Towards a Connection Machine for Logical Inference*

W. BIBEL
Institut für Informatik, Technische Universität München, Postfach 20 24 20, D-8000 Munich, F.R. Germany

and

B. BUCHBERGER
Johannes Kepler Universität Linz, Linz, Austria

This paper gives an outline of the architecture of a multiprocessor machine for deduction in first-order logic and its functional behavior, which fully exploits the parallelism inherent in the deductive process. With respect to the inferential side, the approach is based on the connection method which lends itself to a parallel treatment as illustrated with several examples. With respect to the architectural side, it is argued that in comparison with other well-known parallel architectures the L-networks appear to be most suitable for this kind of problem.

*This work was supported in part by the EC and by Nixdorf Computer AG within the project "Parallel architectures and languages for A.I.P. – A VLSI-directed approach" in ESPRIT. It is based on work supported by the Deutsche Forschungsgemeinschaft (Bi228/3-3) and the Oesterreichische Fond zur Foerderung der wissenschaftlichen Forschung (Projekt 3896).

Introduction

The Japanese Fifth Generation Project has drawn the attention of the international community in information technology to PROLOG as a flexible tool for many kinds of applications. As a language, PROLOG inherits its flexibility from first-order logic of which it is a restricted sublanguage. As a tool it consists mainly of a restricted automated theorem prover for first-order logic which by these restrictions turns out to be reasonably efficient in many applications.
From a logical point of view, however, PROLOG still suffers from serious drawbacks. One consists in its restriction to Horn clauses; rather one would like to fully exploit the expressiveness of first (or even higher) order logic, since this restriction is already felt in simple examples.
Another problem arises from the fact that the efficiency in PROLOG heavily depends upon the sequence in which the clauses are presented by the programmer. This clearly contradicts the logic (or predicative) programming philosophy [1], under which the programmer is supposed to concentrate on presenting his problem correctly rather than to worry about the details of machine execution.
Qualitatively, these (and other) problems cannot be overcome simply by making machines faster. Rather a substantial improvement requires progress in the following two directions. First, the theorem proving techniques currently used in PROLOG systems do by far not take full advantage of the information provided by the syntactic structure of a given formula or program. For instance, in many cases a PROLOG compiler could easily rearrange the sequence of clauses towards a more efficient one rather than expecting this task to be taken care of by the programmer. The connection method in theorem proving [2, 3] is expected to further support progress into this direction.
The second possibility of qualitative improvement lies in parallel computation. Rather than exploring different paths in the search tree for a proof sequentially with a single processor, one would consider in this approach each possible path with a separate processor in a completely parallel way. This is the line we pursue in the present paper.
The global view of theorem proving provided by the connection method also facilitates the task of a parallel treatment of separable pieces of the whole proof search. Therefore we are more ambitious than just exploiting AND-/OR-parallelism in a top-down proof search. Our goal is rather, in addition and in parallel, to take advantage of a bottom-up develop-

ment and of global control considerations. This in turn requires a powerful and flexible organization of the underlying multiprocessor system.
In the sections 2 and 3 we argue that the L-networks [5,6] provide this power and flexibility. Namely, an L-network is a universal programmable parallel machine consisting of a potentially huge number of universal (cheap) node processors that can be interconnected flexibly in order to form cellular, tightly coupled networks of arbitrary topology. All these and other properties make it ideally suited for such an involved task like theorem proving.
In the sections 4 and 5 we give a first, rough outline of a parallel proof procedure based on the connection method, to be realized on an L-network. This outline is at the same time meant as the starting point of a larger project to be carried out in the coming years in the framework of ESPRIT. Its ultimate goal is a VLSI implementation of a logical multi-processor connection machine realizing such a parallel proof procedure. As an aside we mention that on the way towards this goal the language FP2 developed by Jorrand [10] will independently be used.
The paper begins with an illustration of the connection method applied to a simple example.

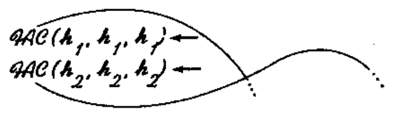
1. The Connection Method

Resolution clearly is the most widely used proof method in the field of Automated Theorem Proving (ATP). In the form of PROLOG systems it became even popular in computer (and other) sciences as an attractive tool for programming. Upon closer inspection of the more advanced work in ATP it may be seen, however, that the interest has actually shifted towards higher level proof methods. This is due to the need for a more global control of the proof search.
The connection method is of such a high-level nature as pointed out in [3]. Although we cannot give an introduction here, of course, it is the purpose of the present section to provide the reader with a feel for its flavor, while introducing material for the later sections. Ref. [2] is a comprehensive source for any details.

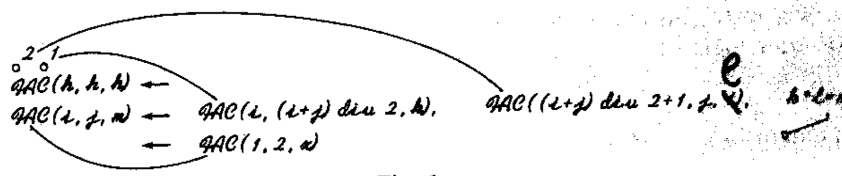
Consider the following PROLOG program for computing the factorial of n:

$$\begin{aligned}
&\text{FAC}\ (h,h,h) \leftarrow \\
&\text{FAC}\ (i,j,m) \leftarrow \text{FAC}\ (i,\ (i+j)\text{div}\ 2,\ k), \\
&\qquad\qquad\quad \text{FAC}\ ((i+j)\text{div}\ 2+1,j,l),\ k*l=m \\
&\qquad\qquad \leftarrow \text{FAC}\ (1,n,x)
\end{aligned}$$

A proof or computation for $n=2$ in terms of the connection method is illustrated in Fig. 1. It depicts 3 arcs connecting pairs of literals on opposite sides of the implication symbols as well as an open-ended arc. Two of the connected nodes are labeled with integers, viz. 1 and 2. This encodes the fact that the first clause is actually to be taken in two copies with different variables, viz. h_1 and h_2. So the first line encodes the following two lines.



A PROLOG program together with such kind of connections qualifies as a connection proof if it satisfies the following two requirements.
(i) In any selection of exactly one literal from each clause at least two of them are connected or one is connected in the open-ended way, that is, each path contains a connection (i.e., the set of connections is spanning), as we say in technical terms.
(ii) There is a substitution of variables by terms, such that connected pairs of literals are identical (by unification) or the connected literal is an identity, both modulo the laws of arithmetic.
The reader is encouraged to note that each of the four possible selections in the sense of (i) in our present example actually enjoys that property. Let us calculate a substitution so that property (ii) holds as well. The connection ending in the goal clause apparently yields the partial substitution $[i \leftarrow 1, j \leftarrow 2, x \leftarrow m]$. Since $(1+2)$ div $2=1$, (ii) is satisfied for the remaining two connections if in addition we have $[k \leftarrow 1, h_1 \leftarrow 1]$ and $[l \leftarrow 2, h_2 \leftarrow 2]$. Since $1*2=2$ the equality literal finally yields $[m \leftarrow 2]$. Combining these two



2 1
FAC(h, h, h) ←
FAC(i, j, m) ← FAC(i, (i+j) div 2, k), FAC((i+j) div 2+1, j, l), k*l=m
← FAC(1, 2, x)

Fig. 1

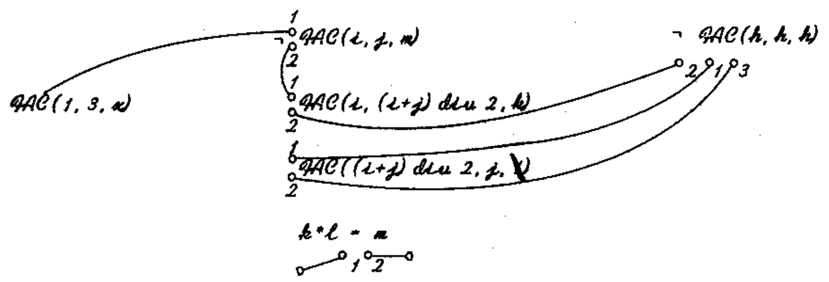


Fig. 2

results into one substitution leads to the expected value 2 for the output variable *x*.
The conditions (i) and (ii) in this or in similar form characterize connection proofs not only for PROLOG programs, i.e. Horn clauses, but also for arbitrary theorems in first-order logic. In contrast to PROLOG, which is based on a refutation procedure, the connection method is usually introduced as an affirmation procedure. That is why then the clauses are presented vertically, so that after elimination of the implication sign the previous program usually has the representation of Fig. 2.
In general, this representation is obtained from the PROLOG representation by a 90° turn and the replacement of the implication signs by negation signs in the topmost row. In this new version the example is shown with a connection proof for input 3 (rather than 2) which in the same way as before results in the value 6 for *x*. Note that here the second clause is taken into account with two different copies and the rightmost clause with three copies.
In the last decade sophisticated procedures have been developed that eventually detect this kind of connection proofs for a given theorem in a sequential way. In principle, they test unifiability for spanning sets of connections in an involved way. This task lends itself to a parallel approach. In particular, different sets of connections may be explored in parallel which corresponds to the usual OR-parallelism; but also the different connections in each set may be processed in parallel, which in turn corresponds to the usual AND-parallelism. In other words, all potential connections may be processed in parallel, a goal which to some extent we eventually want to achieve.
It has been shown by Eder [7] that the unificational part of the proof search may smoothly be adapted to such an extreme parallel approach. Namely, for the resultant substitution it does not matter at all in which sequence it is built together from the various partial substitutions.

It is also clear from the well-explored sequential approach how the different processors have to be controlled in principle so that eventually a connection proof is generated by their combined efforts. For two reasons this task is by far not trivial, however. First of all, the whole process in its full generality is extremely complex; there are many important phenomena of detail which do not become apparent in the previous simple example. This in turn requires a rather flexible machine architecture that at the same time fully exploits the advantages of parallelism in a hopefully efficient way.
The L-systems, to be introduced in the third section, provide the appropriate tools for that purpose. For their full appreciation let us first turn our attention to the various approaches of parallelism.

2. Design Objectives for a Parallel Logical Connection Machine

In the past ten years, a great number of different parallel machine structures has been proposed. (For an overview on the literature see [9, 11]. Different machine structures are the result of different design objectives. Below we analyze the design objectives of a machine structure suitable for a parallel realization of the connection method. A machine concept that meets all these design objectives is then described in the next section. 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 to be chosen for a parallel realization of the connection method.

2.1. Distributed (cellular) versus central control

In our parallelization of the connection method, the various (instances of) connections should be treated in parallel. For each processor working on one of the connections it should be possible to locally stimulate new neighbors for acting on derived connections and to exchange information with these neighbors without interference of a central control. Hence, a suitable parallel machine concept must support a highly *cellular* control mechanism for big numbers of node processors.

2.2. *Asynchronous versus synchronous cooperation of processes*

The treatment of different connections in different processors, typically, will take extremely different amounts of time, because the subformulas to be treated may have different size and structure. Thus, a suitable machine concept must allow an *asynchronous* cooperation of processors.

2.3. *Flexible versus fixed interconnection topology*

In a parallel realization of the connection method, a processor will have to communicate with various neighbors depending on the subformula treated. The pattern of interconnections necessary for these communications will vary with the particular input to the connection method. It even may 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.

2.4. *Tight versus loose coupling*

Typically, frequent communications between processors will be necessary for working in parallel on various parts of a formula in the connection method. In addition, 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.

2.5. *Homogeneous versus heterogeneous structure*

Every processor in the parallel hardware structure must be able to realize every subalgorithm in the parallel connection graph procedure depending on the particular example at hand. Hence, all the processor modules must be of the same type, i.e. the whole structure will be *homogeneous*.

2.6. *Flexible versus fixed synchronization*

According to the necessity at the particular stage of the 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.

2.7. *Many cheap versus few expensive processors*

In the parallel treatment of a formula, several hundred processors may 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, for a parallelization of the connection method 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 computational logic, which has not yet been explored satisfactorily).

2.8. *Universal versus special purpose processor modules*

The algorithmic subprocessors (for example, the unification of subformulas, the creation of new instances etc.) that have to be realized in the single processor nodes of a parallel machine for the connection method are complicated enough to use *universal* microprocessors as the core of the processor/memory modules. Also, at least in this stage of research, one wishes to make experiments with very different variants of the method. This is yet another reason why the node processors must be universally programmable. In a later stage of the project, when more experience on parallel variants of the connection method has accumulated, maybe, a proposal for an appropriate special purpose (universally programmable) node processor will be made.

2.9. *Universal versus special purpose parallel machine concept*

In this paper, the parallelization of the connection method is our main objective. However, the whole field of symbolic computation (computation with symbolic objects like terms, formulae, programs, algebraic objects) provides many examples of problems and algorithms whose parallelization gives rise to design objectives similar to those discussed above. Hence, it is reasonable to think about parallel machine concepts that can be *universally* used for a wide class of parallel algorithms in symbolic computation.

3. L-Networks: A Parallel Machine Concept for Symbolic Computation

The concept of "L-networks" is a parallel machine concept that meets all the design objectives discussed in the last section, i. e. an L-network is a universally programmable parallel machine consisting of a large number of universal (cheap) node processors that can be interconnected flexibly in order to form cellular tightly coupled homogeneous networks of arbitrary topology. The processes in the node processors cooperate asynchronously. All types of synchronization can be programmed flexibly.
The concept of L-networks has been introduced in [5] based on an early forerunner described in [4]. L-networks are built from *four "basic components"* A, B, C, D. A is a processor component, B a memory component, C and D are certain bus switches. From these components larger building blocks called *"L-modules"* can be composed. L-modules can then be used as the processing nodes in *"L-networks"* of arbitrary but fixed topology. A particularly powerful L-network is the *"full L-network"*, in which every node (L-module) is connected with every other node (i. e. the topology realized is the full graph). Every other L-network can be flexible embedded into the full L-network. For n nodes, the full L-network would need n^2 interconnection buses and n^2 components of types C and D. However we have shown in [5] that, by a geometrical transformation that leaves the logical and computational power of the full L-network untouched, one can realize the full L-network using only $2.n$ interconnection buses and n^2 components of types C and D. The resulting hardware structure resembles a crossbar switch. However, it has some extraordinary features. The most salient of them are that the full L-network has no central control and that the whole structure perfectly lends itself to a VLSI implementation. Full L-networks with several hundred nodes seem to be realistic with present VLSI technology.
In this paper we can only give a rough outline of L-networks, in particular full L-networks. For more details the reader is referred to [4–6] and the references given there.

3.1. *The Four Basic Components*

The four basic components are:
A: a microprocessor + private memory + some additional special circuitry.
B: a "shared" memory + some additional special circuitry.
C: a bus switch with an additional "open/close" facility.
D: a bus switch with a "sensor bit".
The processor component A has a private memory, which will be used mainly for storing the program and, in certain occasions, some intermediate data. The additional special circuitry allows the processor to execute certain special instructions in addition to the standard instruction set. These special instructions are of the following eight types:
"open j"
"close j"
"set local sensor j"
"reset local sensor j"
"load local sensor j"
"set non-local sensor j"
"reset non-local sensor j"
"load non-local sensor j"

The meaning of these special instructions will be explained later.

The memory component B is an ordinary RAM with some additional circuitry.
A switch of type C has the structure depicted in Fig. 3.

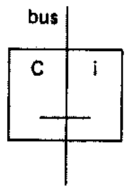


Fig. 3

Here, i is a hardware address of the switch. On the bus the following types of information can arrive:
store and load instructions
"open j"
"close j"
"set non-local sensor j"
"reset non-local sensor j"
"load non-local sensor j".
The horizontal bar symbolizes that the bus can be opened and closed (by the "open/close" instruction).

A switch of type D has the structure given in Fig. 4. Again, *i* is hardware address of the switch. On bus_1 the following types of information can arrive:
store and load instructions
"set non-local sensor"
"reset non-local sensor"
"load non-local sensor"

On bus_2 the following types of information can arrive:
"set local sensor *j*"
"reset local sensor *j*"
"load local sensor *j*".

S is a one bit memory cell that can be accessed both from bus_1 and bus_2. We call this cell a "sensor bit". The operation of switches C and D will be explained after the concept of an L-module will have been introduced.

3.2. L-Modules

A microprocessor component of type A, a memory component of type B, *m* switches of type C and *n* switches of type D can be combined, for example, as shown in Fig. 5. An arrangement of this type is called an *L-module* with *m* "processor paths" and *n* "memory paths". Functionally, it is equivalent to the concept of an L-module introduced in [4].

3.3. L-networks

Arbitrarily many L-modules can be combined for forming "L-networks" of arbitrary (but fixed) regular or irregular structure by interconnecting buses leaving switchs of type C with the buses entering switchs of type D in a different or the same L-module.

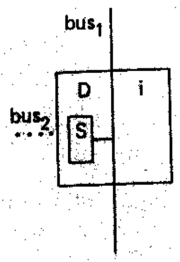


Fig. 4

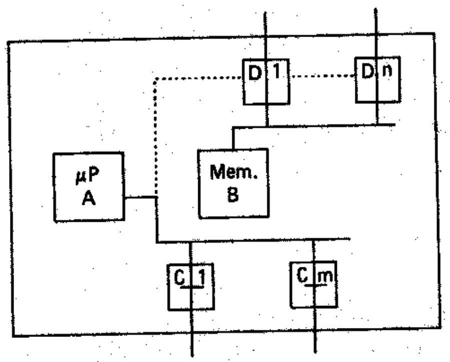


Fig. 5

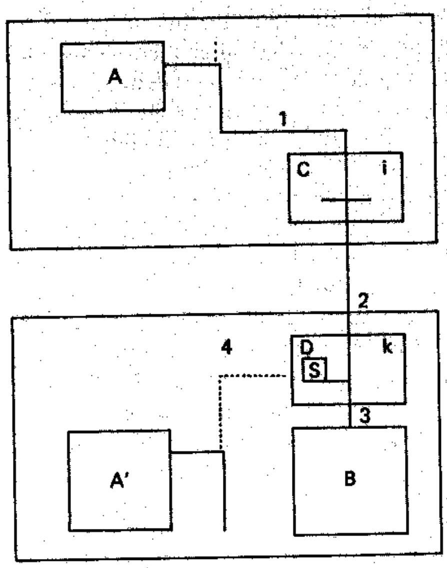


Fig. 6

3.4. Components: Details of Function

For explaining the functional characteristics of the four basic components A, B, C, D, we look at a typical interconnection between two L-modules in an L-

network (see Fig. 6). Switch C with hardware address i analyzes an instruction arriving from A on bus 1 (the "processor bus"). If the information is a store or load instruction, (consisting of operation code, address and data), and the switch is in the position "close", the instruction passes switch C (and switch D) essentially unchanged and causes the corresponding data transfer between processor component A and memory B involving bus 3 (the "*memory bus*"). If the switch is in position "open" the information on bus 1 has no effect on C*i*, D*k* or B.
If the information is an "open *j*" or "close *j*", switch C compares its own hardware address *i* with address *j*. If they are equal, switch C is opened or closed, respectively. If they are unequal, switch C is not affected.
If the information is "set non-local sensor *j*", "reset non-local sensor *j*" or "load non-local sensor *j*", switch C compares its own hardware address *i* with address *j*. If they are equal, then the sensor bit S in switch D, which is connected with C by bus 2 (the "*interconnection bus*") is set to 1, reset to 0 or the content of S is loaded to the accumulator of processor A, respectively. If *i* is unequal to *j*, no operation is executed.
Switch D with hardware address k not only analyzes and handles information arriving on bus 2, but it simultaneously also can handle information arriving from processor A' on bus 4: A store or load instruction, a "close *j*" or "open *j*", a "set non-local sensor *j*", "reset non-local sensor *j*" or "load non-local sensor *j*", and, finally, a "set local sensor *j*", "reset local sensor *j*" or "load local sensor *j*", where *j* is unequal to *k*, on bus 4 does not affect switch D. However, a "set local sensor *j*, "reset local sensor *j*", "load local sensor *j*", on bus 4, where *j* is equal to *k*, causes the sensor bit in D to be set to 1, reset to 0 or the content of the sensor bit to be loaded into the accumulator of processor A', respectively.

3.5. The Full L-Network

Taking *n* L-modules with *n* switches of type C and *n* switches of type D one can also interconnect each of the *n* L-modules with every other one (thus realizing the "full graph"): for example, the switches C1,...., C*n* of the *i*th L-module can be connected with switch *i* of the first,...., *n*-th L-module as in Fig. 7 (displayed for $n = 3$). *n* components of type A, *n* components of type B, n^2 components of type C, n^2 components of type D, *n* processor buses, *n* memory buses and n^2 interconnection buses would be necessary. By a geometrical transformation that keeps the logical and functional behavior of the arrangement completely untouched, the same arrangement can also be realized by an arrangement, in which the number of interconnection buses is zero (see [6]). We will refer to this realization as the "$n \times n$ *parallel L-machine*". It resembles a crossbar. However, it has some extraordinary features:
1. Exactly the same components A, B, C, D can be used for the crossbar as for the L-modules and L-networks.
2. The crossbar allows the programmer to realize, by software means, every special L-network structure he wants.
3. The crossbar needs no additional control component in order to open and close the cross points. This control is exclusively organized by the processor components A themselves. This has important consequences (see 4. and 5.).
4. The components A, B and C/D have an interconnection environment (in terms of the number of lines leaving these components), which allows them to be integrated easily on one chip. The design lends oneself to VLSI implementation. Full L-networks with several hundred components A and B should be realizable in present VLSI technology.

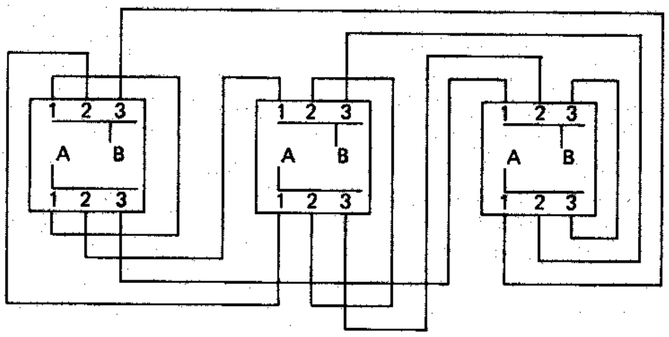


Fig. 7

5. The crossbar is easily extensible. No restructuring of the system is necessary for extension.
At present, a pilot full L-network with 8 components A, 8 components B (and, hence, 64 components C/D) is in operation at the University of Linz (in cooperation with ICA-Vienna). Simulations for a VLSI implementation are in process.
In Section 5 we will present a rough sketch showing how the parallel L-machine can be used for realizing a parallel connection algorithm.

4. Outline of a Parallel Connection Procedure

With the material from the introductory sections 1–3 in mind, we are now in a position to outline a proof procedure for first-order logic based on the connection method that heavily exploits parallelism. Let us refer to it as the parallel connection procedure, PCP for short. The description will be on a very high verbal level, although intuitively it is based on the language constructs provided by the L-networks. That is, we claim that PCP can adequately be realized by an L-network, a project which is further discussed in the next section.
For the purpose of this outline we make a number of simplifying assumptions. The more restrictive ones will of course not be made in a future realization. First of all we restrict ourselves here to formulas in clause form (cf. [8], for a discussion of this restriction). It is also assumed that one of the clauses can be regarded as a goal clause as in PROLOG. The reader may even think of all clauses to be Horn clauses although PCP is not really sensitive to the difference. Also PCP here is designed to stop after detecting a single proof with the resultant output rather than to look for all possible outputs in an exhaustive way (as required e.g. for certain data base applications). Also the case of failure is not discussed.
Further we ignore in the discussion all of the more sophisticated issues of the connection method like preprocessing, selective backtracking, factorizing, splitting, avoiding skolemization, identification of functional patterns for compilation, and the like. Even with all these restrictions PCP will provide a reasonably efficient first version which, in contrast to current PROLOG systems, is even complete since the parallel approach allows us to disregard any order of the clauses.
On the machine side we have in mind the topology of a full L-network, so that there are no restrictions w.r.t. the interconnections needed. We ignore any effects that may arise by overflow situations; in other words we assume that, for instance, the number of processors is (virtually) unlimited. Under all these assumptions we are now going to discuss some of the details of PCP which on the topmost level reads as follows.

```
read/prepare the given matrix;
until one of the processors signals success do the tasks
t1, t2 and t3 in parallel:
t1: develop the potential top-down solution trees in
    parallel;
t2: for each connection $k_i$ in parallel compute the
    weak most-general unifier; for each alternative
    set of neighboring connections in parallel combi-
    ne their unifiers and continue this combination
    with their neighboring connections;
 3: develop heuristic priorities based on global
    considerations, and pass this information to t1
    and t2;
```

For illustration recall the example introduced in section 1. If we ignore in its presentation the details of the literals, we obtain the abstracted graph structure of Fig. 8. Here those nodes are connected that represent pairs of literals of the form ($Ps_1,\dots, s_n$; $-Pt_1,\dots, t_n$), while the equality literal $s = t$ has an open-ended connection. The connections are numbered in an arbitrarily fixed way. A comparison with the two proofs depicted in section 1 shows that on one hand only a subset of these seven connections is needed for the proof while on the other hand some of these connections have in fact to be taken into consideration with more than one instance. For example the first proof in section 1 is made up by the connections 1, 3, 5, and 7, while the second one in addition requires the connection 4 and a second instance of the connections 5 and 7.
In terms of the connection method theorem proving may thus be described as the task to determine an

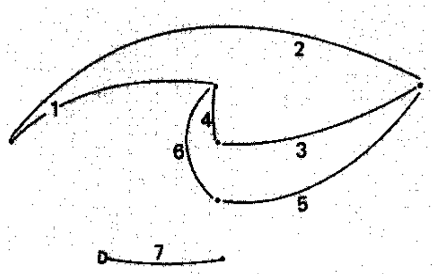


Fig. 8

appropriate subset of the initial set of connections, together with the numbers of their instances needed for the proof. Both the logical structure and the unifiability of terms influence this determination as pointed out in section 1. Let us consider the first of these two influences.
Without restriction of generality we may assume that the goal clause consists of a single literal only (otherwise we add such a singleton connected with an additional literal in the original goal clause). Then any proof may be regarded as a tree the nodes of which are literals (as is well-known and as we will illustrate later). In order to find such a solution tree, several potential solution trees have to be taken into consideration until an adequate one is identified. This is the main task carried out in t1 whereby the different potential solution trees are explored in parallel.
If we pursued a pure top-down development, then at the beginning as many processors would become active as there are potential solution trees, that is very few. The rest, that is many, would be idle. In order to avoid this waste of computational power, pieces of the branches of the potential solution tree will be developed in a bottom-up way in parallel until the top-down development arrives at such a preprocessed piece which will then be incorporated, thus further expanding the tree. This bottom-up development is meant to be carried out in task t2.
Considerations concerning the global structure of the given matrix may be used for heuristically guiding the developments in t1 and t3. If, for instance, it turns out that a certain connection is the only connection in one of the paths then the unifier of this connection is crucial for the rest of the proof, a fact which might discard several of the potential trees or pieces of branches. Such global guidance is meant to be provided by the task t3.
This completes a first superficial description of PCP (on the second level, so to say). We will detail this description a little further on the third level in the rest of this section. This begins with a more precise characterization of notions which were used in a more intuitive way until now.
For any connection k and any clause c, a connection l is called a *neighboring* connection for k (in c) if $K \in k$ and $L \in l$ for some (neighboring) literals $K, L \in c$ with $K \neq L$. For instance, the connections 3-7 all are neighboring for 1 while there is no neighboring connection for 2. Connections which share a literal are called *alternative* connections, like 1 and 2. Note that some pairs of connections are both neighboring and alternative, like 3 and 4. Then a *potential directed*

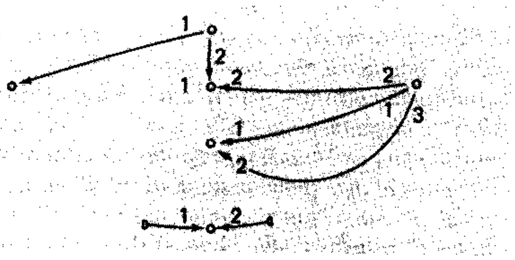


Fig. 9

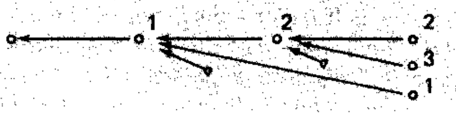


Fig. 10

solution tree is obtained as illustrated for the present example in the following.
One alternative connection of the goal clause is selected, say 1, and directed towards the goal clause. For each neighboring node of the origin of the current directed arc 1, we select one connection containing it and direct it towards this node, whereby different instances of the clauses are distinguished. And so forth with each of the newly selected arcs until there are no neighbors available. For instance, we obtain the directed graph of Fig. 9 for our example. It corresponds to the proof for input 3 shown in section 1. As we said before, it encodes a tree. This may be seen from the equivalent representation (with an obvious correspondence with the previous one) in Fig. 10.

Different potential solution trees are obtained by different selections from alternatives. One of them must represent the desired proof (if there is one at all). Their identification is the task labeled t1 in PCP. This task is supported by the work done in task t2 which is now described in further detail.
Each of the initial connections is taken in charge of one of the processors which knows its neighboring processors in charge of the neighboring connections. Such a processor may activate new processors by need, in case new instances of clauses have to be taken into consideration. So each of all these processors contributes to the development of the potential solution trees as much as it can with regard to its limited local knowledge and its limited processing time, whereby it is avoided that the same work be duplicated.

First of all, these processors compute the weak unifier for their connections, that is the most general unifier (*mgu*) for the two connected literals with no variables in common which is achieved by renaming. After completion of this task one of any two neighboring processors computes the common unifier for their two previously computed unifiers. For logical reasons both connections are part of a potential solution tree, hence these computations anticipate work eventually to be carried out by the topdown development in t1.
Each of these two processors then continues to carry out the same job together with a neighboring processor which is not a neighbor of the other one. This way the branches of the potential solution trees are growing in a bottom-up way whereby different pieces may grow into each other. This kind of process is well-known also in other areas of Artificial Intelligence, for instance in vision, where it is known as the region growing technique.
Eventually, the top-down process t1 will encounter a connection that has already become part of such a piece of a computed branch. This piece can be integrated into the potential directed solution tree under development as a whole in one step. This involves the computation of the common unifier from the two, one of which has been computed so far by t1, the other by t2; it also involves the propagation of the direction of the tree.
Of course, there are plenty of details to be taken care of in this whole process. Some of them are

- the appropriate combination of any two AND-branches;
- the updating of the common unifier in combined AND-branches;
- a treatment of alternative trees such that the work in joint branches is shared.

We cannot now go into any of these details. Nor will we give a more detailed explanation of t3. Both will require a lot of theoretical and experimental exploration which is currently pursued. Let us, however, imagine how this might work with our present example for input 3.
Assume we use 2 processors per connection, k_i, one concerned with the solution tree, the other with unification. Then after, reading and preparing the given matrix, there are initially 14 active processors, say 1.1, 1.2, 2.1,..., 7.2. Each unification processor *z*.2 computes the weak mgu for connection k_z. Note that e.g. *i* (as well as *j*) is taken to be different in the two literals connected by 4. At the same time *z*.1 takes notice of its neighbors. For instance, processor 1.1 selects, say, 3, 5, 7 as its neighbors which in detail means the following.
3.2, 5.2, 7.2 are signaled to stop after completing their current unification and to pass their result to 1.2 for further unification. 1.1 then takes over all the additional neighborhood of 3, 5, 7 (which is empty in the present case) and 1 is deleted as an alternative neighbor for them. Finally, for the remaining alternatives 1.1 passes the role of goal processor (i.e. the direction of the tree) to one of the neighboring processors, say to 4.1, together with the result of 1.2. In the meantime, 3.1 has proceeded similarly. Say, its selected neighbors are 4, 5, 7. As soon as 4.1 receives the role of goal processor, it is programmed by 3.1 to pass this role to 3.1 for this particular selection together with the result of 1.2 which, under control of 3.1, is successfully unified in 3.2. 3.1 thus stops all other activities.

5. Realization of the Parallel Connection Method on the Parallel L-Machine

In this section we show how some typical parts of the parallel connection method can be realized on the nxn "parallel L-machine" (where *n* is assumed to be quite large). We concentrate on task t1, i .e. the top-down development of the solution tree. A complete realization of a parallel connection method on L-networks will be given elsewhere. For brevity, we must restrict ourselves to the case where the substitutions computed for subgoals are compatible.
We assume that, at the beginning, the following program is stored in the program memories in component A of all the L-modules in the parallel L-machine and that the given matrix (formula) together with information about the possible connections is stored in the memories B of all the L-modules. Here, we use brackets [] for denoting parts of the program realizing a dialogue with other L-modules (using the sensor instructions and the open/close instructions). We first give a rough description using self-explanatory high-level constructs (Fig. 11). Later we will show how some of these constructs may be realized by the special hardware instructions available.

Remark: Of course, in a more sophisticated version of the program, a son would only be initiated if more than one subgoal has to be treated.

```
entrance:
[wait until some other L-module (a "father") needs a
 "son" for treating a subgoal;
 receive the following information from the "father":
   physical number of the father,
   substitution for the variables in the goal clause;
 wait until the father initiates computation:]

for all connections leaving the goal clause do
   determine the most general unifier U (after making
   variable sets disjoint);
   if set of subgoals is empty
      then goto exit
      else
         evaluate the arguments of the subgoals;
         for all subgoals do
            [search for a free "son";
             transmit the following information to the son:
               own physical number,
               substitution for the variables in the subgoal:]

[wait until for one of the connections all sons are ready
 and all substitutions computed by the sons are
 transmitted
 or until the father requires termination of
 computation;]

compose these substitutions with the corresponding
substitution U;
[for all the other connections terminate the work of the
 sons;]

exit:
[inform the father about termination;
 send the resulting substitution to the father;]

goto entrance.
```

Fig. 11

We now show for some parts of the program how the dialogues involved may be realized by the instructions of the L-machine. For abbreviation, we use the following notation:

S_j... the "local" sensor bit in switch D_j, (which can be accessed by the local sensor instructions),

T_j... the "non-local" sensor bit in the switch of type D in the neighboring L-module that is connected with switch C_j of the own L-module.

Instead of the internal instructions "set local sensor j" etc. we allow "high-level" notations as, for example, "$S_j := 1$". First, we consider in more detail the procedure of waiting until all sons are ready. This part of the program can be programmed as follows:

```
....
wait until there exists a connection $j$ such that for all
        sons $m$: $T_m = 0$;
for all these $m$ do
$T_m := 1$;
if $T_m = 1$ wait;
...
```

Correspondingly, the part where the father is informed about termination must have the following form:

```
...
$S_f := 0$;
if $S_f = 0$ wait;
send the resulting substitution to the father;
(In more detail:
   close $C_p$ and load the substitution to the private
   memory;
   open $C_p$ and close $C_f$;
   store the substitution from the private memory
   into the shared memory of the father;
   open $C_f$.)
$S_f := 0$;
...
```

Here, p is the own number of the L-module considered, and f is the number of the father. Furthermore, the sensor bit that can be accessed using the name T_m by the father and using the name S_f by the son is set to zero for signalizing that the son has terminated work. Similarly, the instruction "there exists a connection j such that ..." could be realized by the sensor instructions. In Fig. 12 we show how the dialogue at the entrance of the program can be realized.

```
....
entrance:
for all $j \neq p$ do
   if $Sj = l$ then
      for all $l \neq p$ do $T_l := 1$;
      $a_j^{(p)} := 1$; $f := j$;
      for all $l \neq p$ do
         if $S_l = 1$ then begin $S_l := 0$; if $S_l = 0$ wait; $S_l := 0$ end;
      goto start;
goto entrance;
start:
if $S_f = 0$ wait;
...
```

Fig. 12

Here, $S_j = 1$ signalizes that L-module j wants the given L-module p as a son. $T_i; = 1$ signalizes to all L-modules that p is occupied. In memory location $a_j^{(p)}$ the information that j is the father of p is stored. For initialization we assume that all entries in a are zero. "*for all* $l \neq p$ *do if* ..." allows all L-modules l to inspect $a_l^{(p)}$.
Correspondingly, the program part "search for a free son" can be realized as in Fig. 13.

```
...
for all subgoals do
  while no free son found do
    for all $k \neq f, p$ do
      if $S_k = 0$ then $T_k := 1$;
        if $T_k = 1$ wait:
        if $a_p^{(k)} = 1$ then $b := k$;
                              $s := 1$
                              {a son is found};
                              transmit the substitution to the son
                         else $s := 0$;
        $T_k := 1$;
        if $T_k = 1$ wait;
        if $s = 1$ then goto $\beta$;
β: $T_k := 1$;
```

Fig. 13

$S_k = 0$ signalizes that L-module k is free. $T_k := 1$ informs L-module k that p wants k as a son. $T_k = 0$ informs p that p may inspect $a_p^{(k)}$ for seeing whether p is accepted by k as the only father. $T_k := 1$ initializes the computation in the son b. In this synchronization part some subtle theoretical timing problems arise, which can not be treated in this introductory paper.

Acknowledgements

We thank the members of the Intellectics group at the TUM who have provided considerable help in a number of discussions; in particular K. Aspetsberger, E. Eder, C. Freksa, and F. Kurfess have provided several ideas and suggestions. L. Zalewski has prepared the typescript and improved the English.

References

[1] W. Bibel, Prädikatives Programmieren. LNCS **33**, Springer, Berlin (1975) 274-283.
[2] W. Bibel, Automated theorem proving. Vieweg, Braunschweig (1982).
[3] W. Bibel, Matings in matrices, CCAM **26** (1983) 844-852.
[4] B. Buchberger, Computer trees and their programming. 4th Coll. "Trees in algebra and programming", Univ. Lille, Feb. 16-18 (1978) p. 1-18.
[5] B. Buchberger, Components for restructurable multimicroprocessor systems of arbitrary topology. MIMI 83, Lugano, Acta Press, Anaheim (1983) 67-71.
[6] B. Buchberger, The present state of the L-network project. MIMI 84, Bari, Acta Press, Anaheim (1984) 178-181.
[7] E. Eder, Properties of substitutions and unifications. GWAI-83 (B. Neumann, ed.), Springer, Berlin (1983) 197-206; also to appear in the Journal for Symbolic Computation.
[8] E. Eder, An implementation of a theorem prover based on the connection method, in:[1], (W. Bibel, B. Petkoff, eds.), North-Holland, Amsterdam (1984) (to appear)[2].
[9] K. Hwang, F.A. Briggs, Computer architecture and parallel processing. McGraw-Hill, New York (1984).
[10] P. Jorrand, FP2: Functional parallel programming based on term substitution, in:[1] (W. Bibel, B. Petkoff, eds.), North-Holland, Amsterdam (1984) (to appear)[2].
[11] Y. Parker, Multi-Microprocessor Systems, Academic Press, London-New-York (1983).

[1] Artificial Intelligence: Methodology, Systems, Applications
[2] ISBN 0 444 87743 6