Parallel AI Algorithms for Fine-Grained Hypercube Multiprocessors*

Frank Dehne [1], Afonso G. Ferreira [2], and Andrew Rau-Chaplin [1]

[1] *Center for Parallel and Distributed Computing*
School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6

[2] *Laboratoire de l'Informatique du Parallelisme - IMAG*
Ecole Normale Superieure de Lyon, 69364 Lyon cedex 07, France

Abstract. We describe recent advances in the design of parallel Artificial Intelligence (AI) algorithms for *fine-grained* hypercube multiprocessors like the Connection Machine. Two AI problems are considered: (1) Parallel branch and bound, and (2) the design of a parallel frame based knowledge representation system.

Key Words. Branch and Bound, *Fine-Grained* Hypercube Multiprocessor, Knowledge Representation, Multiple Tree Traversal, Parallel Architectures for AI.

1 Introduction

In this paper we describe recent advances in the design of parallel *Artificial Intelligence* (AI) algorithms for *fine-grained* hypercube multiprocessors like the Connection Machine. These studies are part of a joint project of the *Center for Parallel and Distributed Computing* (PARADISE) at *Carleton University* and the *Northeast Parallel Architecture Center in Syracuse*, NY (which granted us access to their *CM1* and *CM2* type *Connection Machines*).

A hypercube multiprocessor is a set P_0, ..., P_{p-1} of p processors connected in a hypercube topology; i.e., P_i and P_j are connected by a communication link if and only if the binary representations of i and j differ in exactly one bit. In a hypercube, there is no shared memory. The entire storage capability consists of local memories, one attached to

* The first author's research is partially supported by the Natural Sciences and Engineering Research Council of Canada (Grant A9173). The second author is currently on leave from the University of Sao Paulo (Brazil), project BID/USP, and partially supported by CAPES/COFECUB (Grant 503/86-9). The third author's research is partially supported by the Bell-Northern Research *Graduate Award Program*. Support from the Northeast Parallel Architecture Center (Syracuse, NY), which granted us access to their Connection Machines, is gratefully acknowledged.

each processor. A hypercube multiprocessor is called *fine-grained* if all local memories have a very small size permitting each processor to store only a constant number of data items. In such a system, the number, p, of processors has to be of the same order of magnitude as the number of data items to be stored and manipulated. Hypercubes with a small number (less than 1,000) of more powerful processors are called *coarse-grained* hypercubes. The *Connection Machine* is a famous example of a fine-grained system with a large number (64K) of relatively small processors. The Intel iPSC/2 is a well known example of a coarse-grained hypercube.

Although parallel solutions for AI problems have been extensively studied for *coarse-grained* parallel systems, *fine-grained* systems have only been studied to a very limited extent. The main advantage of coarse-grained systems is that they allow a relatively easy splitting of problems into subtasks, each of which is then executed fairly independently on a powerful processor which resembles a standard sequential machine. On fine-grained systems, these strategies can not be applied due to the limited amount of memory and processing power at each processor. Instead, it is necessary to share tasks in a truly cooperative way among a large number of small processors.

While the *Connection Machine* has been primarily designed for AI applications, it is surprising that for numerous AI problems there exist no parallel solutions for *fine-grained* hypercubes. In this paper, we describe *fine-grained* parallel solutions to two such problems that are of central importance in AI: (1) Parallel branch and bound [DFR1], and (2) the design of a parallel frame based knowledge representation system [DFR3].

Branch-and-bound (B&B) search methods such as A* search, Alpha-Beta search, depth-first and best-first search are used in areas such as VLSI design, theorem proving, linear programming, chess playing, and many other Artificial Intelligence and Operations Research applications ([Win], [Nil]). Since most of the addressed problems are NP-hard, and therefore the size of the search space grows exponentially, many researchers have studied the parallelization of B&B methods (e.g. [HD], [New], [MC], [UYII]). So far, studies have been aimed at using coarse-grained multiprocessors, in particular coarse-grained hypercubes (e.g. [LW], [Qui], [AC], [AM], [Fel], [MTM], [PW], [SGB]) for parallel B&B implementations. These methods aim at splitting the tree searched by the B&B method into subtrees, and having subtrees searched in parallel by different processors. The essential differences among the proposed coarse-grained algorithms are how they deal with the two major problems arising in such an approach: how to balance the sizes of the subtasks assigned to individual processors (load balancing), and how to distribute (with minimal computational overhead) global information to be shared by all processors. Since all these methods require each processor to store (at least) the back-up path of the node in the tree currently being examined, they can not be applied to a fine-grained hypercube. Furthermore, these solutions often make global knowledge available only to a limited extent or with some delay, which results in some unnecessary search steps (compared to the equivalent sequential method). This is reasonable for the relatively small number of parallel search processes in a coarse-grained system, but becomes intolerable for fine-grained hypercubes.

In Section 2 of this paper we describe an efficient B&B algorithm for fine-grained hypercube multiprocessors that solves both of the above problems.

Section 3 is concerned with the design and implementation of a parallel frame based knowledge representation system [BW, BS, FK, Int, Min]. Knowledge representation is an essential part of AI [New, Woo], and several researchers have studied parallel architectures for implementing knowledge bases [Bic, DM, EH, FHS, HGC, IH, MT, Ric, Sap, Uhr, WL]. The parallel knowledge representation systems presented in the literature have, however, either been based on special purpose parallel architectures or support only the parallelization of one query at a time. The latter implies the (economically infeasible) dedication of a massively parallel computer to one single user (e.g. [EV, IH]).

We consider a *single inheritance frame based knowledge representation system* which answers elementary queries such as top-down and bottom-up inference and assert/retract queries [EH]. The problem of implementing such a system on a *fine-grained* hypercube reduces to multiple parallel traversals of the knowledge base, and we study the problem of efficiently implementing such traversals on a hypercube. In particular, the problem of traversing in parallel multiple randomly chosen subtrees of an inheritance hierarchy is studied. In addition to theoretical time complexity bounds, experimental results obtained from extensive testing (200,000 randomly generated inheritance hierarchies of 16,000 nodes and with up to 16,000 queries, each) of a prototype implementation are presented. They show that a nearly optimal, 98%, processor utilization is obtained for a 70% load factor (number of processors divided by number of queries). In our experiments, the utilization never dropped below 75%, regardless of the load factor and other parameters. Our system adapts flexibly and automatically to varying work loads in a close to optimal way, providing a nearly constant product of response time and number of queries.

2 Parallel Branch and Bound on a Fine-Grained Hypercube

A branch and bound algorithm searches within the space of all the feasible solutions for a given combinatorial search problem. These feasible solutions are usually seen as a search tree S over the solution space. In parallel B&B algorithms, the search in S for an optimal or satisfactory solution is performed concurrently at several nodes of S. These nodes are referred to as *active nodes*. The subtree T of S defined by the union of the back-up paths of all active nodes is defined as the *current back-up tree* as (see Figure 1). Note that, we allow any node of the current back-up tree to be an active node, which allows e.g. the implementation of best-first search. For the remainder, *m* will refer to the size of the current back-up tree.

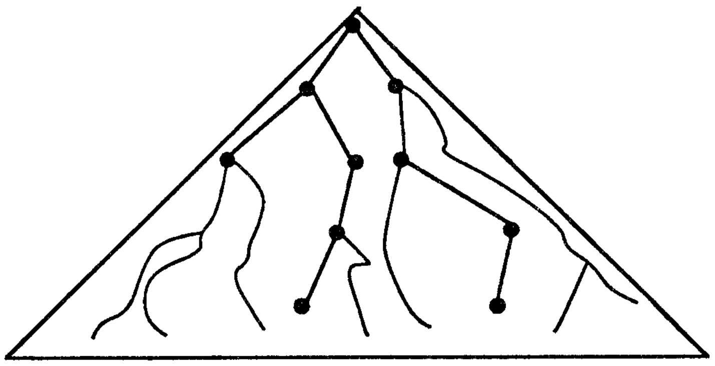
Figure 1. A Current Back-Up Tree (Bold Nodes).

In this section, we will describe a parallel B&B algorithm for fine-grained hypercube multiprocessors. Our method stores the current back-up tree such that each processor needs to store only a constant amount of information. At each iteration of the algorithm, all active nodes of the current back-up tree decide whether they need to create new children, be pruned, or remain unchanged. Based on these decisions, the algorithm described below updates the current back-up tree and distributes global information in $O(\log m)$ steps. This method also includes a dynamic allocation of search processes to processors, such that the work load balancing problem is solved optimally.

Consider a B&B algorithm (e.g., A*, Alpha-Beta, Hill-Climbing and Best-First) defined by five rules (see [AC]):

- *Cost Rule*: Given a leaf node in the search tree and it parent's cost, this rule defines the cost up to and including the leaf.
- *Bounding Rule*: Given a node in the search tree, this rule returns 1 if the node is no longer feasible (i.e., should be deleted) else returns a 0.
- *Selection Rule*: Given a node in the search tree, this rule returns an integer specifying the number of children this node should create in this iteration.
- *Expansion Rule*: Given a node, an integer specifying the number of children to be created, and a pointer to where they should be created, this rule creates the new children.
- *Termination Rule*: Given a set of global information, this rule returns true if a satisfactory solution has been found or all possibilities have been explored.

We assume that every processor $PE(i)$ has a constant size register $n(i)$ to store one node of the current back-up tree T. The current back-up tree T is stored on the hypercube as follows:

Consider the level ordering of the nodes of T as shown in Figure 2. Each node v of T is stored in register $n(i)$ of processor $PE(i)$, where i is the index of v with respect to the level ordering of T.

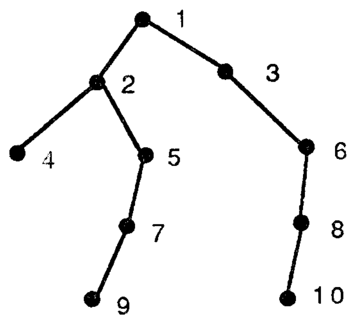

Figure 2. Level Ordering of the Current Back-Up Tree

The B&B algorithm consists of one *main loop* which iterates until the termination rule returns true. The back-up tree T starts as a single root node stored in register $n(0)$ of processor $PE(0)$. In each iteration of the main loop, the algorithm updates the tree by adding new nodes according to the selection and expansion rules, while at the same time pruning the tree using the cost and bound rules. A single pass through the main loop consists of five steps.

First, the evaluation rule is used to calculate the cost function for all the nodes that were created in the last iteration. The evaluation rule may also maintain global knowledge concerning the progress of the search. The maintenance of such global knowledge does not increase the fundamental time complexity of the algorithm, assuming that such global knowledge can be calculated by a global minimization and broadcast.

The second step of the main loop uses the bound rule to identify nodes to be deleted in this iteration. Any particular implementation of the bound rule must never delete a parent node without also deleting its children. The parallel bounding of a large number of nodes is a particularly effective aspect of our fine-grained B&B algorithm, since with very low overhead a large number of nodes in the back-up tree may be deleted and the data structure compacted.

In the third step, the selection rule is used to assign to each node $n(i)$ an integer that indicates the number of children the node needs to create during this iteration. The selection rule may also use global information to instruct each node exactly which of its possible children should be created.

Note that, except for the maintenance of global knowledge, the above three steps involve only $O(1)$ local operations at each processor of the hypercube.

Step 4, referred to as procedure *UpdateTree*, represents the main part of our algorithm. In Steps 1 through 3 all the information needed to extend the back-up tree to its next state has been collected. Now, the back-up tree must be transformed by deleting the bounded nodes and creating space for the new additional nodes. Below, we will discuss procedure UpdateTree in more detail.

Finally, in Step 5, the data for the new leaf nodes are created using the expansion rule. At this point each node has information about how many new children should be created, and about the address of the free processors allocated for them by Procedure *UpdateTree*. Therefore, Step 5 can be implemented with an $O(\log m)$ time complexity by using a distribute operation of [NS] to copy the data of each node creating new children into the processors storing these children, and then creating for each child in parallel its actual data set locally at the respective processor.

As indicated above, the core of our parallel B&B algorithm is Procedure *UpdateTree* which updates the back-up tree after infeasible nodes have been deleted and new nodes have been added in the previous step. The main problem here is that for the new back-up tree, the nodes must again be stored by level number (to allow maximum storage space utilization and performance). This makes it necessary to compute for each node of the new back-up tree its new address and relocate the nodes to obtain the correct storage scheme. This relocation of tree nodes also provides an optimal task allocation mechanism for solving the load balancing problem.

Procedure *UpdateTree* calculates for each node where its children, if any, should be located on the hypercube after the update is completed. Consider the node p in Figure 3. The address (processor number) of the first child of p, after the update is completed, is the sum of two numbers x and y, defined as follows: x is the number of nodes in the updated tree up to (and including) p's level, while y is the number of children of nodes to the left of p in p's level. The sum of these two numbers indicates the position of p's first child in the new back-up tree. Each parent then broadcasts this information to all of its children. Since all nodes may move, all parent nodes must also broadcast their own new address to their children so that the children can update their parent pointers. When these steps have been completed, all non-deleted nodes are routed to their new locations. This route operation will leave the necessary spaces for the new children that are to be created by the expansion rule called in the main procedure.

Procedure *UpdateTree* is composed of 6 steps. In *Step 1*, the number of children at each node is recalculated to account for the nodes that have been marked for deletion and the new nodes that are to be added. This operation is performed by first performing a partial sum over the number of alive nodes within each block of nodes that share the same parent. The result of the partial sum is then sent to each parent by its last child. Step 1 is completed by setting register *n.children*(i) of each node to the sum of its non deleted children and its new children to be created. In *Step 2*, the x value for each node is computed. The x value for any node at level k is one plus the sum of the number of children of all nodes up to (and including) the last node in level k -1. The x value for all nodes can therefore be calculated as follows. First, a partial sum on the number of children, *n.children*(i), of every node is computed and stored in register $x'(i)$; $x'(i)$ is then incremented by one to account for the root node. Then, the last node in each block of nodes sharing the same parent sends its x' value to its parent node. In *Step 3*, the value of y for each node is calculated. This is performed by computing a partial sum on register *n.children*(i) for all nodes in the same level. In order not to count the children of a node in its own $y(i)$ count, *n.children*(i) must then be subtracted from the value provided by the partial sum operation. In *Step 4*, the new address of the first child of each node is computed and stored in register *newFirstChild*(i), using the $x(i)$ and $y(i)$ values previously calculated. This value is then broadcast to all the children of each node; since the back-up tree is stored on the hypercube in level order, all children of a node can calculate their new addresses by adding to the new address of their first sibling the number of other siblings that are before them. Now, every node has its new address. In *Step 5*, all nodes with children broadcast their new address to their children, so that the children can update their parent pointers. *Step 6* completes the *UpdateTree* procedure by moving every node to its new address.

It is easy to see that all of the above operations reduce to data movements described in [NS] Hence, our algorithm can update the current backup tree in time $O(\log m)$, using $O(1)$ memory per processor.

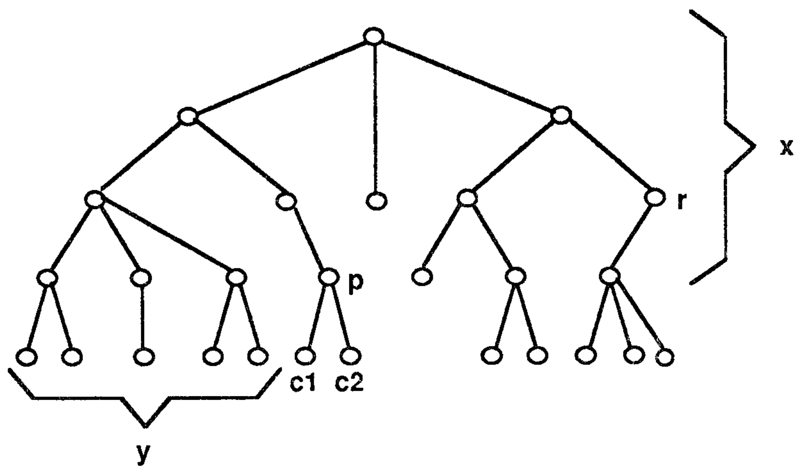

Figure 3. Illustration of the Address Calculation for the New Back-Up Tree.

3 Frame Based Knowledge Representation And Inference

3.1 Problem Definition

Semantic nets alternatively known as *frame-based systems* have been widely studied [Bic, BW, BS, DM, EH, FK, Int, Min, Sap, SBMC Tou], and several *knowledge representation* tools have been designed based on them [BW, BS, EH, FK, Int, SBMC].

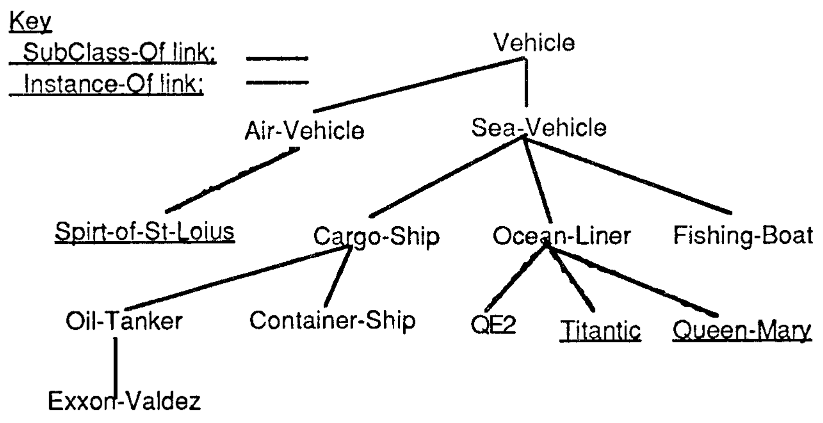

Figure 4. An Example of a Single Inheritance Hierarchy.

A frame language provides the designer of knowledge based systems with an easy way to describe the domain objects to be modeled and their relationships. In a single inheritance frame based system, *frames* are used to describe individual objects or a classes of objects, and these frames are organized in a tree like hierarchy. For example in Figure 4, the class *Ocean-Liner* is used as a prototype to describe all of the properties that are common to all ocean liners, such as the fact that they carry paying passengers. The instance *QE2*, on the other hand, is a frame that represents an individual instance of the class ocean-liner. It specifies knowledge about a particular ocean liner, the *QE2*, such knowledge might include the number of passengers that the liner caries, or the *QE2*'s transatlantic crossing time. Both classes and instances are represented by frames. Each frame consists of a series of *slots* each of them being used to represent a single fact about a particular class or instance. An important property of single inheritance frame based systems is that not all slot values of all frames are stored explicitly. A value assigned to a slot of a frames x is inherited by all frames in the subtree rooted at x, unless for those frames in the subtree which explicitly store for themselves (and all frames in their subtrees) another value for the same slot. Slot values obtained from this *inheritance* mechanism will be referred to as *implicit* slot values.

A *bottom-up inference query*, $q(X)$, is of the form "Does frame X meet conditions a through z". Queries like "What are the values of slots a through z of frame X?", "Is the Exxon-Valdez a vehicle with color black and current-location Alaska?", or "What is the weight the QE2?" are examples of bottom-up inference queries based on Figure 2. Bottom-up queries are always about a particular instance or class frame but, since we are using an implicit representation, may require the examination of all superclasses of that frame.

A *top-down inference query*, $q(X)$, is of the general form "Identify all frames in the subtree (of the inheritance hierarchy) rooted at X such that conditions a through z are true". For example, "Identify all instances of Sea-Vehicles with weight > 1000 tons and paying passengers = 0", or "Identify all classes who are subclasses of Vehicle and have less-than 10 paying passengers" are top-down inference queries based on the hierarchy in Figure 2.

Assert queries are queries that add knowledge to our representation. There are three basic types of assertions: assertion of a new slot value, assertion of a new slot (with initial value), or lastly, assertion of a new class or instance frame (complete with slots and values). *Retract queries* fall into three analogous types: retraction of a slot value, retraction of a slot, and retraction of a class or instance frame.

3.2 Multi-Way Search on a Tree

We introduce some notations and previous results on hypercube algorithms which will be used in the remainder.

Let $T = (V, E)$ be a tree of size k, height h, and out-degree $O(1)$, and let U be a universe of possible search queries on T. A *search path* for a query $q \in U$ is a sequence $path(q)=(v_1, \ldots, v_h)$ of h vertices of T defined by a *successor* function f: $(V \cup \{\text{start}\}) \times U \Rightarrow V$ (i.e., a function with the property that $f(\text{start},q) \in L_1$ and for every vertex $v \in V$, $(v, f(v,q)) \in E$). A *search process* for a query q with search path $(v_1, \ldots, v_h)$ is a process divided into h time steps $t_1 < t_2 < \ldots < t_h$ such that at time t_j, $1 \leq j \leq h$, query q is *matched* with node v_i. A *match* of a query q and a node v_i at time t_j is defined as a situation where there exists a processor which contains a description of both, the query q and the node v_i. Note, however, that we do not assume that the search path is given in advance; we assume that it is constructed during the search by successive applications of the functions f. Given a set $Q=\{q_1,\ldots,q_m\} \subseteq U$ of m queries, $m=O(k)$, then the *multi-way search problem* consists of executing (in parallel) all m search processes induced by the m queries. In [DFR2, DR] it was shown that the multi-way search problem can be solved on a hypercube multiprocessor of size $\max\{k,m\}$ in time $O(\log k \operatorname{loglog}^2 k + h \log k)$. It follows from [CP, DFR2, DR] that for trees with unbounded out-degree, as well as arbitrary graphs, the time complexity increases to $O(h \log k \operatorname{loglog}^2 k)$.

3.3 Parallel Inference/Assertion/Retraction on Fine-Grained Hypercubes

An inference hierarchy with n frames is stored on a hypercube with n processors and a set of n inference/assert/retract queries is given which are to be executed in parallel by the hypercube. Consider the *level numbering* of the frames of an inheritance hierarchy (see Figure 2). For the remainder we will assume that each frame with level number i, together with its links and explicit slot values, is stored at processor P_i. We will now present, separately, algorithms for answering the different types of queries. Note, however, that these processes are to be executed concurrently. Bottom-up and assert/retract queries are fairly easy to process; the main problem is to efficiently process top-down inference queries.

Consider a set of $m \leq n$ bottom-up inference queries where each query is stored at one arbitrary processor. For the remainder, *frames*(i) and *query*(i) refer to the query and frame (currently) stored at processor $PE(i)$. Such queries can be answered in parallel as follows: Each query is matched with the respective frame it refers to (see Section 3.2). Then, using the multi-way search method of Section 3.2, all queries are routed upwards through the inheritance hierarchy until they reach the root. During this routing phase, for each query, all the requested explicit slot values are collected. We obtain that all $m \leq n$ bottom-up inference queries can be answered, in parallel, in time $O(\log n \operatorname{loglog}^2 n + h \log n)$ [or $O(h \log n \operatorname{loglog}^2 n)$ if frames can have an unbounded number of children], where h is the height of the inheritance hierarchy.

For assert/retract queries, these need to be matched with their respective frames. If only slot values of these frames need to be changed, this is straight forward. If new frames have to be added, or existing frames have to be deleted, this can be accomplished in essentially the same was as in Procedure UpdateTree outlined in Section 2. We obtain that all assert/retract queries can be answered in time $O(\log n \operatorname{loglog}^2 n)$.

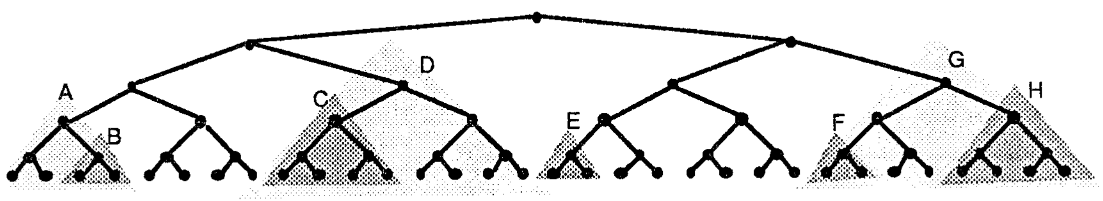

Figure 5. Multiple Subtrees To Be Searched For Multiple Top-Down Inference Queries.

What remains to be solved is to process the top-down inference queries. Consider a set of $m \leq n$ top-down inference queries where each query is initially stored at one arbitrary processor. First we need to answer, for each query $q(X)$ referring to a frame X, the respective bottom-up inference query. That is, each query $q(X)$ has now for all the slots which are specified in it explicitly stored the implicit values at frame X. In the following we study the remaining steps necessary to answer top-down queries, i.e. to search, for every $q(X)$, the subtree rooted at X. Note that, the subtrees are arbitrarily distributed over the inheritance hierarchy (as e.g. trees A through H depicted in Figure 5) and that several queries may search the same subtree. Our algorithm for solving this problem is shown in Figure 6.

```
1)  Match each query q(X) with the node (frame) X it refers to.
2)  Split each query q(X) into two tokens, a search token search-token and a control
    token control-token. Each token contains a copy of the original query. Each search
    token at node X is responsible for searching the subtree rooted at the first child of X;
    each control token at node X is responsible for searching (or having searched) the
    subtrees rooted at the other children of X.
3)  As long as there is still a processor PE(i) storing an unanswered query q(X), repeat
    the following:
    3a) Count the number F of free processors. A free processor is a processor that is
        currently not supporting any search token.
```

3b) Each token t calculates the number $a(t)$ of assistants it could currently use. A Search token can always use as many assistants as there are unsearched subtrees at the node (frame) it is current currently visiting. F new search tokens (assistants) are created and matched with the existing (search and control) tokens in order of the level numbering of their current nodes (frames), each receiving $a(t)$ assistants until all new tokens are distributed.

3c) For each token that has been allocated assistants in Step 3b, assigns a child whose subtree has not been searched yet to each assistant, match the assistants with those children, create for each a corresponding control token, and have them search the respective subtrees.

3d) Each search token examines its current node.

3e) Use multi-way search to advance all search tokens by one step in their preorder traversal.

Figure 6. Hypercube Algorithm for Multiple Tree Traversal.

After each query has been matched with the respective node (Step 1), a search token and control token are created for every query (Step 2). Each search token, for a query $q(X)$, traverses (independently and in parallel) in preorder the subtree rooted at node (frame) X and determines the answers to be reported. Each control tokens remains at the root of the respective subtree to be traversed, indicates to the respective search token the end of its traversal, and creates new assistant processes in the same way as search tokens do. (Note that, the number of control tokens never exceeds the number of search tokens.) The main idea leading to a near optimal speedup (as will be shown below) is to re-use processors released by queries which need to traverse smaller subtrees to improve the performance of the search processes for the larger subtrees. This rescheduling mechanism is outlined in Steps 3a-3c. After each "round", i.e. parallel advancement of all search tokens by one edge in the preorder of their subtrees, processors from finished traversal processes are given to unfinished traversal processes. Every token determines, from the outdegree of the node (frame) it is currently visiting, how many "assistants" it could currently utilize, i.e. ask them to search those subtrees independently and in parallel. For the distribution of available processors, the following heuristic is used: higher priority is given to those search tokens with smaller level number, i.e. tokens that have (in the expected case) the largest subtrees still to be searched.

For observing the correctness of the algorithm note that, for every query $q(X)$ the entire subtree rooted at node (frame) X is searched, and that at every time a node (frame) is examined, all inherited values are present.

In the case of bottom-up inference and assertion/retraction it was possible to get a worst case bound on the algorithm's time complexity. Due to the rescheduling procedure, the performance analysis for the top-down inference algorithm is more complicated. Advancing all tokens one step along their path can be executed on a hypercube multiprocessor of size n in time $O(\log n)$ (if the number of children of each node (frame) is bounded by a small constant) or in time $O(\log n \text{ loglog}^2 n)$ (for unbounded number of children) [CP, DFR2, DR]. The problem lies in determining the number of such parallel steps required by our algorithm, since at the heart of the method is a heuristic rescheduling scheme that reallocates processors to queries. The challenge is to quantify how effective this reallocation technique is.

In order to test our mechanism, we have implemented a prototype system and have performed extensive tests using randomly generated hierarchies and sets of top-down inference queries. We considered the following input parameters:

n = number of nodes (frames) = number of processors,
m = number of queries,
k = max. number of children of a node (frame).

Figure 7 shows the result of our experiments for n=16,000. The graph on the left depicts results for hierarchies with unbounded k, while the graph on the right shows results for hierarchies with a small value of k (k=8). The x-axis in each diagram represents m, the number of queries, ranging from 1 to 16,000 in 1% increments. For each value of m, 1000 experiments where performed, each with a new randomly generated hierarchy and set of queries. The two curves show the average number of parallel steps as well as the average speed up. The speed up was measured by comparing the number of parallel steps with the total number of steps necessary for sequentially processing the same query set on the same hierarchy. It measures the utilization of the massive parallel architecture and, as our results show, a nearly optimal utilization is obtained for a 75% load factor (number of processors, n, divided by number of queries, m). The utilization never dropped below 75%, regardless of the load factor and other parameters.

The shape of the curves in Figure 7 can be explained by two opposing effects. If there are only a few queries to be processed (small load factor), these can not immediately request enough assistants (due to the constraints of the hierarchy) to utilize all processors. On the other hand, it is important for large subtrees to receive assistants early in the traversal process. Hence, if the number of queries is close to the number of processors, there are no (or only very few) assistants available until the smaller trees have been traversed. Therefore, it becomes likely for the larger trees, that late arriving assistants can not be efficiently applied to the traversal.

4 Acknowledgements

We acknowledge the assistance of Alain Bonopera, Kevin Bourgeault, Vincent Ho, Joel Lucuik, and Desmond Weong in implementing the multiple tree traversal prototype and multi-way search.

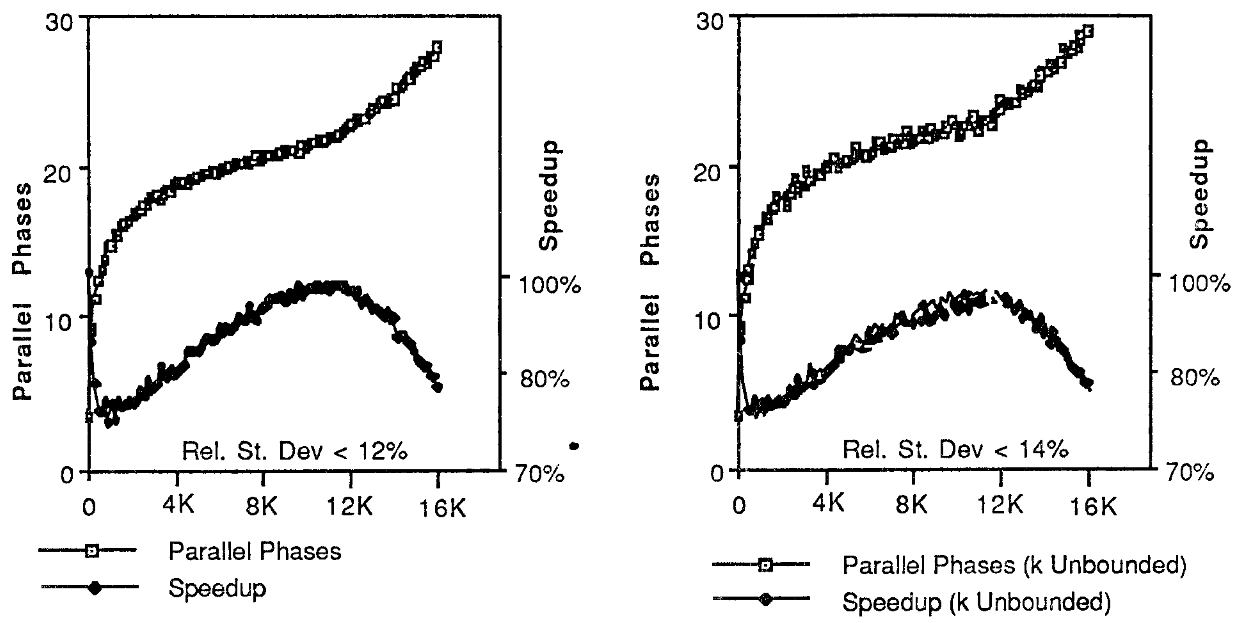

Figure 7. Experimental Results for Top-Down Inference.

5 References

[AC] S.Anderson and M.C.Chen, Parallel branch and bound algorithms on the hypercube, in Hypercube Multiprocessors 1987, M.T.Heath, ed., SIAM Press, Philadelphia, PA, pp 309-317

[AM] T.S.Abdelrahman and T.N.Mudge, Parallel branch and bound algorithms on hypercube multiprocessors, in Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, 1988, G.Fox, ed., ACM Press, pp 1492-1499

[Bic] L. Bic, Processing of semantic nets on dataflow systems, Artificial Intelligence, Vol. 27, 1985, pp. 219-227.

[BS] R. J. Brachman and J. G. Schmolze, An overview of the KL-One Knowledge Representation system, Cognitive Science, Vol. 9, No. 2, 1985.

[BW] D. G. Bobrow and T. Winograd, An overview of KRL, a knowledge representation language, Cognitive Science, Vol. 1, 1977, pp. 3-46.

[CP] R. Cypher and C. G. Plaxton, Deterministic sorting in nearly logarithmic time on a hypercube and related computers, to appear in Proc. ACM Symposium on Theory of Computing, 1990.

[DFR1] F. Dehne, A. Ferreira, and A. Rau-Chaplin, Parallel branch and bound on a hypercube multiprocessor, in Proc. IEEE Int. Workshop on Tools for Artificial Intelligence, Herndon, VA, 1989, pp. 616-622.

[DFR2] F. Dehne, A. Ferreira, and A. Rau-Chaplin, Parallel fractional cascading on a hypercube multiprocessor, to appear in Proc. Allerton Conf. on Communication, Control and Computing, Monticello, Ill., 1989.

[DFR3] F. Dehne, A. Ferreira, and A. Rau-Chaplin, A massively parallel knowledge-base server using a hypercube multiprocessor, Technical Report, School of Computer Science, Carleton University.

[DR] F. Dehne and A. Rau-Chaplin, Implementing data structures on a hypercube multiprocessor and applications in parallel computational geometry, to appear in Journal of Parallel and Distributed Computing.

[DK] M. Dixon and J. d. Kleer, Massively parallel asumption-based truth maintenance, in Proc. Proceedings of the Seventh National Conference on Artificial Intelligence, 1988, American Association for Artificial Intelligence, pp. 199-204.

[DM] J. G. Delgado-Frias and W. R. Moore, Parallel architectures for AI semantic network processing, Knowledge-Based Systems, Vol. 1, No. 5, 1988, pp. 259-265.

[EH] M. Evett and J. Hendler, Parallel knowledge representation on the Connection Machine, in Proc. Parallel Computing 1989, Leiden, The Netherlands, 1989.

[Fel] E.W.Felten, Best-first branch and bound on a hypercube, in the Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, 1988, G.Fox, ed., ACM Press, pp 1500-1504

[FHS] S. E. Fahlman, G. E. Hinton, and T. J. Sejnowski, Massively parallel architectures for AI: NETL, Thistle and Boltzman machines, in Proc. AAAI Annual Conference on Artificial Intelligence, 1983, pp. 109-113.

[FK] R. Fikes and T. Kehler, The role of frame-based representation in reasoning, Communications of the ACM, Vol. 28, No. 9, 1985, pp. 904-920.

[Gup] A. Gupta, Parallelism in production systems,, Carnegie-Mellon, 1986.

[HD] S.R.Huang and L.S.Davis, Parallel iterative A* search : an admissible distributed heuristic search algorithm, International Joint Conference on Artificial Intelligence 89, Preprint

[Hil] W. D. Hillis, The Connection Machine(Ed.), MIT Press, USA, 1985.

[IH] B. Israel and J. Hendler, A highly parallel implementation od a marker passing passing algorithm, Tech. Report No. CS-TR-2089, Dept. of Computer Science, University of Maryland, College Park, 1988.

[Int] IntelliCorp, KEE: Core Reference Manual, 1986.

[HGC] K. Hwang, J. Gosh, and R. Chowkwanyun, Computer architectures for artificial intelligence, Computer, Vol. 20, No. 1, 1987, pp. 19-27.

[LW] G.J.Li and B.W.Wah, Coping with anomalies in parallel branch and bound algorithms, IEEE Trans. on Comp., vol. c-35, no. 6, Jun 86, pp 568-573

[MC] A.Marsland and M.Campbell, Parallel search of strongly ordered game trees, Computing Surveys, vol. 14, no. 4, Dec 82, pp 533-551

[Min] M. Minsky, A framework for representing knowledge, in P. Winston (Ed.), In The Psychology of Computer Vision, McGraw-Hill, New York, 1975, pp. 211-277.

[MT] D. I. Moldovan and Y.-W. Tung, SNAP: a VLSI architecture for artificial intelligence, Journal of Parallel and Distributed Computing, Vol. 2, No. 2, 1985, pp. 109-131.

[New] A. Newell, The knowledge level, Artificial Intelligence Magazine, Vol. 2, No. 2, 1981, pp. 1-20.

[New] M.Newborn, Unsynchronized iteratively deepening parallel alpha-beta search, IEEE Trans. on PAMI, vol. 10, no. 5, Sep 88, pp 687-694

[Nil] Nils J. Nilsson, Principles of Artificial Intelligence, Tioga Publishing Co, 1980

[NS] D. Nassimi and S. Sahni, Data broadcasting in SIMD computers, IEEE Transactions on Computers, Vol. 30, No. 2, 1981, pp. 101-106.

[PW] R.P.Pargas and D.E.Wooster, Branch and bound algorithms on a hypercube, in the Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, 1988, G.Fox, ed., ACM Press, pp 1514-1519

[Qui] M.J.Quinn, Implementing best-first branch and bound algorithms on hypercubes multicomputers, in Hypercube Multiprocessors 1987, M.T.Heath, ed., SIAM Press, Philadelphia, PA, pp 318-326

[Ric] J. Rice, The advanced architectures project, Artificial Intelligence Magazine, Vol. Fall, 1989, pp. 27-39.

[Sap] P. S. Sapaty, A wave language for parallel processing of semantic networks, Comput. Artificial Intelligence, Vol. 5, No. 4, 1986, pp. 289-314.

[SBMC] M. J. Stefik, M. Bobrow, D. G. Mittal, and L. Conway, Knowledge programming in LOOPS: Report on an experimental course, Artificial Intellegence, Vol. 4, No. 3, 1983, pp. 3-14.

[SGB] K.Scwan, J.Gawkowski and B.Blake, Process and workload migration for a parallel branch and bound algorithm on a hypercube multicomputer, in the Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, 1988, G.Fox, ed., ACM Press, pp 1520-1530

[TM] F.S.Tsung and M.H.Ma, A dynamic load balancer for a parallel branch and bound algorithm, in the Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, 1988, G.Fox, ed., ACM Press, pp 1505-1513

[Tou] D. S. Touretzky, The Mathematics of Inheritance Systems(Ed.), Morgan Kaufmann Publishers, Inc, Los Altos, CA, 1986.

[Uhr] L. Uhr, Multi-Computer Architectures for Artificial Intelligence(Ed.), John Wiley & Sons, 1987.

[UYII] H.Usui, M.Yamashita, M.Imai, T.Ibaraki, Parallel searches of game trees, Systems and Computers in Japan, vol. 18, no. 8, 1987, pp 97-109

[Win] Patrick H. Winston, Artificial Intelligence, Addison-Wesley, July 1984

[WL] B. W. Wah and G.-J. Li, A survey on special purpose computer architectures for AI, SIGART News, Vol. 4, No. 96, 1986, pp. 28-46.

[WLY] B.W.Wah, G.J.Li and C.F.Yu, Multiprocessing of combinatorial search problems, Computer, Jun 85, pp 93108

[Woo] W. A. Woods, What's important about knowledge representation?, Computer, Vol. 15, No. 10, 1983, pp. 22-29.