm into a parallel version could start in line **), where the calculation of the minima could be done in parallel using n processor modules. However, in line *) we have to compare O(n) numbers and, thus, all the modules have to be connected with one top module. This leads to a tree of depth 1.



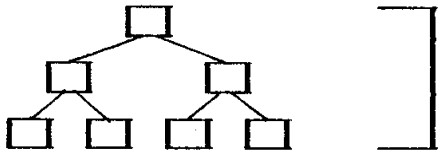
n processor modules

time complexity n

constant time complexity

On this tree line *) can be handled in time O(n). Since we have n iteration steps we again obtain time complexity $O(n^2)$ and a product (time complexity x number of processor modules) of $O(n^3)$. The high parallelism in line **) is lost by realizing line *).

However, the minimum of n numbers can also be computed in parallel using a binary tree of processor modules.



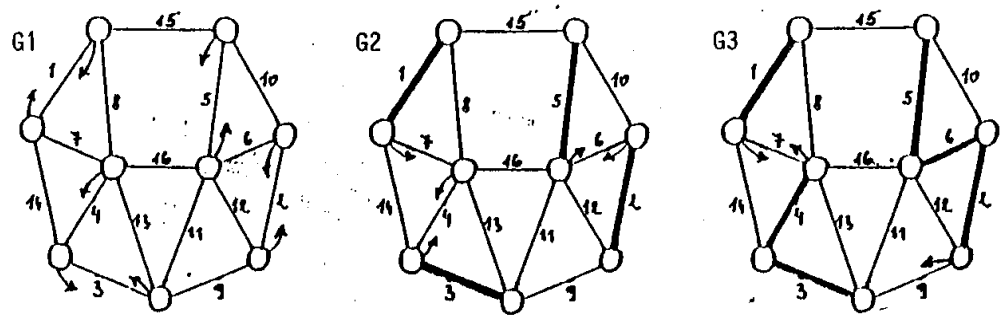
log n levels

n leaves

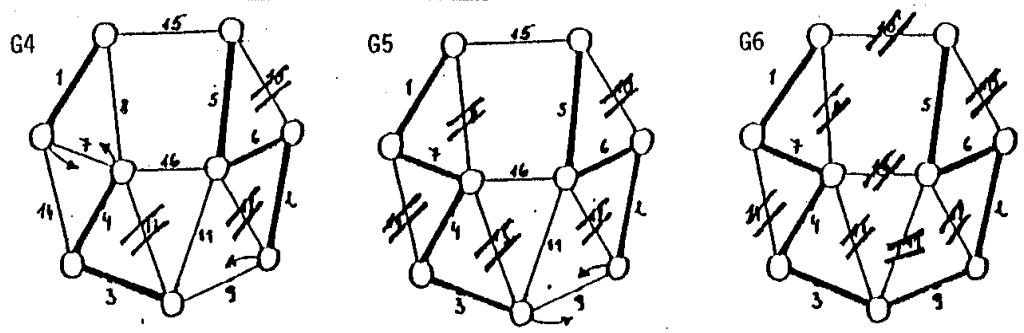
Every processor module of the tree compares the values of its two sons and sends the smaller value to its father. The time complexity of this process is O(log n). For n iteration steps this yields time complexity O(n.log n). The number of processor modules is O(n). Hence the product is now $O(n^2.\log n)$. This parallelization was proposed in [Lichtenberger 79]. Although the time complexity is much better than in the sequential algorithm the (time x number of processors) product is worse. In [Aspetsberger 84] a significant improvement is described that drastically reduces the number of processor modules without affecting the time complexity. The essential point of this method is that the processor modules at the leaves are each used for finding the minimum in line **) for log n vertices in O(log n) time. Thus, one needs only n/log n processor modules for computing all the minima in line **). This reduction of the number of processor modules does not affect the overall time complexity because, for line *) O(log n) many steps were needed anyway. The (time x number of processor modules) product is now O(n.log n) x O(n/log n) = $O(n^2)$. This is the same as it was in the case of the sequential algorithm.

Parallelization of Kruskal's algorithm for Minimum Cost Spanning Trees

Kruskal's algorithm finds a minimum cost spanning tree (MCST) for an undirected, connected graph G=(V,E) with an injective cost function, see for example [Aho, Hopcroft, Ullman 74]. In [Aspetsberger 83] a parallelization of this algorithm is given. The parallel algorithm starts in the same way as the sequential algorithm by constructing a spanning forest for G consisting of trees with only one vertex. In the subsequent steps, to each tree the edge of minimal cost leaving the tree is added. These processes can be done in parallel. (For a better synchronization, one only adds those edges that are of minimal cost for both trees connected by the respective edge. Eventually, such a situation will always be reached for all edges of the MCST). The algorithm stops, when all vertices in V are connected. In the following example we mark the edges of minimal cost leaving a tree by small arrows. G1 is the initial graph. The spanning forest consists of trees with only one vertex. G2 and G3 are the graphs after the first and second iteration step respectively.



In order to facilitate the determination of the edges of minimal cost leaving a tree all superfluous edges in the trees are deleted. G4 is identical to G3 except for the deletion of superfluous edges. G5 and G6 are the graphs after the third and forth iteration step respectively. G6 is the result of the algorithm.



In a suitable implementation each vertex should be represented by a processor module. The calculation for one iteration step is done by the vertex from which the edge e of minimal cost is leaving the repective tree. Afterwards it sends all necessary information to all vertices of the respective tree. Since the size and structure of the trees is changing after each step we would need a dynamically reconfigurable interconnection topology. Instead one may interconnect all processor modules with each other. The particular structure of the graph and the forest of spanning trees can then be represented by sets of edges and vertices. An analysis shows that this parallel algorithm has a (time x number of processor) product of $O(n^3)$, which is much worse than the $O(n^2)$ time complexity of Kruskal's sequential algorithm. [Bentley 80] has given a parallel version whose (time x number of processors) product is $O(n^2)$.

Prolog Execution (Automated Theorem Proving)

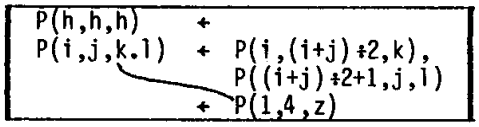
Consider the problem of computing the factorial of a natural number n. Let P(i,j,k) stand for "k is the product of all l, $i \leq l < j$". z = n!, then, can be computed by determining a value of z such that P(1,n,z). P(i,j,k) can be computed by the following PROLOG-like program.

```
P(h,h,h)     ←
P(i,j,k.l)   ←  P(i,(i+j)÷2,k), P((i+j)÷2+1,j,l)
```

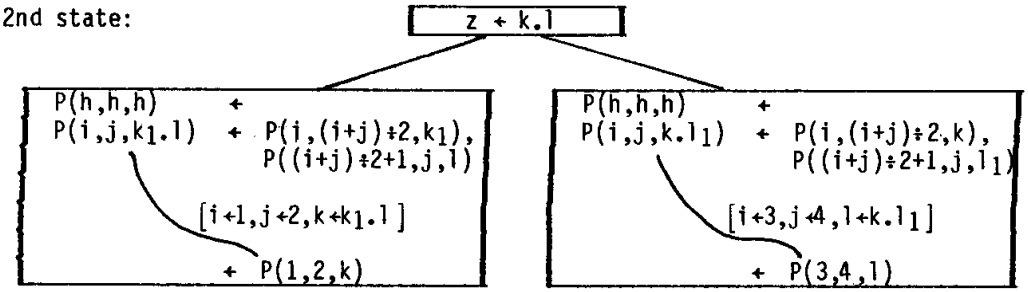
In the following we present a trace of a parallel computation of the factorial of 4 in order to show that the natural interconnection topology in this example is again a binary tree. In more general examples of resolution theorem proving one intermediate goal will be reduced to several subgoals (and-parallelism). Also there could be more than one literal complementary to the goal literal. The problem is solved if one can find a refutation for at least one of them (or-parallelism). For implementing a parallel procedure that meets all these aspects of parallelism in automated theorem proving one needs a dynamically reconfigurable interconnection topology.

In the example we depict pairs of unifiable literals on opposite sides of the implication symbol by arcs. The unifying substitutions are written in the form [i←1,j←4,z←k.1] etc. and are attached to the respective arcs. Each processor module, roughly, contains the whole set of clauses including the arcs connecting literals and the substitutions generated. In the presentation below, for some modules, we show only the (relevant) generated substitutions from which the final result can be composed. For more details see [Bibel, Buchberger 84] and [Aspetsberger 85b].

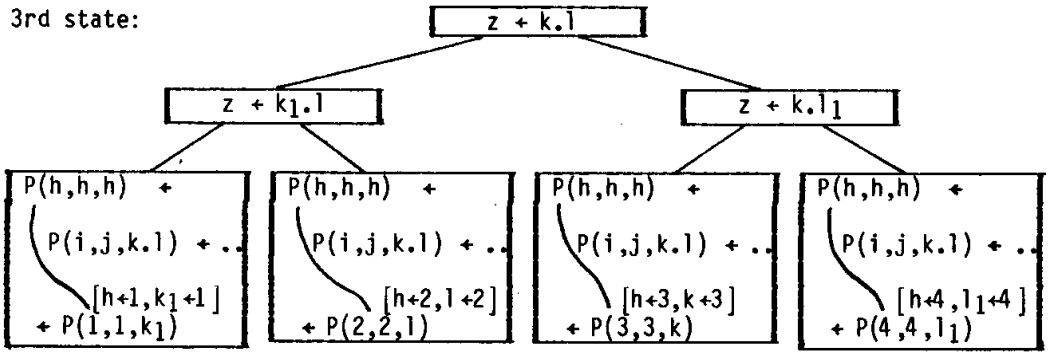
1st state:



2nd state:



3rd state:



All the processor-modules at the leaves of the tree deduce the empty clause. So they send back the interesting substitutions. We do not show the states occurring during sending back these substitutions and composing the final result in the top processor module. A complexity analysis of general theorem proving processes is difficult.

DESIGN OBJECTIVES FOR A PARALLEL MACHINE FOR SYMBOLIC COMPUTATION

We discuss the design objectives by listing pairs of alternative objectives and by explaining, for each of the pairs, which one of the two alternatives is the one we have chosen for the parallel L-machine.

Distributed (cellular) versus central control

In the above examples, typically, the partial processes are totally decoupled. A central control would be an artificial bottle neck. Hence, a suitable parallel machine concept must support a highly cellular control mechanism for big numbers of node processors.

Asynchronous versus synchronous cooperation of processes

In some of the examples, the processes in the different processor modules may take extremely different amounts of time. Thus, a suitable machine concept must allow an asynchronous cooperation of processors.

Flexible versus fixed interconnection topology
In the more demanding examples, the pattern of interconnections necessary for the communication between the partial processes depends on the given input. It may even vary during the execution of the parallel algorithm. Hence, the possibility of establishing a flexible interconnection topology in the parallel machine is desirable. This is in sharp contrast to the parallelization of numerical algorithms, where fixed array or vector topologies are appropriate.

Tight versus loose coupling
Typically, frequent communications between the processor modules are necessary. In addition, sometimes large quantities of highly structured data have to be exchanged in these communications. Hence, tightly coupled networks of processors are necessary, i.e. a processor should have the possibility to access the memories of its neighbors as easily as its own memory.

Homogeneous versus heterogeneous structure
In a given parallel algorithm, all the processor modules (or at least groups of processor modules) will perform the same task. Hence, all the processor modules must be of the same type, i.e. the whole structure will be homogeneous.

Flexible versus fixed synchronization
According to the necessity at the particular parallel algorithm, various types of synchronization between processors should be possible (data driven, result driven and other stimulations of processors). This means that the desired synchronization mechanism must be programmable (flexible) and not fixed by the hardware or the underlying operating system.

Many cheap versus few expensive processors
In the parallel algorithms considered, it is desirable that several hundred processors cooperate. Each of the processors will have to perform a relatively easy task with relatively little (a few K of) memory necessary for storing the relevant data. Hence, given a certain amount of money, a parallel machine concept is appropriate that uses the money for the integration of a huge number of cheap processor/memory modules in one system, with an emphasis on powerful communication facilities, rather than for the combination of a few powerful and expensive computers in a multiprocessor system. (A careful assessment of the trade-off of the two philosophies is one of the most important practical questions in the parallelization of symbolic computation, which has not yet been explored satisfactorily). Roughly, I think we must shift our interest from computation power to interconnection power.

Universal versus special purpose processor modules
The algorithmic subprocesses (for example, the unification of subformulas) that have to be realized in one processor module are complicated enough to use universal microprocessors as the core of the processor modules.

Universal versus special purpose parallel machine concept
If we want to have one parallel machine for a whole variety of parallel (symbolic) algorithms the machine as a whole must be universally programmable.

THE CONCEPT OF THE L-MACHINE

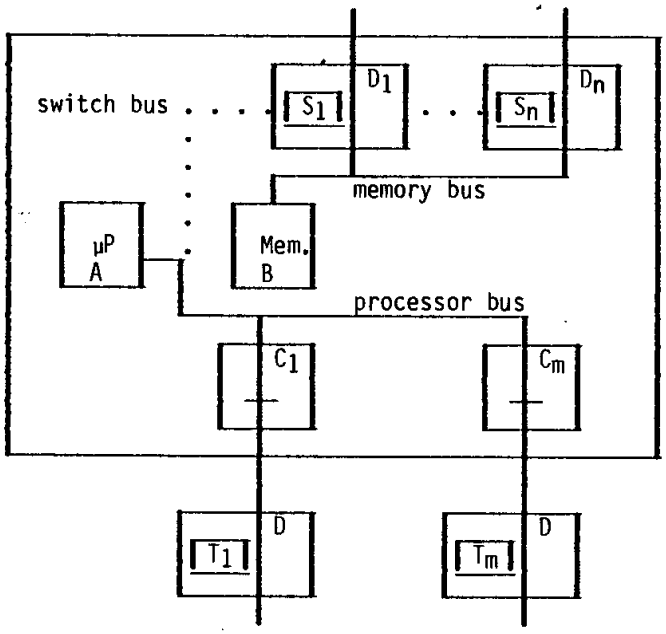
The concept of the L-machine is a parallel machine concept that attempts to meet the design objectives outlined above. The L-machine consists of universally programmable processor modules that can be interconnected flexibly in order to form cellular tightly coupled homogeneous nets of arbitary topology and size. The node processors cooperate asynchronously and all types of synchronization can be programmed flexibly. The concept has been introduced in [Buchberger 78] and has been modified to the now existing version in [Buchberger 83, 84]. At present, a prototype of the L-machine mainly in TTL technology consisting of 8 microprocessors is in operation at the University of Linz. By the extreme modularity of the concept, VLSI implementations seem to be easy and are planned for the near future. Mainly two ver-

sions of the machine are conceivable. One realizes the full graph interconnection topology. Its size will always be restricted by the quadratic increase in interconnection complexity. The other version realizes a particular (regular) interconnection topology of (nearly) arbitrary size. The present prototype realizes the full graph topology.

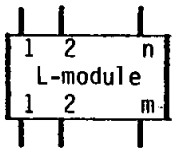
L-modules

The logical building elements of the L-machine are the so-called L-modules (see the figure below), which can be interconnected to form L-nets.

An L-module:



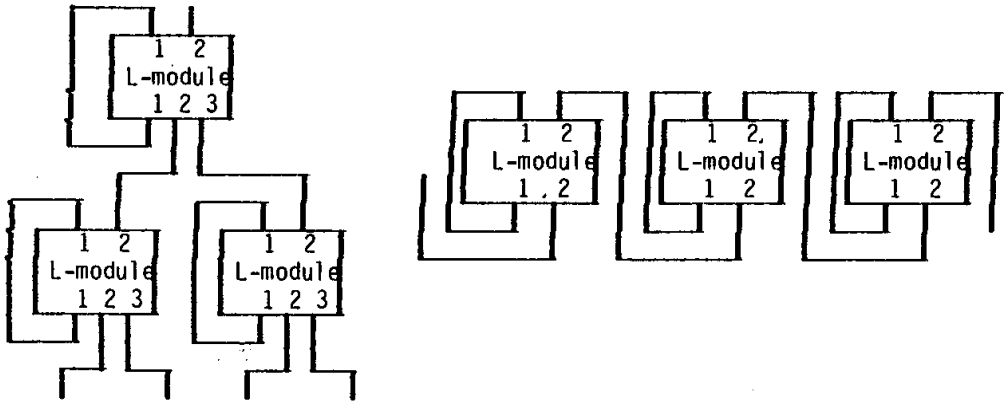
An L-module contains a microprocessor A with a (private) memory to store its program and intermediate data. Since the L-module contains a general-purpose processor, the usual universal instruction set is available. The second main component of an L-module is the shared memory B that can be accessed by other L-modules of an L-network via a bus. These L-modules can be connected via the switches $D_1,\ldots,D_n$. The microprocessor itself can access the shared memory of other L-modules via the switches $C_1,\ldots,C_m$ that can be opened and closed using the special instructions "open j" and "close j". For synchronization and the solution of access conflicts, each L-module has the possibility to use the "sensor bits" $S_1,\ldots,S_n$. These are special memory bits in the switches $D_1,\ldots,D_n$. The sensor bit S_j in the switch D_j can be accessed simultaneously by its own microprocessor and the microprocessor of the L-module connected to D_j. The special instructions "set local sensor j", "reset local sensor j" and "load local sensor j" are available for accessing the sensor bit S_j of the own L-module and the special instructions "set non-local sensor k", "reset non-local sensor k" and " load non-local sensor k" enable an access to the sensor bit of that L-module with which the L-module is linked via the swith C_k. In the figure above, $T_1,\ldots,T_m$ denote sensor bits in neighbouring L-modules. As a short-hand, we will use the following picture for an "L-module with m switches of type C and n switches of type D".



L-nets

Arbitrarily many L-modules can be combined to form "L-nets" of arbitrary (but fixed) regular (or irregular) structure, for example,

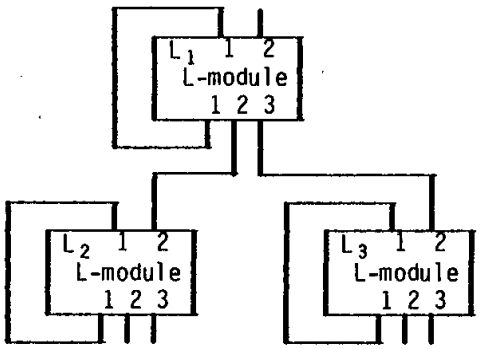
binary trees or pipelines



The numbers m and n must be chosen appropriately for the L-modules used for realizing a particular topology. (For example, m = 3, n = 2 for the binary tree and m = n = 2 for the pipeline.) In general, the neighbourhood parameters n and m will be small and so we can construct L-nets of every size and structure.

Data exchange (The use of the special "open/close" instructions)

Let us now have a closer look to a typical interconnection between L-modules in an L-net. Given the interconnections drawn in this figure, L_1 can store and load data to and from memory B of L_2 and memory B of L_3 and to its own memory B. L_2 and L_3 have only access to their own memories B.



For realizing a data exchange with L_3, L_1 has to execute the following instructions: open 1; p := b; close 1; open 3; a := p ; close 3. By this sequence, L_1 stores the content of cell b in its own memory B to cell a in memory B of L_3, where p is some variable in the private memory of L_1. We assume that all switches C initially are closed. The sequence above will be abbreviated by the notation

$$a^{(3)} := b^{(1)} .$$

Store instructions can be executed in parallel: for example L_1 could store the content of cell b of its own memory B simultaneously to cell a of memory B in L_2 and L_3. This can be realized by the following sequence of instructions: open 1; p := b; close 1; open 2; open 3; a := p; close 2; close 3.

Synchronization (The use of the sensor instructions)

Let us consider the situation where L_1 and L_2 execute the following instructions in parallel (f is some elementary operation):

L_1:	L_2:
$a^{(2)} := b^{(1)}$.	$a^{(1)} := f(a^{(1)})$.

Of course, in this situation an access conflict would arise for memory B of L_2. We can synchronize the L-modules L_1 and L_2 by forcing L_2 to wait until L_1 has finished transmission. For this purpose the sensor bits in the switches D are used. By appropriate use of the sensor instructions, all types of handshakes can be programmed. In the example at hand, the following sequences of sensor instructions could realize the desired synchronization:

	L_1:	L_2:
α:	load non-local sensor 2	set local sensor 2
	if accu = 0 then goto α	
	$a^{(2)} := b^{(1)}$	β: load local sensor 2
		if accu = 1 then goto β
	reset non-local sensor 2	$a^{(1)} := f(a^{(1)})$

For such sequences of instructions we will use a more convenient notation:

L_1:	L_2:
while not T_2 wait	$S_2 := 1$
$a^{(2)} := b^{(1)}$	while S_2 wait
$T_2 := 0$	$a^{(1)} := f(a^{(1)})$

It is crucial for the concept that the hardware realization of the sensor bits guarantees that the two processors connected to a given sensor bit have truely simultaneous access. The accesses must be triggered by the instruction cycles of the respective processors and must not be sequentialized or delayed by an arbiter in D. Sensor bits of this type are realized in the L-machine prototype.

The full graph L-machine

With the components above, L-nets of arbitrary but fixed topology can be realized. If one wants to go one step further, in principle, one could use a full graph L-net in order to be able to realize every other interconnection topology solely by software means. However, this would need n^2 switches of type D and D and n^2 interconnection buses. In [Buchberger 83] we have shown how, by a geometrical transformation, one can realize the full graph L-net using only 2n interconnection buses without affecting the logical and computational power of the L-net. No new components are needed for the realization of this machine, which we call the (full graph) L-machine. Due to space limitations, we cannot give any details of this geometrical transformation in the present paper.

The L-language

An "L-program" written in the "L-language" consists of the description of the programs residing in the L-modules of an L-net and the description of the topology of the L-net. Crucial extensions to ordinary programming languages are necessary. The description of an L-net by an L-program can either be conceived as a description of the interconnections necessary for the realization of that particular L-net. Or it can be conceived as the description of a computationally and logically equivalent L-

net to be realized on the full graph L-machine by software means. In the latter case the description of the L-net by the L-program can be automatically compiled to a particular embedding of the L-net in the full graph. A compiler of this type is currently realized, see [Hintenaus, Buchberger 85].

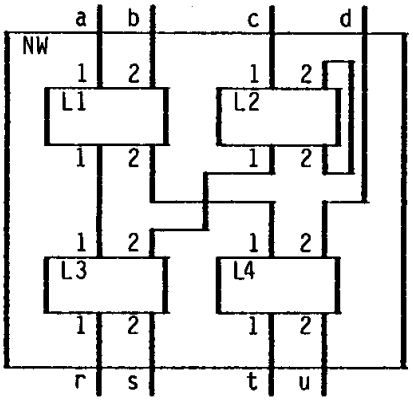
The nucleus of the L-language is an ordinary high-level language. Programs written in the nucleus language are meant to reside in the private memory of an L-module and are executed under the control of the processor in the module. A variable v denotes a storage region in all those shared memories that are "attached" to A via the "open" switches of type C. The first extension to the nucleus is the possibility of declaring private variables. A variable declared private denotes a storage region in the private memory of a component A. The next extension is the addition of the special instructions of the eight types

"open j", "close j"
"set, (reset, load) local sensor j"
"set (reset, read) non-local sensor j".

Also, one must have symbolic names for the paths available in a particular L-program. L-programs residing in one L-module, thus, have the following typical structure:

```
net L1: elementary
        [private p, q;
        ....
        some program text using, for example,
           close 1; p:= a; open 2; if S2 = 1 then ....;
        ...]
        processor paths 1, 2;
        memory paths 1, 2.
```

The crucial additional feature of the L-language is the possibility to describe net structures. It should be immediate, what is meant by the following graphical notation of an L-program NW:

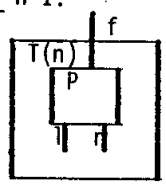


In syntactically more conventional "linear" notation the L-program NW reads:

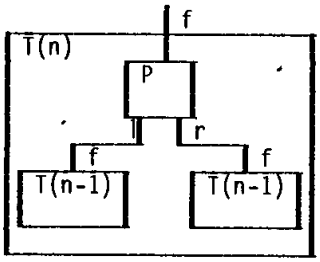
```
net L1: .... ; net L2: .... ; net L3: .... ; net L4: ....  :
net NW: compound
     [L1; L2; L3; L4 ]
      processor paths
      r:= 1 of L3; s:= 2 of L3; t:= 1 of L4; u:= 2 of L4
      memory paths
      a:= 1 of L1; b:= 2 of L1; c:= 1 of L2; d:= 2 of L4
      connections
      1 of L1 with 1 of L3; 2 of L1 with 1 of L4;
      1 of L2 with 2 of L3; 2 of L2 with 2 of L2.
```

Typically, L-nets of highly regular structures are used for the parallelizations of algorithms and, in addition, the "size" of these regular structures depends on the value of the inputs for the algorithm. In order to define such variable size L-programs the names of L-programs must allow "parameters". As an example, consider the definition of an L-net with tree structure. In semigraphical notation the definition is:

net T(n): compound
case n=1:



case n>1:



Note that recursive definitions of L-nets are possible in the L-language. This is a feature that was not available in any other programming language. In the example, P is assumed to be some (elementary) L-network with paths l, r, f. In linear notation the same definition is:

```
net T(n): compound
    case n=1:  [P]
               paths f:= f of P;
    case n>1:  [ P; 2 copies of T(n-1) ]
               paths f := f of P;
               connections
               l of P with f of copy 1 of T(n-1);
               r of P with f of copy 2 of T(n-1).
```

PORGAMMMING EXAMPLE

As an easy programming example we present a parallel procedure by K. Aspetsberger, see [Aspetsberger, Bayerl 85], realizing the connection method (see [Bibel 83]) for testing the validity of a formula in propositional logic. We assume that the formula to be proved has been transformed by a preprocessing procedure into a matrix containing only literals as elements. For instance, if we want to establish the validity of the formula

$$((A \vee B) \wedge (A \rightarrow C) \wedge (B \rightarrow D)) \rightarrow (C \vee D)$$

it should first be transformed to the following matrix

$$\begin{array}{cccc} \neg A & A & B & \\ & & C & D \\ \neg B & \neg C & \neg D & \end{array}$$

A formula is valid, if every possible path through the corresponding matrix contains at least one pair of complementary literals. In our example we can find the following eight sequences of literals indicating the possible paths through the matrix. Each sequence contains a pair of complementary literals and so the formula is valid.

$\neg A\ A\ B\ C\ D$ $\quad \neg A\ A\ \neg D\ C\ D$ $\quad \neg A\ \neg C\ B\ C\ D$ $\quad \neg A\ \neg C\ \neg D\ C\ D$

$\neg B\ A\ B\ C\ D$ $\quad \neg B\ A\ \neg D\ C\ D$ $\quad \neg B\ \neg C\ B\ C\ D$ $\quad \neg B\ \neg C\ \neg D\ C\ D$

In the parallel procedure one tries to find all the paths through the matrix

without pairs of complementary literals. If no such path can be found, the formula is proved to be valid. Otherwise the procedure will produce all counterexamples. More specifically, one successively generates sequences without pairs of complementary literals of length 1, 2,... until one eventually obtains sequences of length n, where n is the number of clauses in the formula. The generation of these sequences can be done in parallel. First, the clauses of the matrix are stored in different L-modules. Each of the L-modules then generates successively sequences of length 1, 2, ... starting with the literals of the respective clause. After each step the L-module transfers the set of paths to the left neighbor, which makes use of this information when generating the set of paths of the next higher length. Finally, the set of paths of length n is obtained in the first (left-most) L-module.

Before giving more details we trace the procedure when testing the validity of the formula in the above example. For initialization we load the set of literals of each clause to the respective module. These are the paths of length 1.

L-module 1	L-module 2	L-module 3	L-module 4	L-module 5
{ ¬A, ¬B }	{ A, ¬C }	{ B, ¬D }	{ C }	{ D }

In the fist step sequences of length 2 are generated by combining the paths in each L-module with the paths of the respective right neighbor. In every step we are checking whether the new path contains complementary literals. In this case we delete the respective path.

{ ¬A ¬C, ¬B A, ¬B ¬C }	{ A B, A ¬D, ¬C B, ¬C ¬D }	{ B C, ¬D C }	{ C D }

In the second step sequences of length 3 are generated by combining the paths in each L-module with the paths of the right neighbor. Now one has to check additionaly whether the paths are "combinable", i.e. whether the last literal of the first path is equal to the first literal of the second path. Note that, for testing that a combined path is free of complementary literals, we only have to check whether the first literal of the first sequence is complementary to the last literal of the second sequence.

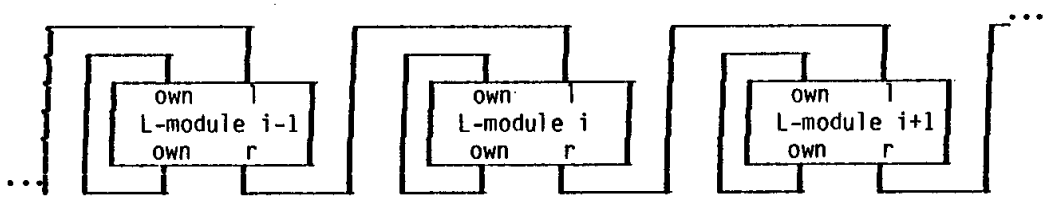
{ ¬A ¬C B, ¬A ¬C ¬D, ¬B A ¬D, ¬B ¬C ¬D }	{ A B C, A ¬D C }	{ B C D }

In the third step the sequences of length 4 are generated. Now two paths are combinable, if the last two literals of the first sequence are equal to the first two literals of the second sequence.

{ ¬B A ¬D C } { A B C D }

Finally in the fourth step the first L-module recognizes that the two sequences are not combinable. Thus the new set of paths is empty, i.e. the formula is valid.

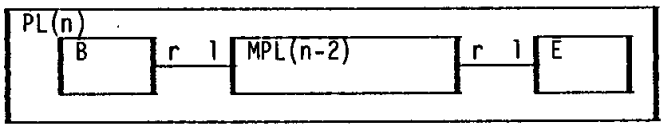
For implementing this parallel procedure on the L-machine we form a pipeline of L-modules:



We now give a recursive description of the L-program PL ("pipeline") with n L-modules using the L-language. We present the definition "top-down" starting with the net description.

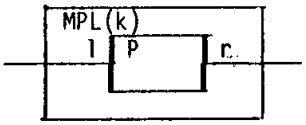
net PL(n): compound

case n>2:

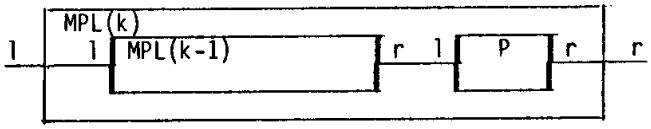


net MPL(k): compound

case k=1:



case k>1:



B, P and E are elementary L-programs. For simplicity we did not draw the connection from "own" to "own" of the same L-module. It should be clear how the the same definition could be given in linear notation. Now we describe the elementary L-programs P. Comments are given in brackets () behind the instructions.

net P: elementary

```
private   n    (number of all clauses in the formula)
          i    (number of the L-module in the pipeline)
          PR   (set of paths produced by the right neighbor in the previous step)
          PH   (intermediate set of paths for the new step)

for j := 1 to n - i do
    (in the j-th iteration step the i-th L-module creates all the sequences
     of literals of length j+1 starting from clause Ci and containing no
     complementary literals)

    Sl := 1        (we signalize to the left neighbor that it may access our
                    shared memory)

    while Tr = 0 wait     (we wait until our right neighbor has finished the
                           previous iteration step and we may access its shared
                           memory for fetching its set of paths from the
    PR := P(r)             previous step)

    Tr := 0    (we signalize to the right neighbor the end of path transfer)

    while Sl = 1 wait     (we wait until the left neighbor has terminated
                           transmission)
    PH := {}
                                  (we create the new set of paths of length j+1
    for all p ∈ P(own)  do         combining the paths of length j starting from
                                   clause i with the paths of length j starting
        for all q ∈ PR do          from clause i+1)

            if combinable(p,q,j) and pl is not complementary to qj

               then  PH := PH ∪ {combine(p,q,j)}

    P(own) := PH
```

```
S1 := 1      (we signalize to the left neighbor that we have terminated the
              iteration step)

processor paths     own, r;
memory paths        own, l.
```

The meaning of the predicate "combinable(p,q,j)" and the function "combine(p,q,j)" should be clear from the consideration of the above example.

The program B in the first L-module of the pipeline is slightly different from P. Since the first L-module has no left neighbor all instructions concerning S_l are deleted. Additionally, upon termination of the program, the first L-module sends one of the two messages "The formula is valid" or "The formula is not valid. Counter examples: ...." depending on whether P in its shared memory is empty.

```
net B: elementary

    private   n    (number of all clauses in the formula)
              PR   (set of paths produced by the right neighbor in the previous step)
              PH   (intermediate set of paths for the new step)

    ...
    (same program as in P except that the sensor instructions concerning the left
     neighbor are omitted)
    ...

    processor paths     own, r;
    memory paths        own.
```

The program E in the last L-module of the pipeline is a special case of P. Since in this case i is n, the for-loop can be skipped. Thus, we get a very easy program:

```
net E: elementary
    S1 := 1
    memory paths        1.
```

ACKNOWLEDGEMENT: This project is supported by a grant from SIEMENS München (Dr. H. Schwärtzel). Special thanks to K. Aspetsberger for invaluable help in the preparation of this paper.

REFERENCES

Aho A.V., Hopcroft J.E., Ullman J.D., 1974.
The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company, Reading, Massachusetts.

Aspetsberger K., 1980.
Algorithmentypen für Multi-Mikrocomputer-Systeme. Diplomarbeit, Inst. f. Math., University of Linz.

Aspetsberger K., 1983.
Parallel Algorithm for the Minimum Cost Spanning Tree Problem on L-Networks. Tech. Rep. CAMP-Publ.-Nr.: 83-12.0, Oct. 1983.

Aspetsberger K., 1984.
Some Examples of Parallel Algorithms for L-Networks. MIMI 84, Bari, Acta Press, Anaheim, p. 182-185.

Aspetsberger K., 1985a.
Towards Parallel Machines for Artificial Intelligence: Realization of the ALICE Architecure by the L-Components. Proceedings of the Austrian Workshop on Artificial

Intelligence, Vienna, Sept. 24-27, 1985. Informatik-Fachberichte-KI, Springer Verlag.

Aspetsberger K., 1985b.
Substitution Expressions: Extracting Solutions of Non-Horn Clause Proofs. Proceedings of EUROCAL 85, Linz, April 1-3, 1985. Lecture Notes in Comp. Scie., Springer Verlag, Heidelberg.

Aspetsberger K., Bayerl S., 1985.
Two Parallel Versions of the Connection Method for Propositional Logic on the L-Machine. Proceedings of the German Workshop on Artificial Intelligence, Dassel/ Solling, Sept. 23-28, 1985. Informatik-Fachberichte, Springer Verlag.

Bentley J.L., 1980.
A Parallel Algorithm for Construction Minimum Spanning Trees. Journal of Algorithms 1, p. 51-59.

Bernutat-Buchmann U., Rudolph D., Schloßer K.-H., 1983.
Parallel Computing I, A Bibliography. Bochumer Schriften zur Parallelen Datenverarbeitung. Ruhr Universität Bochum.

Bibel W., 1983.
Matings in Matrices. CACM 26, pp. 844-852.

Bibel W., Aspetsberger K., 1985.
A Bibliography on Parallel Inference Machines. J. Symbolic Computation (1985), 1/1, p. 115-118, Academic Press Inc., London.

Bibel W., Buchberger B., 1984.
Towards a Connection Machine for Logical Inference. Int. Symb. on Fifth Generation and Super Computers, Rotterdam, Dec. 11-13, 1984. Appeared in: Future Generations Computer Systems, 1/3, p. 177-188, North Holland Publishing Company.

Buchberger B., 1978.
Computer Trees and their Programming. 4th Coll. "Trees in algebra and programming", Univ. Lille, Feb. 16-18, p. 1-18.

Buchberger B., 1983.
Components for Restructurable Multi-Microprocessor Systems of Arbitrary Topology. MIMI 83, Lugano, Acta Press, Anaheim, p. 67-71.

Buchberger B., 1984.
The Present State of the L-Network Project. MIMI 84, Bari, Acta Press, Anaheim, p. 178-181.

Fessler G., Paepcke A., Schröter N., 1981.
Parallelrechner: Probleme und Strukturen. Elektronische Rechenanlagen, 23/5, p. 211-220.

Hintenaus P., Buchberger B., 1985.
The L-Language for the Parallel L-Machine (A Parallel Architecture for AI Applications). Proceedings of the Austrian Workshop on Artificial Intelligence, Vienna, Sept. 24-27, 1985. Informatik-Fachberichte-KI, Springer Verlag.

Hwang K., Briggs F.A., 1984.
Computer Architecture and Parallel Processing. McGraw-Hill Book Company, New York.

Lichtenberger F., 1979.
Speeding up Algorithms on Graphs by Using Computer Trees. In: Graphs, Data Structures, Algorithms (Nagl M., Schneider H.-J., eds.). Appl. Comp. Scie. 13, Hanser Verlag, p. 66-79.

Paker Y., 1983.
Multi-microprocessor Systems. Apic Studies in Data Processing No. 18. Academic Press Inc., London.

Todd S., 1978.
Algorithm and Hardware for a Merge Sort Using Multiple Processors. IBM J. Res. Develop., 22/5, Sept. 1978.