RADC-TR-90-410
Final Technical Report
December 1990

DISTRIBUTED MULTI-AGENT PLANNING

Auburn University

William B. Day

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-410 has been reviewed and is approved for publication.

APPROVED: John J. Crowter

JOHN J. CROWTER
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: R Raposo

RONALD S. RAPOSO
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Final 4 Jun 90 - 6 Jul 90

4. TITLE AND SUBTITLE
DISTRIBUTED MULTI-AGENT PLANNING

5. FUNDING NUMBERS
C - F30602-88-D-0026,
Task B-0-3355
PE - 62702F
PR - 5581
TA - 27
WU - PF

6. AUTHOR(S)
William B. Day

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Auburn University
Department of Computer Science and Engineering
Auburn AL 36849-5347

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Air Development Center (COES)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-410

11. SUPPLEMENTARY NOTES
RADC Project Engineer: John J. Crowter/COES/(315) 330-3655
The prime contractor for this effort is Calspan-UB Research Center, PO Box 400, Buffalo NY 14225.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report investigates methods for extending the Distributed Datalog (DDL) system to allow OR-parallelism. A synopsis of implementation issues is given along with a summary of previous attempts at OR-parallalism. Two methods are examined for possible incorporation into the DDL. This effort also examines the application of the DDL to distributed multi-agent planning.

14. SUBJECT TERMS		15. NUMBER OF PAGES
Planning Distributed Planning Multi-Agent Planning	Distributed Systems Logic Programming PROLOG	36
		16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

1. Introduction

The task of this contract is to "design, develop, and execute experiments with OR-parallelism in the Distributed DataLog environment with a focus on its applicability/requirements for distributed multiagent planning systems". Since the effort was contracted to a month, only the design phase has been completed and is reported herein.

The Distributed DataLog (DDL) interpreter is a model for distributed logic programming. DDL is the distributed version of the Parallel Knowledge-based System, which is simulated in ADA on a VAX 11/780. Implementation details of the Parallel Knowledge-based system can be found in [10]. The DDL has been implemented in C on a network of four SUN workstations. The DDL interprets DataLog (a subset of Prolog) programs only. In DataLog arguments of literals must be either constants or variables; functors are not allowed. The DDL has implemented only AND-parallelism, but the model has allowed for future expansions to accomodate OR-parallelism and parallel knowledge base maintenance. This expansion of DDL to implement OR-parallelism is investigated in this report.

Section 2 consists of a synopsis of the implementation issues of OR-parallelism and of a summary of previous models of OR-parallelism. Section 3 then examines two of these methods which can be adapted to the DDL model.

In a distributed multiagent planning problem, a planner has to generate feasible actions that match each agent's specialities, constraints, and reasoning capabilities. Typically the planner refines goal components based on an agent's functional features and environmental constraints. When cooperation

among the agents is required, the planner must instruct agents about their goals and coordinate their actions by providing synchronization conditions. Because of the distributed nature of the environment, only the high level task assignment is communicated to agents. The detailed action planning and execution are conducted by each individual agent.

Section 4 examines the features of multiagent planning systems for potential OR-parallelism in the DDL, including plan representations, resource classification, condition classification, error identification and location, error recovery procedures, and truth maintenance.

2. Models of OR-parallelism

Classic OR-parallelism in logic programming is the simultaneous pursuit of a goal using two or more clauses. Thus, if a program holds these two clauses for the goal p(X):

p(X) :- q(X), r(X).

p(X) :- s(X), t(X).,

then there exists the potential for OR-parallelism. If there are also two processors available (one for each clause), then the two clauses can be used simultaneously to reduce the goal p(X).

The first issue that must be decided in constructing a model for OR-parallelism is allocation of the processes. In models which dynamically allocate processes to processors, availability of resources is controlled by a central scheduler or by advertisements from idle processors. Dynamic allocation of processes promises the maximum OR-parallelism but incurs a run-time expense which can outweigh its utility. Static allocation, in contrast, has low run-time costs, but may miss potential OR-parallelism if a procedure (clauses with the same head-predicate) is not well distributed among the processors.

There are also two semantic issues associated with OR-parallelism. The first issue is the intention of a procedure; i.e., what is the intended meaning of OR within a procedure? If the procedure consists of many facts, then the intention is usually the logical-OR; i.e., any one of these facts is equally acceptable for reducing the goal. The logic-OR intention is also possible when the procedure consists of rules as well as facts. When there are more clauses in a single procedure than processors, it is necessary to examine se-

quentially those clauses assigned to the same processor. The extreme case is where an entire procedure is assigned to a single processor. Then there is no potential for simultaneous OR-parallelism. This is the current state of the Distributed DataLog (DDL) interpreter.

When more than one clause of a procedure is assigned to the same processor, the order of examination may be criticial. Programmers of sequential Prolog are well aware of the top-to-bottom order in which clauses are examined, and consequently may arrange the clauses in an accommodating order. If such an arrangement is necessary, the intention is not that of the logical-OR but rather OR-then. The most common examples of the OR-then intention are in recursive predicates and in if-then-else predicates. In recursive predicates it is common to write the base case (usually a fact) before the recursive rule, knowing that infinite loops can be avoided in the depth-first search strategy if the base case is checked before the recursive rule. In if-then-else rules, it is common to separate possible cases by inserting a boolean comparison test as the first literal in the body of the clauses; e.g., p(X) :- X>0, etc. If the second clause's body begins with X<=0, then there is no intention of allowing the clauses to execute simultaneously since one is always doomed to fail. Frequently programmers omit this second boolean test since they know that the second clause's body will be correctly chosen when the X>0 test fails in the first clause. Such procedures must be specially marked by a programmer in order to avoid wasted effort in OR-parallelism. A simple syntatic addition to standard Prolog is the following. Suppose two clauses for p(X) are know by the programmer to have OR-then intent. Then the programmer may enter these clauses collectively as p(X) :- body1; body2. rather than as p(X) :- body1 and p(X) :- body2. Although both syntaxes are allowed in standard Prolog, one could restrict the notations so that the form with the semicolon

refers only to OR-then intentions and the form with two separate clauses refers to logic-OR intentions. OR-then clauses will always be allocated to the same processor and will never be solved in parallel.

The second semantic issue associated with OR-parallelism is an indication of the number of solutions that a procedure should produce. Assuming that an answer exists, the number of solutions sought can range from one to all, with the two extremes being the most common choices. The precedence of sequential Prolog to return one answer promotes the single answer option. But recently won favor for logic programming in database applications, has added credence to the all-solutions response. The all-solutions strategy has the distinct advantage over all other strategies in that it does not require the model to have a backtracking capability. The price it pays, however, is in overhead for collecting multiple solutions at each non-leaf node of the goal tree. Models for the intermediate cases which seek some but not all solutions of a goal must be able to backtrack and to collate multiple answers. Single solution strategies require only a backtracking ability.

2.1 Classical Models of OR-parallelism

Conrey [6] has categorized the models for OR-parallelism into the trinity of pure, process, and distributed search. All classes must deal with the combinatorial explosion of processes and with multiple variable environments. The combinatorial problem necessitates many processes being assigned to a single processor; consequently, some task management must be done. Parlog uses a syntatic extension to Prolog to control this growth. In Parlog clauses are terminated by either a semicolon or a period. The semico-

lons are used as markers for separting waves of OR-parallelism. For example, all clauses before the first semicolon are executed in parallel, then all clauses between the first and second semicolons are executed in parallel, etc. The multiple environment problem is exacerbated by the fact that variable reference are not limited to the two-level distance between choice points but may span many levels in the goal tree.

In pure OR-parallelism models, binding of shared variables is realized through structural extensions that identify and locate local copies. An efficient memory representation was presented by Ciepielewski and Haridi [5] in which each process has an associated directory tree of pointers to frames in the run-time stack, and the process writes only to a copy of a frame if unbounded variables are associated with the frame. Another approach is given in Lindstrom [13], in which all unbound variables are copied into the environment of each called clause. A variation of this delays the copying until necessary.

Methods for controlling the quantity of solutions include using the first solution only and exerting global control of the search through explicit directives for ordering the execution at choice points or through learned heuristics.

In contrast to pure OR-parallelism, Or processes, the second category of models, are "independent interpreters created to solve a goal statement consisting of exactly one literal" [6]. OR processes must communicate with their immediate ancestor and descendants. This communication compensates for run-time overhead by providing a means for controlling the spectrum of parallelism from sequential to pure.

Another model interpreter of the OR process class is that of Furukawa et al. [12], which is similar to the DDL interpreter. It is termed backup OR-parallelism, and it produces solutions sequentially. But as soon as one answer

if found and reported to the parent, the child process begins to look for the next solution.

The AND/OR process model of Conery [6] is a compromise of the extremes of reporting solutions. Here all solutions are found, stored in a buffer, and reported one at a time as demanded by the parent. Equivalently, the answers may all be reported once and stored in a buffer of the parent processor. The size of the buffer limits the number of accepted solutions. A child buffer size of one is analagous to the backup OR-parallelism model.

An advantage that the process models hold over the pure models is that small environmental structures are required for the process model. Pure models hold one large structure in a single memory, and a variable may be located anywhere in the structure.

The third model is the distributed search parallelism in which clauses are distributed among processors. PRISM [15] is an early system that used this approach. PRISM used a central problem solver, a database of facts, and a database of rules. Parallelism is obtained by broadcasting a goal from the central problem solver to the databases and acccepting a set of bindings. The system of Warren et al. [20] improves on PRISM by using a loosely coupled network of processors and broadcasting requests from a master node (potentially any node) to servant nodes. This provides a natural distribution for the problem solver. This model is similar to the DDL master/servant relationship.

By way of comparison, the pure models require little communication and coordination, but they assume a shared memory. They don't scale well and can potentially have high run-time overhead for task scheduling. Process models are more modular, but they incur higher communication overhead. This overhead may not be acceptable for small problems. The strength of the

distributed search models is found in the partitioning of a program, but their communication costs for unification may become excessive unless the number of communications is limited. Neither the pure nor the distributed search model has provided a way to distribute the clauses of a program. This distribution problem has been investigated by Day [8] in relation to the DDL, and recent results [9] appear to be practical.

2.2 Recent Models of OR-parallelism

A recent AND/OR model of Kale et al. [14, 16, 17, 18] has been extensively developed and contains several interesting features. The Reduce-OR Process Model (ROPM) is currently limited in its implementation to independent AND-parallelism, i.e., AND-parallelism in which only literals without shared variables may be pursued simultaneously. An independent join operation of solutions of independent literals is performed as the solutions arrive. ROPM supports full OR-parallelism including consumer instance parallelism [17]. In this special use execution of each instance of the second of two variable-sharing literals begins with the production of each solution by the first literal. This is particularly appropriate for generate and test procedures [14]. Furthermore, another form of parallelism, pipeline, is available for generate and test procedures. Typically, a generate and test predicate occurs as this: answer(X) :- generate(X), test(X). After the first solution s1 is produced by the generate predicate, the testing of s1 is done while the first predicate simultaneously produces a second solution. Clearly, such pipelining is inappropriate for single solution problems. Potential bottlenecks exist in the OR-parallelism of ROPM since multiple solutions at

OR-nodes must be processed sequentially.

One of the most innovative features of ROPM is the special binding environment which accomodates execution on either a distributed or shared memory machine. Unifications that occur in this binding environment are similar to those of Lindstrom [13]. Unification consists of two phases. If unification is possible, as determined by the first phase, pointers between arguments of the goal and responding clauses are passed to the second phase where references from the child to the parent are eliminated. This may entail some copying. This binding scheme parallels the closed environments algorithm on Conery [7]. Distinctions between the two methods are given in [17].

Structure sharing, rather than structure copying, is also used in ROPM as an efficient means of communication. In distributed memory machines, structure sharing allows all references to variables to be made by using indices if the skeletal code of the program is initially broadcast to all nodes [14]. This is an efficiency that will be useful in extending the DDL to an interpreter that allows structures as predicate arguments.

The ROPM has also been implemented as a machine independent application on top of the Chare Kernel, which supports the dynamic creation of processes and is driven by messages. The Chare Kernel is intended for medium and fine grain parallelism. It separates the decomposition of an application from the allocation of parallel actions to processors. Basically the Chare Kernel is a resource manager which provides a run-time environment by handling scheduling and load balancing and by offering choices for queueing strategies, memory management, and communication primitives. The Chare Kernel also supports quiescence detection, thereby eliminating its inclusion in the ROPM.

The final innovation of the ROPM is the change from the FIFO queue

for messages to a priority queue. In the messages, the priorities are represented as bit-vector fields. This allows a rapid calculation for the first solution of a query. Priorities associated with messages have also been found to be useful for consumer instance parallelism [18].

Tseng and Biswas [19] have proposed a stream based dataflow execution model which combines OR-parallelism with Restricted AND-parallelism. The authors' central contribution lies with the notion of passing binding environments between goal literals. They introduce the non-strict structure known as a stream of streams (or S-streams) as a way of dealing with multiple binding environments. Their premise for OR-parallelism is that all solutions of a program are sought. Descriptor tokens are introduced to eliminate the potential bottleneck in the matching operation of a dataflow execution graph. The most questionable activity of this model is the excessive copying of S-streams between processors since the size of a stream can grow quickly.

Biswas et al. [3] have also looked at the problem of limiting the quantity of solutions produced through unrestricted OR-parallelism. This model is based on a distributed memory machine but makes the restrictive assumption that the program's code is replicated at all processors. For small to medium sized problems this may not be crucial, but it is not suitable for large problems and thereby limits the model's scalability. The restricted OR-parallelism introduced is similar to backup OR-parallelism of [12]. A late binding scheme is also introduced. This delays instantiation of parental variables until the child is completely solved. Copying of partial parental frames to children is needed to effect this idea.

A new approach to parallel logic programming was introduced in [1] as Shared Prolog (SP). In SP a set of logic agents communicate through a blackboard. Although SP claims to be a superset of Prolog, the additional

constructs that must be added alter Prolog severely. In SP implicit global backtracking is forbidden. Syntax additions to Prolog include specifying a database of facts (the blackboard) and a set of agents which simultaneously work on theories. A theory is composed of multiple patterns, each of which consists of a preactivation part (similar to the guard in the Flat languages) and postactivation part. As with the Flat languages, once the preactivation part of a pattern is satisfied, other patterns halt. This is identical to the committed choice determinism of the Flat languages. Other extensions are made to Prolog to effect the required communications of a distributed system. These send/receive extensions are similar to those of the DDL interpreter in that the parallel interpreters of the nodes (agents) are running under Unix and communicating through internet sockets. In SP, however, these are specifically coded with Prolog extensions. There is no mention of AND-parallelism in SP. Although SP requires as many processors as theories in the initial goal, the blackboard is not restricted to being centralized.

An innovative approach to parallelism has been taken by Beer [2]. His premise is to look for parallelism not in the code of the programming language (Prolog) but rather in the machine (Warren Abstract Machine) code into which the language is ultimately translated. The result is pipelined parallelism among machine instructions while running on a shared memory hardware. Although the individual processors are allowed to have local memories, the principal data structures of the interpreter/compiler and the executing program use shared memory. Synchronization is achieved by passing pointers (alias tokens) to the global data structures (e.g., the local stack) from one processor to its right neighbor, in the case of a ring topology for the processors. Clever optimizations are also achieved in the WAM code.

3. OR-parallelism in the DDL

The DDL interpreter was originally implemented without OR-parallelism. Two forms of OR-parallelism are now being implemented. They are Latent OR-parallelism, as initially described by [12], and Competitive OR-parallelism, a variation on Conery's OR process model.

3.1 Latent OR-parallelism

Latent OR-parallelism produces solutions sequentially, but after a solution is found by the servant node and reported to the master, the servant begins to look for the next solution in anticipation of a REDO message from the master. The DDL changes required for this variation are the following. A special queue is maintained by each node. It is known as the OR queue and is activated only when the JOB queue becomes empty. Entries are made to the OR queue when reporting a SUCCESS message to a master node. This double queue structure gives priority to any REQUEST message over any REDO message.

Specific applications may find it more advantageous to retain a single queue and assign priorities to all REQUEST and REDO messages. This alternative generalizes the scheduler (double queue). This allows REQUEST and REDO messages to be interspersed. A priority queue scheduler also allows the depth-first search strategy to be altered for specific applications. For example, priorities can be set for particular levels within a proof tree in order to find the first solution as quickly as possible. Such a system of priorities is

discussed in [16]. The priority queue is but one alternative to the FIFO queue data structure for the scheduler. In a production model, the scheduler may be implemented as a system of interrupts or as a timing mechanism in which the requesting node cancels the ORDER and take "no" for an answer as a default.

To minimize communication with its master node, a servant node uses a local JOB/OR cache to store alternative solutions, with a maximum of one solution per JOB. In processing a REDO message, a node first examines its local JOB/OR cache for an alternative solution before continuing to work on the proof tree for that JOB.

3.2 Competitive OR-parallelism

Competitive OR-parallelism requires adjustment to the local allocation map and to the manipulations of the ORDER and JOB queues. For each predicate used in the head or body of a clause, the local allocation map now holds a set of nodes rather than a single node. If the allocation map shows that predicate p is held locally, it is solved locally and no ORDER is sent until all local solutions have been used. If the local allocation map shows that p is held only by one or more remote nodes or if all local solutions have been exhausted and more solutions potentially exist remotely, then ORDERs are sent to all remote nodes by using a multiple ORDER. However, the requesting node accepts only one answer at a time from its servants. The winning solution is the first SUCCESS reply. A cache is attached to the ORDER queue for storing alternative answers. This cache is checked before ORDERs are placed.

An alternative to using an ORDER cache for multiple responses is to remove any other solutions from the master node's JOB queue and to send a multiple CANCEL order to the other servant nodes after the first answer is obtained. Although the second approach could possible save a remote processor from wasting its time on a problem which has already been solved, there are many detrimental features. First, the multiple CANCEL order creates more messages. Second, the CANCEL causes the work done at the remote node always to be useless. Third, if two solutions have been enqueued at the master node before the first solution is processed, removal of the second solution will cause it to be lost since a REDO message to that node which produced the second solution will cause that node to begin in the middle of its proof tree (just after the place where the solution was found). All these negative aspects greatly outweigh the single positive factor.

4. Planning for Multiagent Systems

The use of both AND- and OR-parallelism in multiagent planning problems is dependent on the structure of the planning system. At one extreme is the central planner, which constructs the detailed plans of the individual agents and coordinates the agents. In this case the agents act as drones, meekly doing as ordered by the central planner. The potential for parallelism in the planning phase of the system is severly limited if the central planner resides on a single processor of the network. Parallelism for a system with a central planner is restricted to the action phase; i.e., agents can pursue their goals in parallel (essentially AND-parallelism), and the central planner can divide its time among monitoring, alternative planning, and scheduling the subordinate agents while the original actions are being done. OR-parallelism can also occur during the action phase of the system if two or more agents are simultaneously pursuing the same goal. If the central planner is only interested in achieving a goal and requests several agents to work on it concurrently, the central planner will likely issue cancel orders to all other agents as soon as the goal is achieved by one agent. This is competitive OR-parallelism, and it is particularly effective in a system which has built-in redundancy. Here redundancy is used to describe the situation where more than one agent is capable of achieveing a particular goal. Both synchronous and asynchronous tasks can be achieved using a central planner, although the synchronous form is more common. The primary advantage of a central planner is that it minimizes communication. Its principal disadvantage is that it inhibits parallelism and consequently may not be adept in real-time environments. Furthermore, the central planner makes the system brittle and allows for catastrophic failure of the system if the central planner is lost.

At the other extreme of planning system lies the fully distributed planner. In such systems many agents participate in the planning phase. The hierarchy is a natural structure for such systems because of the inherent top-down decomposition of goals that is ofter used in planning. Here the full power of both AND- and OR-parallelism can be achieved. For example, as plans are partially constructed at one level and passed down to the next level, planning at the lower level begins. Actions of the agents can be interspersed with the planning, and all nodes of the hierarchy can become engaged in acting and planning simultaneously.

4.1 Replanning in Multiagent Systems

A successful replanning process should be able to accomplish the following: 1) re-achieve destroyed goals, 2) adjust the remaining plan elements according to the changes in the environment and constraints, and 3) provide replanning efficiently.

In order to replan correctly and effectively, the dependencies and interactions of a plan's actions and resources must be represented in a way that permits plan rationale and structure to be retrieved efficiently. Toward this need, an explicit representation must be designed. Typically, a "recovered" plan provides fixtures to the plan elements that are directly affected by unanticipated changes. Since the fixtures may have side effects on other plan elements, a truth maintenance mechanism is needed for propogating the effects through the remaining plan to guarantee consistency.

The complete system integrates planning, execution monitoring, and

replanning subsystems. The planning subsystem formulates original plans. At the current stage, this subsystem is based on the problem-analysis approach of [4]. However, it can be modified according to any other hierarchical or constraint-satisfaction approach. The only required extension to these approaches will be the recording of the dependencies/interactions information during the plan formulation. The execution monitoring subsystem monitors the progress of plan execution. Any unanticipated change in the environment can enter the system by interrupting the program at any time during plan execution. When an unanticipated change is entered, the replanning subsystem will take over the system and perform the necessary procedures.

For the purpose of replanning, contextual information about how a plan was formulated has to be stored along with the plan. In addition to this plan rationale, the plan expansion structure must also be hierarchically represented in a data structure so that change-affected parts can be located efficiently. The Wedge Table and the Action Table are devised for these purposes.

The replanning process searches the plan tree for the contaminated parts. In order to achieve efficient retrieval, a plan tree is represented in a data structure call the Wedge Table. A node that does not have any descendants is called a "primitive" condition. Such a node is either an iniitial condition or a condition induced from an earlier action. A wedge consists of a subgoal and its expanded subtree. If a node is not a primitive condition, it is expanded by the attached preconditions of the action that would achieve the condition. Each of the expanded nodes in turn can be expanded if it is not a primitive condition. The expansion of a wedge will terminate with primitive conditions. Each entry of the Wedge Table denotes a node of the tree and may contain one or two parts. The first part is a list of node numbers, and the second part is an action number that refers to the Action Table. If an entry

contains only one node number, it indicates an initial condition. If an entry has two nodes numbers, but without an action number, the first node is satisfied subordinately when the second node is achieved. All other entries represent non-primitive conditions. The first node number of these entries is the top node of the wedge, and the rest of the list contains the immediate descendants of that node. Each of these descendants in turn may be a top node of another wedge. Each top node of a wedge has an action number. When all of the action's immediate descendants become true, this action can be executed to achieve the top node. The basic AND-parallelism of the DDL can be used to achieve simultaneously the immediate descendants of a node provided that the descendants are held distributedly and that they require no synchronization.

The triagle table introduced in STRIPS [11] can be extended with more contextual information to construct the Action Table, which is a rectangle. Tne table is built for each agent. Each column of the table represents an action, and the column entries indicate the conditions on which this action is based. Actions are numbered according to the execution sequence.

The first column has no action number. It represents the initial state. Each row denotes a predicate of an object which is involved in the plan formulation. Each entry of the first column contains only one entity which represents the initial value of the corresponding predicate. Each table entry indicates the dependency of an action with a resource condition and the interaction between an action and other actions. A complete table provides an explicit dependencies/interactions scenario of an agent's plan. There are three possible elements in each table entry except those in the first column. The first element is a condition type. The condition type indicates how the associated predicate affects the action. The second element is a node number

which indicates the corresponding node in the plan tree. The third element indicates the newly derived postcondition of the action.

For a multiagent Action Table, the second element may be a list of n pairs of numbers which indicates interaction among n agents. When agents interact with each other, one of two possible columns (Wait or Signal) is inserted in each agent's actions. In these interaction columns, each pair of the list contains the interacting agent's name and its action number.

4.2 Resource Classification

A replanning process must have a resources reasoning scheme since most actions are related to domain resources. In order to achieve efficient replanning, resources are grouped into three classes according to their peculiarities. The resource classes are coalescent, preëmptive, and sharable. Any consumable or smelting resource like fuel or solder is called a coalescent resource. A resource (such as a tool) which can be used by only one agent at a time and reused by any agent later is a preëmptive resource. Resources that can be used by concurrent actions of multiple agents are called sharable resources. The resource classification imposes important constraints on recovering from unanticipated changes. For example, when a specific preëmptive resource is demanded by more than one action simultaneously, the recovery procedure may try to instantiate a different resource or to synchronize these actions.

Coalescent resources present the most likely source of parallelism since many copies (versions) can be used simultaneously. Single instances of preëmptive and sharable resources offer no parallelism; however, multiple

preëmptive resources that belong to a similarity class may provide OR-parallelism to the multiagent system.

4.3 Condition Classification

When unanticipated changes are detected, plan recovery procedures should be based on the roles of the changed conditions. This also leads to the need for classification. Five types of conditions have been identifid for this purpose. They are precondition, subcondition, postcondition, umbrella-condition, and decision-condition.

A condition which is required for an action to be executed in any circumstance is a precondition of that action. Preconditions are represented as the immediate descendants of a node in a plan tree. If a precondition of an action is not true at execution time for unanticipated reasons, it must be achieved. However, satisfying a precondition may destroy other preconditions of the action. These conflicting recovery activities are eliminated by using precondition-precedence rules to solve this problem. An example of such a rule is this: if a condition must be maintained until other conditions are satisfied, it has higher precedence over the other conditions.

If a condition is not a precondition of an action but contextually affects execution of the action, it is called a subcondition. An action can be activated even if its subconditions are not satisfied. However, the subcondition violation of an action may make the action unnecessary or destroy the effects of the action. Therefore, subconditions of an action should be secured before executing the action. Preconditions and subconditions together are the primary source of parallelism in multiagent systems.

Postconditions of an action are the state instances induced by applying the action. When a postcondition is not satisfied, a backward scanning should be done in the Action Table to look for a matched state. If there is a column that matches the current state, plan execution can be resumed from that point. When such a column cannot be found in the Action Table, the violated post-condition must be re-achieved.

An umbrella-condition may not be revealed as a node in a plan tree. However, an umbrella-condition causes the wedge expansion of a node, and all actions in the wedge expanded from that node depend on it. When an umbrella-condition is violated during plan execution, the whole expanded wedge may become invalid. An action may change its umbrella-condition. However, if an umbrella-condition is changed as a postcondition of an action, it is not an umbrella-condition violation.

When there is more than one choice for expanding a node, a condition that is true or assumed to be true at the planning time and is a factor in decision making is called a decision-condition. Usually, a decision is made based on some domain heuristics such as preference of agents and/or plan optimality. Decisions can use OR-parallelism to weigh the individual choices before commitment to a solution is made.

4.4 Recovery Procedures

Proper recovery depends on the role and the type of effects that a change has on the plan. Standard recovery procedures can be designed for each category of role-and-effect to achieve efficiency. Nine recovery procedures have been found to be need. They are Expand-wedge, Insert-

precondition, Re-adjust, Redo, Re-expand-wedge, Remove-wedge, Re-instantiate, Replan, and Undo. Combinations of these recovery procedures are used for both resources and conditions. In complex replanning problems, combinations of these recovery procedures can involve all forms of parallelism.

For example, in the case of a precondition violation, there are four cases that must be considered. The case considered here occurs when a violated precondition has higher precedence over any other preconditions with respect to the rule shown in section 4.3. The lower precedence preconditions must be confirmed after the violated precondition is re-achieved. "Re-expand-wedge" will be applied to the violated preconditon to re-achieve it. "Re-adjust" will confirm the truth of the lower precedence preconditions. The special procedure "re-expand-wedge" makes a node a subgoal and expands a wedge for it. The procedure "re-adjust" adjusts nodes of wedges affected by the replanned actions. In some situations, expanding the wedge for a node may satisfy or destroy some conditions of other wedges. In such cases, the nodes of the affected wedges must be re-adjusted.

4.5 Truth Maintenance

The truth maintenace procedures can be divided into three categories: no conflict, resource conflict, and condition conflict. These procedures are collectively called the action dependency calculation. In the case that the remaining actions of an agent do not interact with any actions of other agents, there is no conflict and no plan adjustment is necessary.

When actions of different agents compete for a resource (Resource Conflict), the conflict should be solved according to the characteristics of the

involved resource. Two primitive actions are defined to synchronize action, "Wait" and "Signal". Each of these two actions has two arguments, an agent name and an action number. When an agent is interacting with another agent, both arguments will be recorded in the Action Table. Notations and assumptions used in the resource conflict resolution procedure are these: G=Agent encountering change, Gi=Agent interacting with G, A= Interacting action of G, Ai= Interacting action of Gi. A simplified resource conflict resolution algorithm is as follows:

> If a remaining action of G, A, demands a resource at the same time as Ai does, then
>
> > If the resource (r) is sharable, then the truth maintenance is not not necessary since the resource can be shared by the agents;
> >
> > If the resource (r) is preëmptive, then insert Wait(G,A) before Ai so that Gi has to wait until G signals that it has completed A and insert Signal(Gi,Ai) after action A so that G will send a signal to Gi after it completes A;
> >
> > If the resource (r) is coalescent, then instantiate another resource that can substitute for resource r.

If the actions of a recovered plan negate any conditions of other agents' actions which are currently believed true, these conditions should be reachieved and synchronized. The procedures for such recoveries are similar to that shown for the resource conflict.

5. Conclusions

This report has presented a discussion of the pertinent issues and a variety of models of OR-parallelism in logic programming. Two of the models, latent OR-parallelism and competitive OR-parallelism, were shown to be appropriate for the Distributed DataLog interpreter. The design of the required modifications for the DDL were also described.

This report has also presented an overview of potential parallelism in planning and replanning models for multiagent systems. Particular emphasis has been placed on identificiation of parallelism in plan representation, resource classification, condition classification, error identification and location, error recovery, and truth maintenance.

Future work in this area will be directed to the development and execution of experiments with OR-parallelism in the DDL as well as planning models in multiagent domains. These experiments will use the SUN workstations network and, if available, a distributed memory multiprocessor.

6. References

1. V. Ambriola, P. Ciancarini, M. Danelutto, "Design and Distributed Implementation of the Parallel Logic Language Shared Prolog," Proc. Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, April 1990, Seattle, WA, pp. 40-49.
2. J. Beer, Concepts, Design, and Performance Analysis for a Parallel Prolog Machine, Springer-Verlag, New York, NY, 1987.
3. P. Biswas, S.-C. Su and D. Y. Y. Yun, "A Scalable Abstract Machine Model to Support Limited-OR (LOR)/Restricted-AND Parallelism (RAP) in Logic Programming," Proc. Fifth Int. Conf./Symp. on Logic Programming, August 1988, Seattle, WA, pp. 1160-1179.
4. K. H. Chang and W. Wee, "A Planning Model with Problem Analysis and Operator Hierarchy," IEEE Trans. on PAMI, Vol. 10, No. 5, 1988, pp. 672-675.
5. A. Ciepielewski and S. Haridi, "Control of Activities in the OR-parallel Token Machine," Proc. 1984 Int. Symp. on Logic Programming, Atlantic City, NJ, February 1984, pp. 49-57.
6. J. S. Conery, Parallel Execution of Logic Programs, Kluwer Academic Publishers, Boston, MA, 1987.
7. J. S. Conery and D. F. Kibler, "Parallel Interpretation of Logic Programs," Proc. Conf. on Functional Programming Languages and Computer Architecture, Wentworth-by-the Sea, NH, October 1981, pp. 163-176.
8. W. B. Day and W. Chung, "Allocation Schemes for Distributed Problem-solving in Logic Programming," Proc. Ninth IEEE Int. Phoenix Conf. on Computers and Communications, Phoenix, AZ, March 1990, pp.

727-733.

9. W. B. Day and B. K. Walls, "An Allocation Algorithm for Distributed DataLog Pograms," submitted for publication.

10. W. B. Day, "Parallel Logic Programming Architectures," Final Report for USAF Contract #F30602-88-D-0027, Submitted to Rome Air Development Center, Griffiss Air Force Base, NY, June 1989.

11. R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," Artificial Intelligence, Vol. 2, 1971, pp. 189-208.

12. K. Furukawa, K. Nitta, and Y. Matsumoto, "Prolog Interpreter Based on Concurrent Programming," Proc. First Int. Logic Programming Conf., Marseille, France, September 1982, pp. 38-44.

13. G. Lindstrom, "OR Parallelism on Applicative Architectures," Proc. Second Int. Logic Programming Conf., Uppsala, Sweden, July 1984, pp. 159-170.

14. L. V. Kale, B. Ramkumar, and W. Shu, "A Memory Organisation Independent Binding Environment for AND and OR Parallel Execution of Logic Programs," Proc. Fifth Conf./Symp. on Logic Programming, August 1988, Seattle, WA, pp. 1223-1240.

15. S. Kasif, M. Kohli, and J. Minker, "PRISM: A Parallel Inference System for Problem Sovling," Proc. Eighth Int. Joint Conference on Artificial Intelligence, Karlsruhe, Germany, August 1983, pp. 544-546.

16. B. Ramkumar and L. V. Kale, "Compiled Execution of the Reduce-OR Process Model on Multiprocessors," Proc. N. American Conf. on Logic Programming, October 1989, Boston, MA, pp. 313-331.

17. B. Ramkumar and L. V. Kale, "A Chare Kernel Implementation of a Par-

allel Prolog Compiler," Proc. Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, March 1990, Seattle, WA, pp. 99-108.

18. V. A. Saletore and L. V. Kale, "Obtaining First Solutions Faster in AND-OR Parallel Execution of Logic Programs," Proc. N. American Conf. on Logic Programming, October 1989, Boston, MA, pp. 390-406.
19. C.-C. Tseng and P. Biswas, "A Data-driven Parallel Execution Model for Logic Programs," Proc. Fifth Int. Conf./Symp. on Logic Programming, August 1988, Seattle, WA, pp. 1204-1222.
20. D. S. Warren, M. Ahamad, S. K. Debray, and L. V. Kale, "Executing Distributed Prolog Programs on a Broadcast Network," Proc. 1984 Int. Symp. on Logic Programming, Atlantic City, NJ, February 1984, pp. 12-21.