B-LOG: A BRANCH AND BOUND METHODOLOGY FOR THE PARALLEL EXECUTION OF LOGIC PROGRAMS

G. J. Lipovski
and **M. V. Hermenegildo**

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712

Abstract: We propose a computational methodology -"B-LOG"-, which offers the potential for an effective implementation of Logic Programming in a parallel computer. We also propose a weighting scheme to guide the search process through the graph and we apply the concepts of parallel "branch and bound" algorithms in order to perform a "best-first" search using an information theoretic bound. The concept of "*session*" is used to speed up the search process in a succession of similar queries. Within a *session*, we strongly modify the bounds in a local database, while bounds kept in a global database are weakly modified to provide a better initial condition for other sessions. We also propose an implementation scheme based on a database machine using "semantic paging", and the "B-LOG processor" based on a scoreboard driven controller.

1 Introduction

Logic programming is a major new facet of fifth generation computing [15]. Simultaneously, parallelism is widely proposed as a means to reach the performance goals imposed on fifth generation machines, which are not attainable with conventional sequential processors. However, parallel computation of logic programs has been shown to be difficult. Herein, we propose a computational methodology -"B-LOG"- based on the concepts of Logic Programming [10] offering the potential for a more effective implementation in a parallel computer than that of Prolog. We also propose an architecture to implement this methodology, which we call a "B-LOG machine".

The basic ideas are simple: the execution of a Logic Program can be modeled as a search process through an AND/OR tree [4] or through an OR-tree. In our approach weights are added to each branch of the OR-tree. In this way the notions of branch-and-bound algorithms can be used to perform a "best-first" search rather than the simple depth-first search present in Prolog [13]. Obviously one of the main problems which have to be solved when selecting such an approach is that of which particular bound to use. We propose one which is related to the information content of the decision, and may be modified by previous searches in an adaptive control strategy.

From the point of view of implementation, another interesting feature is proposed: the retrieval of data from a semantic paging disk memory [5]. The realization of the B-LOG processor itself is, on the other hand, proposed using an associative controller similar to the CDC 6600 scoreboard. These implementation techniques are sketched here to provide an effective definition of the search strategy.

The layout of the paper is as follows: the next section describes the database and search tree, using Prolog as a starting point and section 3 introduces the branch and bound approach. In section 4 the weighting scheme is described and section 5 presents the search and weight update strategies in the B-LOG machine. Section 6 describes a possible implementation in a parallel computer/database machine. Section 7 discusses AND-parallel extensions to the model and, finally, section 8 gives our conclusions.

2 A Model for the Data Base and Search Tree

In order to present our model of the search tree and database, let us consider the problem of finding all solutions to a query using conventional Prolog in the classic example given by Conery and Kibler in [4]. A Prolog listing for this example is given in figure 1. It shows rules, facts (the database) and the series of searches generated by a certain query.

RULES

```
gf(X,Z) :- f(X,Y),f(Y,Z).

gf(X,Z) :- f(X,Y),m(Y,Z).
```

DataBase

```
f(curt,elain).   f(sam,larry).    m(elain,john).
f(dan,pat).      f(larry,den).    m(marian,elain).
f(pat,john).     f(larry,doug).   m(peg,den).
                                  m(peg,doug).
```

Queries

```
?- gf(sam,G) -> gf(X,Z) :- f(X,Y),f(Y,Z)     (X/sam,Z/G)

?- f(sam,Y),f(Y,G) -> f(sam,larry)           (Y/larry)

?- f(larry,G) -> f(larry,den)                (G/den)
```

Figure 1: A Prolog Example

The fact that curt is the father of elain may be expressed as

```
f(curt,elain).
```

(constants are lower case, variables are capitalized in Prolog). Thus, there are ten facts in the example. A rule stating that X is grandfather of Z if X is father of Y and Y is father of Z may be coded as

```
gf(X,Z):-f(X,Y),f(Y,Z).
```

Thus, there are also two rules in this example.

We can apply queries to this system of facts and rules. The query (or "goal"), "Who is a grandchild of sam?" is stated as

```
?:-gf(sam,G).
```

Prolog will try to answer this query by searching through the database. The steps followed in this search are also shown in figure 1.

Execution is as follows: the first search for a match to gf(sam,G) produces two matches to the rules. In Prolog, the top rule is chosen, instantiating X to sam and Z to G (Z and G "share"). The next goal will be f(sam,Y) which is resolved as f(sam,larry) instantiating Y to larry. The subsequent search for f(larry,G) produces f(larry,den) instantiating G to den. In this way we conclude that den is a grandchild of sam.

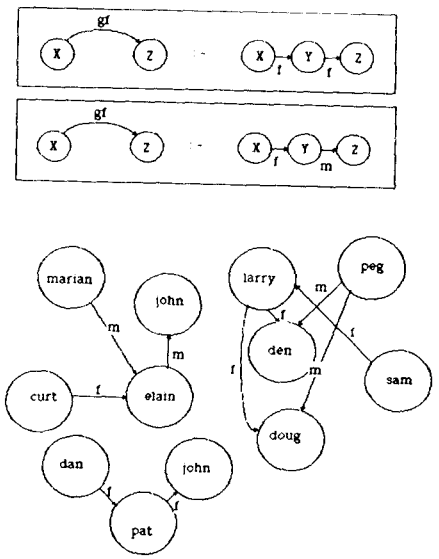

Figure 2: The Data Base

The database can be graphically shown as in figure 2. The facts, shown on the bottom, represent persons (marian) as *nodes* and relationships (mother of) as *arcs* in a network model. Rules in the top of the figure are shown as *equivalences of graphs*, in a consistent notation. The graph to the left of the :- can be replaced by the graph to the right of :- as indicated by the *Horn clauses* of the Prolog listing.

An OR-tree that gives *all solutions* to the query

```
gf(sam,G).
```

is shown in figure 3. As we have seen Prolog using depth-first search would generate the chain from the root to the leftmost leaf. In our representation of the search for all solutions, the query is shown as the root, and each resolution step, that is, each search for that graph in the database and rules, is shown by an arc below that node. A search is thus conducted by a graph query. A match is found wherever this graph can be embedded as a subgraph in the data base or in the left side of a rule. The top half of each node is one match to the goal shown on the bottom of the node above that node. Thus, matching the subgraph

```
(sam)--gf-->(G)
```

on the left of the two rules produces the two graphs

```
(sam)--f-->(Y)--f-->(G)
```

and

```
(sam)--f-->(Y)--m-->(G)
```

which are shown on the top halves of the nodes below the root node.

Consider the left node just produced. The complex graph

```
(sam)--f-->(Y)--f-->(G)
```

is decomposed into simpler graphs

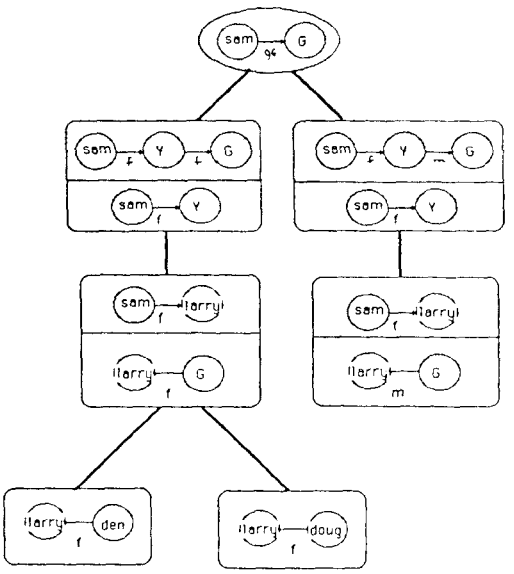

Figure 3: The Search Tree

```
(sam)--f-->(Y)
```

and

```
(Y)--f-->(G)
```

and the first simpler graph is chosen (depth-first search). We write that graph in the bottom half of the node to indicate that it is the next graph to be searched for. The match to this search produces one result, which is

```
(sam)--f-->(larry)
```

shown in the top half of the node below the previous node. We now decide which graph to search next. Traversing from this new leaf towards the root, we collect all unused graphs, finding

```
(Y)--f-->(G)
```

in the middle node. We make this our next search comparand. We find the two results

```
(larry)--f-->(den)
```

and

```
(larry)--f-->(doug)
```

Transversing from the node

```
(larry)--f-->(den)
```

rootward, we see no unsatisfied unknowns to be matched in graphs, so this is a solution. Similarly, the node

```
(larry)--f-->(doug)
```

is a solution. Following the right node below the root produces the same search for

```
(sam)--f-->(Y)
```

giving the same answer

```
(sam)--f-->(larry)
```

as obtained earlier, but the search for

```
(larry)--m-->(G)
```

produces no matches. Since there are no matches, this portion of the search is unsuccessful.

3 Sequential and Parallel Search: a Branch and Bound Approach

The tree in figure 3 is clearly an OR tree. Each "fan-out" below a node is an alternative solution to the query stated at the bottom of the node. There are no "fan-out" arcs representing an AND condition as in the formulation given by [4] of the Prolog search tree: in our simplified model we consider AND-trees now only in a sequential way, in very much the same way Prolog does. We will still discuss some AND-parallel extensions to the model after presenting the OR implementation.

Each chain from a leaf to the root is either a solution to the query at the root or an unsuccessful search (a "failure"). Each arc in a chain represents a decision made towards the solution of the query or unsuccessful search. This tree is basically a branching graph that represents the enumeration of all solutions in a branch-and-bound algorithm.

Some aspects of the problems involved in the parallel implementation of logic programming are evident upon inspection of this tree. Obviously, for NP-complete problems, no matter how much parallelism we use, because the number of processors is in reality fixed by limitations of hardware, each processor still has a sub-problem that is NP-complete. In addition, the scheduling problem makes it impossible to always use the total number of processors available in a useful manner. Thus, parallelism by itself would seem not to render a solution to the problem of building effective logic programming machines. However, a solution for the general purpose parallel inference machine has to exist: the existence and rather satisfactory operation of the human brain gives us hope in this sense, and some studies of the **average** complexity of search algorithms show that in practice many problems that are NP-complete are much better behaved in the average case (to the point of sometimes being linear). This has been shown for depth-first search algorithms with a suitable bound [14].

Also, depth-first search, though useful in single processor implementations, does not lend itself easily to parallel processing. Breadth-first search would seem to get a great number of processors working on different independent problems, but it tends to work near the root of the tree, doing extra work before a solution is found.

An approach based on a branch-and-bound algorithm seems more appropriate using best-first search guided by a bound. Strictly speaking, a bound is a number that is monotonic on each arc in any chain from root to leaf (say it is monotonically increasing) and is a measure of the goodness of the result so that the solution is sought with the minimum value of this bound. Once a solution is found, its bound can be used to cut off any searches on other chains if their bound is greater than the one found. If a solution can be found quickly, its bound can be used to save a lot of effort in growing chains that cannot produce a better bound.

Parallel searching is possible in a branch-and-bound problem and a number of schemes have been proposed [11]. From the point of view of implementation, suppose there are n processors in a MIMD computer. As the tree is developed, referring to the final form of the tree, at any time there is an imaginary line or "wave front" cutting across the tree such that all arcs rootward from this front are found and all arcs leafward from it are to be found. Assume a number m>=n of nodes are on this wave front, and corresponding to each such node is a chain from it to the root of the tree. Each such chain has a bound computed in some way from the weights of the arcs in the chain. Each processor P_i i=1..n works on the n chains with the lowest bounds. A sorting network like Batcher's [1] could be used to sort the bounds, assigning the n lowest bounds to the n processors and communicating the associated chains to them to work on. A sorting network is costly, and communication costs restrict this approach, but a reasonable approximation is effectively possible. Such a design is considered in section 6.

4 The Weighting Scheme

Considering the advantages of best-first searching over depth-first or breadth-first searching, we have concluded that a best-first search strategy is an attractive possibility for use in a logic programming parallel computer. Of course, the main question is: what is the bound that we can use for this case?

The solution to this problem is not easy and we presently have to settle for a compromise approach. We will present a bound whith which we could feel comfortable about in a theoretical way, but would have difficulty implementing in hardware, and another that we can implement, but have little theoretical basis for. We present the first bound and the theory behind it, to define our computational methodology (B-LOG) and to provide some basis in order to justify the second heuristic bound that we will use in the B-LOG machine.

Consider a tree that is constructed after obtaining all complete solutions to all queries put to a database and set of rules, assuming that each solution is equally likely, and each decision is statistically independent of the others. The root of this tree is the primeval query (?) and its descendants are the roots of trees, like the tree in figure 3, that represent the complete solutions to each such query. We attempt to use as bounds in the branch and bound algorithm values assigned to each chain from the root to the node being considered, which have been computed from the (unnormalized) probabilities of the arcs in the chain.

Let p(k) be the (unnormalized) probability that arc k is in a successful solution in the following sense:

1. If an arc appears twice in a tree (as the arc from (sam)--f-->(Y) to (sam)--f-->(larry) in figure 3), they have the same (unnormalized) probability. This is required if these probabilities are to be stored in a database that is common to all queries.

2. The probability of each chain representing a successful solution must be equal to 1/(the number of successful solutions).

3. The probability of each chain representing an unsuccessful search must be 0, for the bound to be meaningful.

Note that with this definition once a node is arrived at, the probabilities of the arcs coming out of the node are independent of the path used to arrive at that node. This process has thus a Markovian flavor but the unnormalization of the probabilities prevents us from drawing any further conclusions in this sense.

Since in our model the arcs represent statistically independent decisions, the probability of a chain is the product of the (unnormalized) probabilities of the arcs in it. We are lead to define the bound of a chain as the product of the (unnormalized) probabilities of the arcs in it. While this would be a useful bound, it would require multiplications of fractions. However, using logarithms, we could add rather than multiply. Converting to logarithms, the bound of each successful solution would still be equal to that of any other successful solution.

In order to use the more efficient logarithm implementation, we define the weight W(k) of an arc k to be the negative of the logarithm (base 2) of the (unnormalized) probability of the arc in the sense given above and we define the bound B(n) of any chain n = (i,j,k,...) to be the sum of weights of the arcs in the chain W(i)+W(j)+W(k)+.... As a chain is built from the root, the bound is monotonically increasing, since the logarithm of a fraction is negative and we add the negative of these negative components, and all successful solutions have the same bound. Thus, it properly satisfies the requirements of the branch and bound algorithm. Incidentally, the weight of an arc resembles the information or "surprise" associated with making a decision, as quantized in Information Theory, whether by coincidence or from some very fundamental reason. In the solution process, the branch and bound approach tries to minimize this "surprise", seeking the most "obvious" solution.

However, do such probabilities exist? If N is the number of both complete solutions and unsuccessful solutions, and M arcs are used in them, we have N equations in M unknowns to solve, which are linear equations formulated in terms of the weights of the arcs. Since M>>N we expect to have such bounds. Generally, there may be many solutions, and any one will satisfy our branch-and-bound requirement. However, pathological cases exist where no solution is possible. For instance, if an unsuccessful query has only arc A, then the weight of A must be infinity, but if A is an arc in a successful solution, it may not have a weight of infinity. In such a case, there are no weights. When weights exist, we do have a properly formulated branch and bound algorithm.

Of course, in a practical case, we do not want to wait until all solutions to all queries have been found, and then try to solve a large number of linear equations in a larger number of unknowns to get the weights. This notions, however, will serve as a guideline in constructing the heuristic rules that we will actually use in a B-LOG machine.

As an illustration of the above described scheme, consider the example in figure 3, as if that query were the only one ever presented to the database. One valid set of weights which can be verified by inspection is the following: The arcs above (sam)--f-->(Y)--f-->(G) and both instances of (sam)--f-->(larry) have probability 1, those above (larry)--f-->(den) and (larry)--f-->(doug) have probability 1/2 and that above (sam)--f-->(Y)--m-->(G) has probability 0. The probability that a chain from root to leaf is a solution is the product of the probabilities of each arc in the chain. Both solutions have probabilities 1/2 and the unsuccessful solution has probability 0. The weights of arcs with probabilities 0 would be infinity, of those with probability 1/2 would be 1, and of those with probabilities 1 would be 0.

We define a B-LOG machine as a MIMD computer that approximates a "best-first" search strategy on a logic program, using weights in order to guide the search. These weights will be updated with each query so that they will eventually converge to be proportional to those described by the theoretical model above as all queries are presented to the database, as long as the contents of the database (except for the actual weights) are not modified. We call it "best-first" in quotes because it will be only an approximation to true best-first searching.

5 Search and Updating Strategies

The database (see figure 2) will be stored as a linked list data structure, with blocks representing each Horn clause (rule or fact), and pointers to blocks representing other rules or facts in the database that can resolve the rule. During a session, we aim to set the bounds of all successful queries to the same constant, which we arbitrarily set to a number N. The weights of the arcs in the search tree correspond to weights on pointers in the database. Each pointer will have an "unknown" weight, initialized to N+1 (which will be larger than a known solution that has a bound N). Some of the arcs may have weights set by earlier queries, which we will call "known" if they are set because of a successful search, or "infinity" if set by an unsuccessful search. If the longest chain in a search tree is A arcs, we code "infinity" as A*N. If a failed search occurs and it does not already have an arc with infinite weight in the chain, we will set any one of the unknown weights to infinity. The choice of which weight to set to "infinity" is similar to the backtracking problem in Prolog; we think it should be the unknown nearest the leaf in the chain. If a solution to the query is found, we will reset all unknown or infinite weights as follows: if the known weights add up to a number greater than N, set them to 0, else if there are k unknown or infinite weights, set them equally so that the sum of weights is N i.e. if the known weights add up to M, set them to (N-M)/k

Consider this example:

```
A:- B,C,D.
B:- E.
B:- F.
C:- G.
D:- H.
```

The set of Horn clauses shown above would have the data structure in figure 4. Note that each clause is represented by a block, and that just below each named pointer is a weight. It may be recognized that these blocks

are much like inverted files kept for each rule. The updating process for this data structure will be similar to the updating process for inverted files. This substantial increase in database size and update complexity is needed so that weights can be maintained for each arc, in order to use "best-first" searching.

Consider evaluating a query in a single processor using the B-LOG methodology. See figure 4 for an instance of the weights. We will consider the parallel approach in the next section. When a query "?:-A" appears, it will match the first clause (provided unification succeeds). The next search could be for B (with two possible matches) or C or D (with one match each). Examining the bounds of each chain obtained so far (which are just the weights of each of the four pointers), B-LOG will choose the chain with the least bound (which is the pointer with the least weight). The second pointer to B has the lowest weight (3). Then the Bs will be chosen for the first fan-out below the node representing A in the search tree, and the right side of the Horn clause of the second B would be searched. The bounds associated with the chain to F (the sum of the weights of the second B and of F) would be compared with the bounds (which are the weights) of the first B, of C and of D. The first B is chosen because its weight is lowest and a new chain is grown from the root node to the first B. Note that the next search from the first B is similar to a breadth-first search.

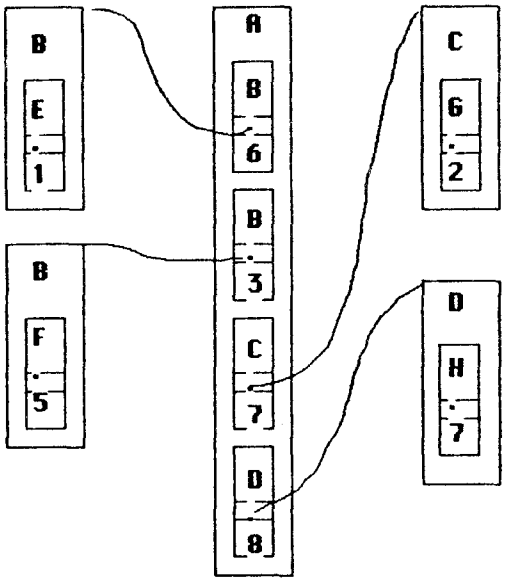

Figure 4: A Linked List Structure

A different set of weights would cause a different order of search. Suppose the weight of the first B pointer is the Horn clause for A (middle of figure 4 were 1 rather than 6. Then the Bs would be chosen for the first branch of the search tree, as before. But the Horn clause B:-E would be expanded next, before the second B would be chosen, because the sum of weights for this chain (2) would be lowest. This appears to be a depth-first search, as in PROLOG. In general, the "best" chain would be expanded first, rather than depth-first or breadth-first.

We reflect on the probabilities and the weights related to them. It is tempting to normalize the probabilities of arcs out of a node. For instance, we might make all the probabilities of arcs away from node A sum up to 1. This "best-first" methodology, however, compares bounds which are weights of arcs out of node A with those out of nodes B, C and so on. The weights are thus defined in terms of the probabilities of an arc in any solution to the query, that is, they are globally rather than locally defined. Also note that the weights are stored with the pointers, rather than at the beginning of each block. This speeds up the search process because we can decide whether we wish to retrieve another block by examining these weights, before we access the block from the slow secondary storage.

As long as no infinite weights are reset to known weights and the sum of known weights does not exceed N, this heuristic yields one possible solution to the branch and bound algorithm (the weights will be proportional to those of the algorithmic approach). When these anomalies appear, it appears too hard to completely correct the entire data base, and we may not be able to do so anyhow. Still, we must remember that all we are doing is trying to keep a lot of processors busy doing useful things when we use this "bound". We must keep in mind that the algorithm is only a guide to this end, and small deviations from the theoretical model will reduce efficiency, but the correct solution(s) will still be found.

This heuristic employs some adaptive control strategy. If a successful query is found, the next search will try this path early and if an unsuccessful search is detected, its path will be avoided until all the others have been attempted. Especially where a user tries a second and third query that is similar to the first one with some minor changes, later searches should become more efficient.

In order to make the above described convergence possible we have to provide a strategy for maintaining and updating the weights in the B-LOG machine. One important issue at this point is to determine the scope and extent of these changes. To do so, we will introduce the concept of *session*. A *session* is defined as succession of queries during which no permanent updating of weights is done in the global database in secondary storage. During a session, weight updates are kept in a separate buffer or in local copies of the subset of the graph being used in primary memory.

The user declares the end of the current and the beginning of a new session when the next query is not related to the previous queries. At the end of the session the global database will be updated in a "conservative" way, e.g., no infinities will override previous non-infinite weights, while other weights will be modified in the direction indicated by the results of the session. This less drastic modification will provide an improved initial condition at the beginning of the new session. Averaging of modifications over different sessions is thus achieved, hopefully facilitating convergence to the theoretical model.

Other bounds may be used, and some perhaps may show more useful than the one defined for the B-LOG model. For example, conditional probabilities (conditional information) might be added to the model, since a decision should depend on what has been previously decided, but maintaining the database in this model is clearly more difficult than our approach. We thus feel that this model is

both simple and useful enough to justify its application in our first approach to an effective parallel implementation of logic programming.

6 The Parallel Computer And Database Machine

We now consider the storage of the database and rules, and the design of a processor for B-LOG. This parallel computer will have one or more database machines, and one or more processors, connected by some interconnection network. We consider the database machine first, then the processor, and finally we comment on the interconnection network.

Although we have worked on a powerful database processor (CASSM [5]), we consider that a disk-only based processor will be too slow for the evaluation of the heuristic described in the previous section. Still, the database will necessarily be large. Even the storage of rules will take a lot of space, as pointers to other rules and facts will be stored for each rule in a similar way to an inverted file. An immediate consequence of this fact is that there is little reason to have a separate database for rules and for facts as in PRISM [3]. A compromise solution uses data base machines to do some of the retrieval of portions of the graph, while fast processors do the main processing in the heuristic search inside this local subgraph. This is illustrated in figure 5.

In that figure we can see how the database (that is, the graph) is partitioned into a number of database processors (semantic paging disks, described below). There are also a number of processors with local memories, which contain copies of small subsets of the global graph. These processors use these subsets to work on their portions of the search tree. When a new node is needed for expansion the semantic paging disks will provide the appropriate subset of the graph, while the minimum seeking network will select the most likely candidate taking into consideration the current set of weights. At any given point in time, the search tree is distributed over a number of processors, each of them working on different parts of it, and the database is distributed over a number of semantic paging disks that search concurrently for new nodes for expansion. If a processor finds its chains to greater bounds than the other processors, it can stop its work on the subtree in it and transfer another chain with lower bounds into it, as the top processor does in figure 5.

Thus, the basic task of the database machine is to store a graph, implemented using pointers, and to extract a subgraph consisting of some selected nodes and all nodes within some Hamming distance of the selected nodes. We have described a "semantic paging disk" [12] (SPD) that works on pointers, and is well suited to this approach. Data so extracted is included into the processor memory, as in a paging scheme in virtual memory. However, rather than organizing data in fixed size pages, data is semantically organized in terms of a graph, and a page is a subgraph defined by the state of the process at run time. The earlier paper describes such a machine. However, since then, cheap RAM has made a cache attractive in a disk system, and the use of a cache simplifies the design of this machine. We summarize this cache oriented SPD below (see figure 6).

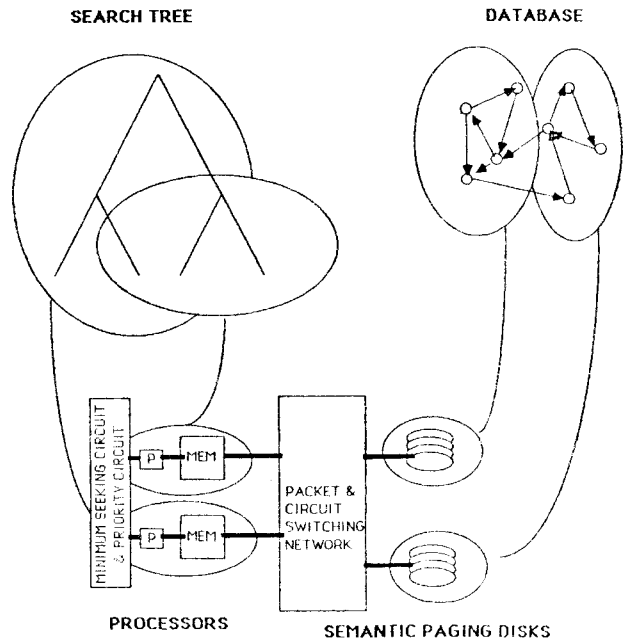

Figure 5: Parallel Computing Environment

The SPD consists of one or more search processors (SP). Each SP has one or more tracks (a moving head disk would have all the tracks on a surface in an SP), a read-write head and associated drivers and amplifiers, a random access memory (a cache) able to hold a track's data, and logic to implement the actions described below. The blocks of the linked list are stored in variable length records, which have a block number that is defined to be the number of blocks above it in the track. The contents of a block contain some data (possibly ASCII characters) and named and weighted pointers (name, pointer to another block, weight) as in figure 4. The pointers are the block numbers of the blocks pointed to. As the cache is loaded from a track, the location in the

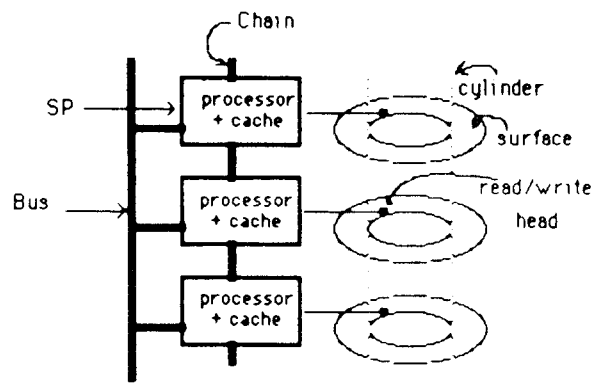

Figure 6: A Semantic Paging Disk (SPD)

cache of the beginning of each block is noted and tags for marking the blocks are provided in a table. The logic is able to

1. Search the data in a block associatively and mark the blocks.

2. Follow all pointers, or only pointers with specified names, from marked blocks to other blocks and mark them.

3. Output, replace, insert and delete words in a marked block.

Using (1), we can find some blocks. Using (2) N times successively, we can find all blocks within Hamming distance N from the nodes we found. Using (3), we can output or update the database.

Where more than one SP is used, they can work independently (MIMD mode) or interdependently (SIMD mode). In SIMD mode, all SPs work on the same track on their surface (a cylinder), and the tracks in a cylinder are presumed ordered in a chain. A global block number is defined for each record (block), and can be computed when the cache in each SP is loaded. It is the number of records above its record in the current track, plus the number of records in all the tracks above this track. The pointer becomes a pair (cylinder number, global pointer). If the pointer is to the cylinder that is cached, communication and hardware between the SPs can find which SP a global pointer is in, and the SP can mark the record. The associative search operation (1) and the pointer transfer (2) can be performed simultaneously in all SPs that are connected in SIMD mode. If the pointer is to another cylinder, pointer transfer is handled by saving the pointer until the other cylinder is loaded into the cache. The control of the SPM is simplified: all the external processor needs to know is which cylinder(s) to search on all SPs, not where the data is on such cylinders. Garbage collection between tracks in a cylinder can be done in the SPs without interacting with external processors. A paper more fully describing this device is in preparation.

The processor in turn will obtain data from the SPDs, storing it in its local memory. The design of the processor should avoid the "von Neumann bottleneck" even in the operation of its controller. In our approach, we propose to use an inference driven scheme for the execution of the controller: recall that in the CDC 6600 [2], a scoreboard is used to keep busy a collection of adders, multipliers and the like, and resolves the use of variables from one operation that are needed in another operation. We should build some specialized units, for example, to instantiate variables. When a unit has completed its operation, it should consult the scoreboard to determine what operation it can do next. The actual design of this units is presently one of our main areas of research(a) . The idea is to define a local interpreter of the B-LOG language in terms of production rules. We then implement each unitary action in a hardware unit and use a scoreboard to schedule their use. Note that a single processor will thus be multitasked, able to develop several chains of the search tree at one time. Also, the delays due to disk access can be compensated for by developing other chains that are not waiting for the slow disk. This may also be the correct design approach for an effective (pipelined) uniprocessor approach, and this is another point we are presently studying further.

One possible bottleneck that our preliminary analysis shows is that a multitasked processor will spend a lot of time copying data received from the disk, and data in its own memory, as new chains in the search tree are sprouted. This is a consequence of the very peculiar character of the logic variable, since most structure sharing schemes are difficult to implement in parallel [16]. Thus, the processor memory should be designed to write multiply, as well as singly in the normal sense of a random access memory. Using a shift register inside the memory, along side the address decoder, the shift register threading through each successive word in RAM, multiple writes can be effected. By setting several bits in the shift register (using the decoder), we can write the contents of all words that have a 1 in the shift register. We could then shift the whole bit pattern down one location so that we can write the next word of each copy in one memory access. Continuing this operation, a block of data can be copied many times into memory for example to assist in multitasking.

As far as the interconnection network (connecting processors together and processors to database machines) is concerned, it should support bursty traffic from the database machines, and a circuit to determine the minimum value of several bounds produced by the processors. Bursty traffic is well handled by a network that uses packet switching to find paths, and circuit switching to move the data. This scheme is used in the CEDAR machine [8], according to Gajski. The sorting network suggested in section 2 is probably lightly used since a processor will have to perform a lot of work, and wait for slow I/O, between times that it uses the sorting network; instead, a circuit that determines the minimum, and a priority circuit to arbitrate among several waiting processors to determine which will process the minimum, would be adequate. Several circuits have been presented which can very efficiently find a minimum, one of which is a tree where each node selects the minimum of its descendants and passes that to its parent. A priority circuit can be implemented in a tree-shaped carry-lookahead circuit. A linear cost non-rectanguiar banyan can implement these mechanisms, and this is another of our current subjects of research.

The parallel B-LOG machine will thus appear to work in the following way. Each of N processors has the capability of supporting M tasks at the same time. Each processor keeps track of the weights of the chains it has found and is able to send the minimum bound into a minimum seeking network. Initially, one processor is given the initial query, which it sends to the SPDs to page in part of the graph to work on. The other processors use the minimum seeking network to wait for some chain to work on. As chains become available, they are sent to the awaiting processors. The priority network assigns a minimum to just one awaiting processor at a time. Thus, initially, the tree is searched breadth-first to get all processors working. This is done with only one task in each processor. After all processors have been given work to do, the minimum seeking network keeps track of the lowest bound of the chains not yet expanded. Ignoring communication costs, when a task completes its extension of a chain, it will acquire a new chain, as determined by the minimum seeking network, and work on it. However, this would generate excessive traffic through the interconnection network. We choose a value D, which reflects the communication cost of moving a chain. If the minimum over the network is D lower than the minimum

(a)After this paper was submitted for review Graham [9] proposed a similar concept.

of the tasks in a processor, the freed task would acquire the chain through the network, else it would work on the minimum chain given by some task in its own processor. D can be modified at run time, based on the measured communication overhead.

7 Exploiting Other Sources of Parallelism

We have presented our model based on the OR-tree representation of the Logic Programming search space introduced in section 3. In this sense our model represents an intelligent, bound guided implementation of OR-parallelism as defined by Conery and Kibler [4]. OR-parallelism is specially effective in speeding up non-deterministic programs, specially when more than one solution is needed. Search-parallelism is also implemented very effectively in the semantic paging scheme through the use of several SPD's working concurrently.

Another source of parallelism present in logic programs is AND-parallelism, that is, the concurrent execution of several goals within a clause body. AND-parallelism can be very effective in speeding up highly deterministic programs, specially if only one solution is needed. In general our model could also support AND-parallelism, but some special cases have to be taken into account.

Its inclusion is a relatively simple issue for conjunctions of goals which do not share variables and the same basic model described in the preceding sections can be used in this cases. Unfortunately this case is not as common as desired. Calls which share variables can be executed in sequence using the same scheme as Prolog. Alternatively a join algorithm can be applied. In our implementation a highly efficient semi-join algorithm can use the marking capabilities of the SPD's.

Also, at run time, many of the dependencies apparent at compile time can disappear because of the particular bindings of the variables at the time the call is made [6]. A run-time analysis can thus grant the maximum level of parallelism but the support needed can result in high overhead [7]. An alternative to this approach is to do extensive data dependency analysis at compile-time. Similar extensions are being considered in our model but are left for future implementation.

8 Conclusions

We have presented a methodology for parallel processing of logic programming, and sketched a parallel processor that could implement this methodology. Further work is in progress in most areas: definition of the semantics of the B-LOG language, design of the data structures and processing units needed in its implementation, design of the database machine and the interconnection network, and evaluation of alternative bound generation and updating algorithms. Several schemes for supporting AND-parallelism are also being considered. In addition, we are analyzing specific applications in the context of this methodology. The validity of our present approach, specially in the choice of the bound generation and update algorithm, can be verified in this way.

B-LOG offers an alternative to Prolog's sequentially oriented depth-first search, without giving up completeness by incorporating control annotations. At the same time, it tries to overcome the combinatorial explosion of other search strategies which are not driven by heuristics. We thus feel that it can be the foundation for a resolution-complete and effective parallel implementation of logic programming.

References

[1] K. E. Batcher.
Sorting Networks and Their Application.
AFIPS Conf. Proc. 32:307-314, 1968.

[2] James E. Thornton.
Parallel operation in the Control Data 6600.
Computer Structures: Readings and Examples.
McGraw-Hill, 1971.

[3] U.S. Chakravarthy, S. Kasif, M. Kohly, J. Minker, and D. Cao.
Logic Programming on ZMOB: A Highly Parallel Machine
Proceedings of the 1982 Conference on Parallel Processing :347-349, 1982.
IEEE Press, New York.

[4] J.S. Conery and D.F. Kibler.
Parallel Interpretation of Logic Programs.
In Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture, pages 163-170. October, 1981.

[5] Copeland, G.P., Lipovski, G. J., Su, S. Y.
The architecture of CASSM: a cellular system for non-numeric processing.
Proceedings of the 1st. annual symposium on computer architecture :121-128, 1973.

[6] J.S. Conery.
The AND/OR Process Model for Parallel Interpretation of Logic Programs.
PhD thesis, The University of California at Irvine, 1983.
Technical Report 204.

[7] Doug DeGroot.
Restricted And-Parallelism.
Int'l Conf. on Fifth Generation Computer Systems, November, 1984.

[8] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh.
CEDAR- A Large Scale Multiprocessor.
Proc. of the Int'l Conf. on Parallel Processing :524-529, 1983.

[9] C. J. Graham.
Providing Architectural Support for Expert Systems.
Computer Architecture News 12(5):12-19, 1984.

[10] Kowalski, R.A.
Predicate Logic as a Programming Language.
Proc. IFIPS 74 , 1974.

[11] Kumar, V. and Kanal, L. N.
Parallel Branch-and-Bound Formulations for AND/OR Tree Search.
IEEE transactions on pattern analysis and machine intelligence 6:768-778, November, 1984.

[12] G. J. Lipovski.
Semantic Paging on Intelligent Disks.
Fourth Workshop on Computer Architecture for Non-numeric Processing , 1978.

[13] Pereira, L.M., F. C. N. Pereira, and D. H. D. Warren.
User's Guide to DECsystem-10 Prolog
Dept. of Artificial Intelligence, Univ. of Edinburgh, 1978.

[14] Harold S. Stone.
The Average Complexity of Depth-First Search.
IBM Research Report #RC 10717 (#48044) 9/6/84.

[15] Sunichi Uchida.
Towards a New Generation Computer Architecture: Research and Development Plan for Computer Architecture in the 5th. Gen. Computer Project.
Technical Report TR-001, ICOT-Institute for New Generation Computer Technology, July, 1982.

[16] D.S. Warren.
Efficient Prolog Memory Management for Flexible Control Strategies.
1984 International Symposium on Logic Programming, Atlantic City, pages 198-203. IEEE Computer Society Press, Silver Spring, MD, February, 1984.