STRATEGIES FOR DISTRIBUTED CONSTRAINT SATISFACTION PROBLEMS

Q. Y. Luo P. G. Hendry J. T. Buchanan
Department of Computer Science, University of Strathclyde
Glasgow G1 1XH, UK
qyl@cs.strath.ac.uk pgh@cs.strath.ac.uk iain@cs.strath.ac.uk

May 27, 1994

ABSTRACT

Constraint satisfaction problems are important in AI. Various distributed and parallel computing strategies have been proposed to solve these problems. In this paper, these strategies are classified as distributed-agent-based, parallel-agent-based, and function-agent-based distributed problem-solving strategies. These different strategies are presented and discussed. Parallel-agent-based strategies are found to be very versatile. Computational experience is presented.

1 INTRODUCTION

A large number of problems in AI and other areas of computer science can be viewed as special cases of the constraint satisfaction problem (CSP). Some examples [13] are machine vision, belief maintenance, scheduling, temporal reasoning, graph problems, floor plan design, the planning of genetic experiments, and satisfiability.

CSPs have three basic components: *variables, values* and *constraints.* The goal is to find an assignment of values to variables, from their separate domains, such that all the constraints are satisfied.

Many sequential algorithms for solving CSPs have been developed. However, in the real-world, we may have to deal with distributed CSPs. There are two main reasons to address distributed CSPs. Firstly CSPs themselves may be logically or geographically distributed. These problems may best be solved by a multi-processor platform. Secondly parallel or distributed computers may provide more computing power if used effectively and this is important in considering the amount of computation required to solve CSPs.

A distributed constraint satisfaction problem (DCSP) is defined as a CSP in which multiple agents (software processes) are involved. DCSPs are important sub-problems in distributed artificial intelligence (DAI). As described by [29], various DAI problems can be formalised as DCSPs, and in turn DCSPs can provide a formal framework for studying various DAI methods.

2 DISTRIBUTED ALGORITHMS: AN OVERVIEW

In some special classes of CSPs, such as the problem of labeling a 2-D line drawing of a 3-D object, there exists a fast parallel algorithm [12] that executes in time $O(\log^3 n)$ with $O((m + n^3)/\log n)$ processors on an exclusive-read exclusive-write (EREW) parallel random access machine (PRAM), where n is the number of variables and m is the number of constraint relations. However, in this paper, the DCSPs addressed will mainly be general problems to be solved on more practical MIMD machines and so algorithms for special classes of DCSPs will not be covered.

DAI is viewed by [1], [11] and [10] as providing a strategy for problem-solving. Where the problem is naturally distributed, it can be divided among a number of agents, and a control regime has

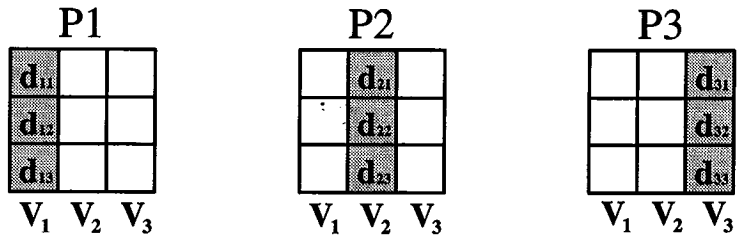

Figure 1: **A DAB example for the 3-queens problem**

responsibility for co-ordinating the problem-solving activities of the respective agents. Different control regimes, such as centralised, discentralised, and other co-ordinating mechanisms may constitute different DAI methods.

On the basis of [1] and [10], parallel artificial intelligence (PAI) involves the application of parallel computing resources to a problem without a restructuring of that problem. The purpose of PAI is to achieve a (linear) speed-up in response to applying multiple processors to a given problem.

However, for any particular problem, which method (different DAI methods or PAI) is the best? There is little literature to answer this question. Most research papers deal solely with one of them, without a comparison between methods.

In this paper, DAI is defined as being concerned with problem-solving situations in which several agents can be used (together or separately) to achieve a common set or separate sets of objectives. This is intended to encompass all of the classes above. As a consequence, different distributed problem-solving methods can be compared without losing generality.

The structure of a distributed constraint satisfaction algorithm is dependent on which DPS strategy it relies. Generally, the basic elements of DPS strategies related to the nature of distributed computing and the hardware are important, such as control structures (centralised or decentralised), search space types (shared or separated) and communication (message-passing or shared memory). By considering these, the most common DPS strategies to solve DCSPs may be classified as *distributed-agent-based* (DAB), *parallel-agent-based* (PAB) and *function-agent-based* (FAB). Different DPS strategies may involve different control structures, problem spaces and communication methods.

One significant difference among these three methods is whether conflicts may exist among the problem-solving agents. In fact, only in the DAB strategy does there exist the possibility of conflicts among agents, due to the shared search space. To guarantee the soundness and completeness of algorithms, there is a need for an inter-agent backtracking or negotiation mechanism. Therefore, co-operation among agents is necessary in this strategy. As is to be expected, this inter-agent backtracking or negotiation mechanism may cause a communication bottleneck, or complexity problems and delays due to synchronisation of the actions of agents.

2.1 Distributed-agent-based Strategy and Algorithms

In the DAB strategy (Figure 1 [1]) , the problem is distributed, based on the variables. Each agent is in charge of one or more variables and their domains. A variable is controlled by only one agent attempting to assign it a legal value from its domain.

The search space is shared among agents and the action of one agent may affect other agents directly or indirectly because of the constraints acting between variables.

Difficulties arise because of the asynchronous behaviour of agents. Other problems include distributed memory (no shared global data), infinite processing loops and communication overheads (which can be very high in many cases).

[1] This is a problem of placing 3 queens on $3 * 3$ chess board (i.e. three variables and each variable V_i has a domain - a set of d_{ij}) so that no two queens attack each other. In these figures, each processor (P_k) can place queen(s) on its part of the chess board marked with shadow. In Figure 3, FC stands for the forward check function

For DAB algorithms, control mechanisms to resolve conflict can be centralised (for example by way of a total linear order) or decentralised (by way of negotiation).

The total order is an attractive method to control both search and conflict resolution because of the ease of implementation and similarities to sequential algorithms. However, it also introduces several problems of its own, such as communication overheads and load imbalance due to the fixed search order where the higher order agents may become bottlenecks.

A DAB system that does not have a centralised control strategy requires some other control method that does not require global knowledge. Control mechanisms within such a negotiation strategy are complex, due to lack of global knowledge, and they may introduce a large overhead in problem-solving.

In this section, we examine mechanisms to support centralised and decentralised structures.

A Root Agent Algorithm: *A root agent algorithm* is reported in [25]. This algorithm collects all information on variables, their domains and constraints in a special agent - *the root agent.* Since the root agent has all the information on the problem space, it can find a solution. The centralised mechanism is able to use any sequential search algorithm to solve DCSPs. On the other hand, this algorithm does not fully exploit parallelism and communication with the root agent can be bottleneck.

A Distributed Hill Climbing Method: *A distributed hill climbing method* is introduced in [21]. This method irrecoverably changes variable values to remove constraint violations, and does not perform backtracking. In this algorithm, each agent asynchronously instantiates its variable so that its own constraints are satisfied. Here the algorithm is not guaranteed to find a solution, since no backtracking is performed.

Depth First Search with A Self-stabilising Protocol: In [8] the network consistency problem is tackled by using depth-first search with backjumping, but [4] argued that [8] is not self stabilised since the system may not settle to a legal set of configurations from *any* initial configuration. Both depth-first search with backjumping and a self–stabilising protocol for a connectionist architecture are used in [4].

Multistage Negotiation: Instead of a self-stabilising protocol, [5] suggested the use of *a multistage negotiation* strategy to solve DCSPs. A co-operation paradigm and co-ordination protocol for a distributed planning system consisting of a network of semi-autonomous agents with limited internode communication and no centralised control is presented. A multistage negotiation paradigm for solving DCSPs in this kind of system has been developed. Nonetheless, [5] did not clearly point out how to prevent infinite processing loops and possibly potential message complexity.

An Asynchronous Backtracking Method: An asynchronous backtracking method to address DCSPs was introduced in [29]. It performs the distributed search concurrently. It manages asynchronous change by a method called *context attachment* where an agent must include its current beliefs about other agents when sending a conflict-related message. To avoid infinite processing loops [29] used a *total order* among agents. The use of a total (or linear) order unifies the search space even though the algorithm is on a distributed platform. By using the idea of a *total order*, [29] adopts a centralised DPS control structure, although the *total order* may appear "unfair" to agents, compared with the *almost uniform protocol* in [4]. It is quite straightforward and easily realised on MIMD machines. In its absence, it is necessary to establish and implement protocols among agents to avoid infinite processing loops and to resolve any conflicts between agents, and this may bring a large increase in message complexity.

However, as the searching order is fixed and the agents with certain orders, for example the higher orders, may have to perform more search steps and process more messages than other agents, the total order causes problems with load balancing and with message passing volume, and [29] [26] failed to address either of these problems.

In [28] two heuristics for increasing the search efficiency of the asynchronous backtracking algorithm are introduced. By using heuristics, the total ordering can be changed partially. The heuristics are *min-conflict* heuristic (when selecting a value for a variable, select a value that minimises the

number of conflicts) and *change-oldest* heuristic (when selecting a variable to change from conflicting variables, select a variable that has not changed for the longest period). An upper bound may be used to control the range of the order that can be changed.

An upper bound on priority values is necessary to guarantee that this algorithm does not involve an infinite processing loop. If the priority values of all variables reach this upper bound, the algorithm becomes identical to the basic asynchronous backtracking algorithm (with a total order) and performs an exhaustive search until a solution is found.

Discrete event simulation in [28] shows, by comparison of the search steps, that the asynchronous backtracking algorithm can solve large-scale problems which otherwise cannot be solved within a reasonable amount of time.

However, [28] did not point out the potential message complexity this algorithm may cause. When it processes each backtrack, it may change the ordering of variables and then must propagate this new ordering around the network. Because of the changing order, this asynchronous backtracking algorithm takes on the features of a sequential backtracking algorithm, since all the distributed tasks based on the old variable ordering are performed to be later undone.

Distributed Extended Forward Checking while Conflict-directed Backjumping: A set of distributed search algorithms is presented in [15] [16]. These include distributed forward checking while backtracking (DFC-BT), distributed forward checking while conflict-directed backjumping (DFC-CBJ) and distributed extended forward checking while conflict-directed backjumping (DEFC-CBJ). These use the *context attachment* and *total order* ideas of [29]. The algorithms address methods for reducing the amount of communication required and ways in which redundant computation can be avoided. A simple load balancing strategy is also proposed but is not sufficient to solve the imbalance caused by using the total order.

To avoid unnecessary search paths and consistency checks DEFC-CBJ uses distributed extended forward checking which records nogood values (causing conflict) and uses them during search. It also employs a conflict-directed backjump method [22], so when resolving conflicts it attempts to find the root cause of any conflict.

An important part of the algorithm of [15] [16] is its ability to cope with dynamic changes within a DCSP. Many real-world problems are dynamic and it is desirable not to lose valuable information or, even worse, to have to start search again when the problem changes slightly. In [27] a similar method called the asynchronous incremental relaxation algorithm is reported. It uses constraint relaxation in DCSPs by way of nogoods.

A Hybrid Search Algorithm: Both [29] and [15] made the simplification of having one agent in charge of only one variable. This leads to an increase amount of communication as there are no local CSPs to be solved and the ratio of computation to communication time decreases. Furthermore, due to the total order, it is very difficult to use the dynamic variable ordering heuristic to assist distributed search. However, both [15] and [21] noticed that a (static) variable ordering does have a marked impact on the performance of distributed asynchronous backtracking. Dynamic variable ordering may have a large impact on the performance of distributed search.

An alternative scenario is developed in [17] and [28] where each agent is in charge of several variables. A mapping mechanism is demonstrated in [17], with several variables per agent meaning that both distributed search algorithms (between agents) and sequential search algorithms (within each agent) may be used. In turn this allows heuristic methods (attempting to guide search towards a solution) to be used locally, where these methods may have already been developed for sequential algorithms.

To even the load balance problem between processors and speed up the search, this hybrid (distributed and sequential) search algorithm also supports the backtrack-guided *overstep search*. When lower order agents become idle, they perform overstep search. This means that they continue to generate partial local solutions based on the current beliefs about other agents. These partial

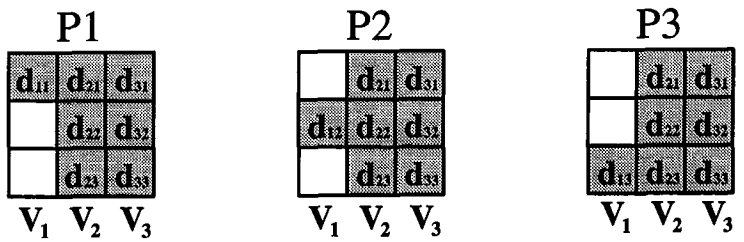

Figure 2: **A PAB example for the 3-queens problem**

solutions may then be referenced when a higher agent backtracks. If there is a valid partial solution, the agent sends it out, if not, the agent again searches based on backtracking information.

Distributed Constrained Heuristic Search: In [24] a *distributed constrained heuristic search* method is presented to address DCSPs. It provides both structure and focus in individual agent search spaces to 'optimise' decisions in the global space. The method achieves this by integrating distributed constraint satisfaction and heuristic search. The notion of textures that allow agents to operate in an asynchronous concurrent manner is introduced. The employment of textures coupled with distributed asynchronous backtracking, a type of distributed dependency-directed backtracking, enables agents to instantiate variables in such a way as to substantially reduce backtracking. However, this asynchronous backtracking method does not guarantee the completeness of the algorithm.

2.2 Parallel-agent-based Strategy and Algorithms

In the PAB strategy (Figure 2), the problem is distributed, based on the domains of the variables. Each agent may have a complete search space or a part of the complete search space. Each complete or partial search space involves all variables and is independent of other search spaces. Each processor is therefore solving a unique CSP problem and no communication is necessary.

A PAB strategy has several advantages. If the problem is not naturally distributed, it can still be addressed. Solving the same problem using DAB may cause serious communication bottlenecks and load balance problems. PAB can directly use any sequential CSP algorithms, needs little communication and can use established global heuristic search strategies.

However, PAB may have difficulty if each agent has only a partial disjoint search space and the problem is over-constrained. In this case, it is difficult to establish the cause of conflict due to a lack of logical links among different search spaces. After all agents fail to find any solution, the algorithm may need to check all conflicts again to find the true cause of failure. This approach may not be suitable for problems that are geographically distributed.

Some strategies used in PAB are now developed.

Parallel Searching Separated Spaces: In [2] the search space is divided among a set of processors each executing a sequential CSP algorithm. In this case, forward checking (FC) and the dynamic variable ordering heuristic is used although any other sequential algorithm could replace it. Dynamic load balancing is also addressed through dividing the search space. Redundant work may however still be done because one processor does not make use of the search information obtained by another during search.

It seems that [2] and [29] are two extremes. In [2] communication is not used to aid search whereas [29] relies heavily on its communication of partial results, at a heavy cost to performance. In [17], it tries to find a middle ground where the parallelism of [2] is used along with a limited sharing of partial results to prune the search space.

The presentation of [23] is similar to [2]. It compares simple (standard) backtracking with a heuristic (dynamic variable ordering) backtracking parallel search method.

Parallel Search Separated Spaces while Sharing Information: It is widely believed that a group of co-operating agents engaged in problem-solving can solve a task faster than either a single agent or the same group of agents working in isolation from each other. Thus in [3] a blackboard is

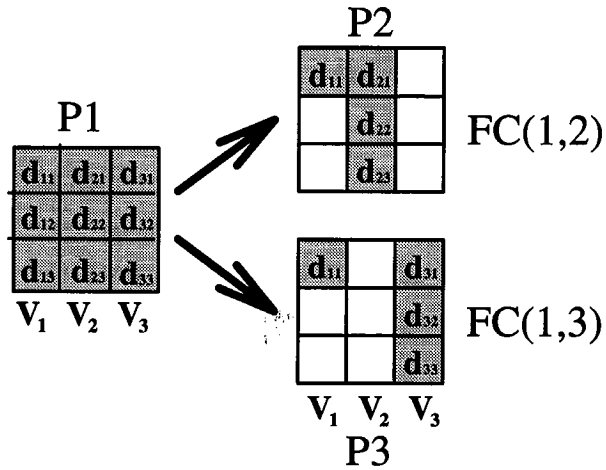

Figure 3: **A FAB example for the 3-queens problem**

applied to share useful *hints* among agents. Hints are potentially useful information such as partial results, and can be used to guide the selection of search paths.

In [17] the nogood mechanism (reported in [6] and [15]) is used to share information between agents who have different search spaces. A *nogood* contains a (*variable*, *value*) pair that is disallowed due to conflict. The reason for the conflict is also stored so that, if this reason no longer holds (i.e. the search state that caused the conflict changes) the nogood value can be released. If there is no change in the conflicting variables then the nogood is not released. This allows the search space to be pruned. When there is no conflict variable left in a nogood, its value will be not released. This special kind of nogood is called *a dead-end.* Agents share these dead-ends and make decisions based on the current dead-end information. When an agent receives a dead-end from another agent, it will check that dead-end based on its belief. If it is also a dead-end to the agent, it will prune the disallowed value from its domain to reduce search.

Dynamic Load Balancing by way of Dead-Ends: In [19] a method is proposed for sharing partial information. When performing dynamic load balancing, instead of sending out half of the remaining search space, this method sends out *newly built dead-ends* and *other dead-ends.* The *newly built dead-ends* are used to split the remaining search space, and *other dead-ends* are useful partial search information. Any values in these dead-ends will not appear in the solution. Since the two agents search different parts of the same search space, there will be no problem in them sharing and using these useful dead-ends.

Compared with the method of sending out half the available search space [2], this method has some advantages. It needs less communication bandwidth, as it only sends out a few dead ends, not the complete search space. There is no change to the original search space. The disjoint search spaces are created by building dead-ends, and so the search space that is given away is already pruned since the nogoods are still valid. As a consequence, there will be no need to do redundant search (by different agents), unlike the case of directly sending out half the remaining search space, where pruning information is lost.

2.3 Function-agent-based Strategy and Algorithms

A FAB strategy attempts to use parallelism by using spare processors to perform repeated tasks. For example, if search is taking place on one processor using the forward checking sequential algorithm then spare processors can be used to perform the actual forward checking (the domain filtering) in parallel (Figure 3). This will, of course, incur some overheads starting new processes but these will be small if the problem space is large. This method is only suitable for shared-memory machines where the information in a parent process can be seen and manipulated by its children.

A FAB algorithm is presented in [18]. The domain filtering is performed in parallel. If a domain becomes empty then the variable will change its value or backtrack without waiting for other forward checks to finish.

The main problem with this approach is the overhead involved in starting processes to perform the domain filtering. If the problem size is large, requiring much filtering, then this overhead is relatively small.

3 COMPARISON OF REPRESENTATIVE ALGORITHMS

This section presents some computational results obtained by implementing these DCSP algorithms on a SEQUENT SYMMETRY using the C programming language, and a MEIKO Computing Surface (a transputer array) using CSTools.

Two sets of results are presented. The first is for the Zebra problem. This is described in [7] and [22] and is composed of 25 variables that correspond to five groups (each containing five variables) in which the variables have constraint relations between each other and additionally there are a number of constraint relations between the variables in different groups. The order of the variables is important to the efficiency of search. For this reason, some of the results that follow are the average values taken over all ordering permutations of the groups (5! = 120 permutations in total). The order within each group was fixed. In this paper, we use the definition of the Zebra problem in [22] that has 11 solutions. The density ρ[2] of the Zebra problem is 0.20.

The second problem is the well-known n-queens problem. This is the problem of placing n queens on an $n * n$ chess board so that no two queens attack each other. Although this is quite an 'artificial' problem, it represents a 'worst case' scenario where every variable has a constraint with all other variables. The density ρ of the n-queens problem is 1.0.

The computational results are obtained mainly from the various *representative* distributed or sequential algorithms [15] [14] [18], based on the algorithm forward-checking while conflict-directed backjump (FC-CBJ) supported by nogoods (NG) and the postponed revision (PR) mechanism (except that DEFC-CBJ [15] does not support postponed revision). The main objective here is to examine these different (sound and complete) algorithms with the same effective core search methods on the same problems. This is by no means to say that the observations in this paper will stand for all other cases. However, they may give us a partial integrated view of the merits and weaknesses of each of these algorithms and, therefore, help us to find the best method for certain problems.

In the tables, the algorithm extensions used are: HY for a hybrid using both a distributed search algorithm and a sequential search algorithm in DAB algorithms, SN for sharing nogoods among different processors in a PAB search strategy, DO for dynamic variable ordering (in DAB algorithms, dynamic variable ordering is only performed locally) and OS for overstep search. All PAB algorithms are supported by a dynamic load balancing mechanism. In Table 1 all the values (except time) are *averages* over 120 runs of the algorithms, and the time includes all overheads such as the initial propagation of constraints. In these tables, NC stands for the number of consistency checks, NB for the number of backtracks and NM for the number of messages. The ratio in the tables is the rate of useful partial solutions to all generated partial solutions performed by overstep search algorithms. In Table 4, LB stands for the average CPU usage [3] (i.e., load balance) observed by the UNIX function call time. In the Tables 4 and 5, the algorithms find all solutions for the single ordering Zebra and n-queens problems. S_{simple} and E_{simple} refer to the speedup and efficiency in Table 4 for the same problems and the same algorithms with only simple relation check operations. In the tables the abbreviation ALG is used to denote the base algorithm FC-CBJ-NG-PR and the times were obtained using the *time* system call under UNIX, in units of seconds. Most performance terms such as speedup and efficiency used are on the basis of [9].

[2] Density $\rho = \frac{c}{n(n-1)}$, where c is the number of constraints within the problem and n is the number of variables.

[3] $LB = \frac{\textit{the CPU usage in total}}{\textit{the number of CPUs}}$

3.1 Speedup and Efficiency

Table 1: **Comparison of Algorithms for the Zebra against 120 orders**

Algorithms	NC	NB	NM	Ratio	Time[s]	S	E
To find the first solution							
Sequential ALGs (one processor)							
ALG	1636	128	n/a	n/a	108	n/a	n/a
ALG-DO	268	0	n/a	n/a	17	1	1
Distributed-agent-based ALGs (three processors)							
DEFC-CBJ	3356	555	2859	n/a	379	0.045	0.015
HY-ALG	2036	155	363	n/a	154	0.11	0.037
HY-ALG-DO	1022	49	154	n/a	74	0.23	0.077
HY-ALG-DO-OS	1457	130	169	0.24	79	0.22	0.73
Parallel-agent-based ALGs (three processors)							
ALG	2155	130	n/a	n/a	66	0.26	0.086
ALG-SN	2012	123	n/a	n/a	58	0.29	0.098
ALG-DO	816	17	n/a	n/a	30	0.57	0.19
Function-agent-based ALGs (three processors)							
ALG-DO	270	0	n/a	n/a	20	0.85	0.28
To find the all solutions							
Sequential ALGs (one processor)							
ALG	7780	830	n/a	n/a	579	n/a	n/a
ALG-DO	2322	270	n/a	n/a	207	1	1
Distributed-agent-based ALGs (three processors)							
HY-ALG	10994	1113	2090	n/a	692	0.30	0.10
HY-ALG-DO	8942	801	1542	n/a	549	0.38	0.13
HY-ALG-DO-OS	11626	1414	1520	0.46	708	0.29	0.098
Parallel-agent-based ALGs (three processors)							
ALG	9305	905	n/a	n/a	298	0.69	0.23
ALG-SN	8570	874	n/a	n/a	251	0.82	0.27
ALG-DO	4059	425	n/a	n/a	160	1.29	0.43
Function-agent-based ALGs (three processors)							
ALG-DO	2363	270	n/a	n/a	221	0.94	0.31

Table 2: **Comparison of Algorithms for the n-queens**

Algorithms	NQ	NS	NC	NB	NM	Ratio	Time[s]	S	E
Sequential ALGs (one processor)									
ALG-DO	8	all	12702	1008	n/a	n/a	4.9	1	1
ALG-DO	15	one	1084	14	n/a	n/a	0.59	1	1
Distributed-agent-based ALGs (three processors)									
HY-ALG-DO	8	all	18341	1540	2542	n/a	11.10	0.42	0.14
HY-ALG-DO	15	one	19937	886	1228	n/a	12.60	0.39	0.13
HY-ALG-DO-OS	8	all	17418	1627	2355	0.88	9.07	0.54	0.18
HY-ALG-DO-OS	15	one	19057	1021	1375	0.64	10.72	0.46	0.15
Parallel-agent-based ALGs (three processors)									
ALG-DO	8	all	13184	1027	n/a	n/a	2.8	1.75	0.58
ALG-DO	15	one	3317	41	n/a	n/a	1.2	0.49	1.6
Function-agent-based ALGs (three processors)									
ALG-DO	8	all	12702	1008	n/a	n/a	6.9	0.71	0.24
ALG-DO	15	one	1087	14	n/a	n/a	0.93	0.63	0.21

The sequential runtime T_s is defined as the time taken to solve a particular problem instance on one processing element. The parallel runtime T_p is the time taken to solve a particular problem instance on an ensemble of P processing elements. The speedup S achieved by a parallel system is

Table 3: **FAB Algorithm and Sequential Algorithm for the 25 queens problem to find the *first* solution**

Algorithms	NP	NC	NB	Time[s]	S	E
Sequential ALG-DO	1	9246	184	3.96	1	1
Function-agent-based ALG-DO	3	9408	185	2.89	1.37	0.46

defined as gain in computation speed achieved by using P processing elements regarding a single processing element.

$$S = T_s/T_p$$

The efficiency E denotes the effective utilisation of computing resources. It is the ratio of the speedup to the number of processing elements used.

$$E = S/P$$

From Tables 1, 2 and 3, we may make the following observations on speedup and efficiency, related to each method:

a) To find the first solution to the problems, DAB algorithms perform no better then their sequential equivalents. DAB algorithms simply perform more checks, backtracks and delayed backtracks (i.e. inter-agent backtracking that is heavily dependent on the communication complexity) and spend much time communicating. For PAB algorithms, though they may require time to propagate constraints among processors (this is an overhead that cannot be avoided but which would become insignificant for larger search spaces), when the problem is hard enough, they may out perform their sequential equivalents.

b) To find more or all solutions, PAB algorithms will perform better than their DAB and sequential equivalents. The reason for this is that a parallel agent based algorithm exhausts the search space using several processors, each in a different part of the search tree, whereas the sequential algorithm must exhaust the search space alone. PAB algorithms also use little communication and use the parallel computing power in a balanced way.

c) For DAB, when the ratio of useful partial solutions to all generated partial solutions is higher, the search speed of the overstep search algorithm may be faster than its equivalents without the overstep search mechanism, especially if only one solution is required. However, when the ratio is lower, the overstep search algorithm may be a little slower. The density of the problem may have a significant impact on the over-step search ratio. When the density of the problem is high, the overstep search algorithm has a better chance of speeding up search. This gives us a hint that the over-step search algorithm should only be used when the problem has a 'high' density.

d) Furthermore, in these tests, PAB algorithms are generally faster than DAB algorithms for solving the problems. A reason for this may be that the problems tested are not naturally distributed. When DAB algorithms are used to solve them, the problems may cause serious communication bottlenecks and load imbalance.

e) There is little difference in checks and backtracks performed by the sequential and FAB algorithms. However, the latter may perform better due to parallel domain filtering.

Initial tests [18] using a negotiation strategy to control conflict resolution showed that the extra message passing incurred, and overheads from mechanisms to prevent problems such as processing loops, caused the algorithm to take considerably longer to find a solution than other DAB methods. In fact, for the 120 orders of the Zebra problem the negotiation algorithm was between 8 and 12 times slower than DEFC-CBJ alone. It is also very difficult, when not using some global knowledge, to have an algorithm that will find all solutions to a problem.

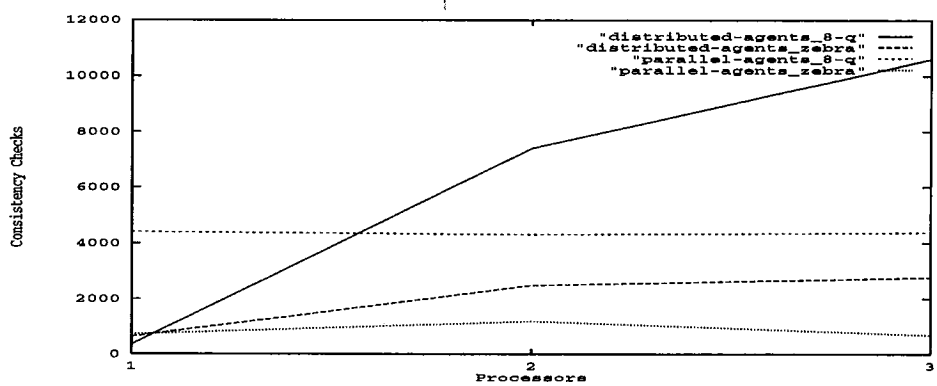

Figure 4: **Load Balancing for Distributed and Parallel-agent-based algorithms**

3.2 Parallel Processing and Search Overheads

Table 4: **Parallel Processing and Search Overheads**

Algorithms	Prob	NC	NB	NM	Time[s]	S	E	LB	S_o
Sequential ALGs (one processor)									
ALG-DO	single Zebra	2322	270	n/a	1.9	1	1	87.5	1
ALG-DO	8-q	12702	1008	n/a	4.9	1	1	94.2	1
Distributed-agent-based ALGs (three processors)									
HY-ALG-DO	single Zebra	5887	563	775	4.64	0.41	0.14	51.9	2.54
HY-ALG-DO	8-q	18341	1540	2542	11.10	0.44	0.15	47.9	1.44
Parallel-agent-based ALGs (three processors)									
ALG-DO	single Zebra	2597	291	n/a	1.9	1	0.3	46.8	1.12
ALG-DO	8-q	13184	1027	n/a	2.8	1.75	0.58	70.7	1.04
Function-agent-based ALGs (three processors)									
ALG-DO	single Zebra	2364	270	n/a	2.59	0.73	0.24	75	1.02
ALG-DO	8-q	12702	1008	n/a	6.9	0.71	0.24	89.4	1

Table 5: **Comparison of Algorithms for problems with complicated checks**

Algorithms	Prob	NC	NB	NM	Time[s]	S	E	S_{simple}	E_{simple}
Sequential ALGs (one processor)									
ALG-DO	single Zebra	2322	270	n/a	10.16	1	1	1	1
ALG-DO	8-queens	12702	1008	n/a	48.23	1	1	1	1
Distributed-agent-based ALGs (three processors)									
HY-ALG-DO	single Zebra	5538	518	682	12.69	0.8	0.27	0.41	0.14
HY-ALG-DO	8-queens	17955	1485	2409	51.87	0.93	0.31	0.44	0.15
Parallel-agent-based ALGs (three processors)									
ALG-DO	single Zebra	2597	291	n/a	5.75	1.77	0.58	1	0.3
ALG-DO	8-queens	13273	1030	n/a	18.25	2.64	0.88	1.75	0.24
Function-agent-based ALGs (three processors)									
ALG-DO	single Zebra	2364	270	n/a	8.88	1.14	0.38	0.73	0.24
ALG-DO	8-queens	12702	1008	n/a	47.83	1.01	0.37	0.71	0.24

Processors executing a parallel or distributed search algorithm incur a number of overheads. These include the communication overheads, idle time due to load imbalance and contention over shared memory structures. Clearly, due to these overheads the speedup should be less than P, if both the sequential and parallel formulations of an algorithm do the same amount of work.

Table 6: **Scalability of DAB on MEIKO Computing Surface for DEFC-CBJ to find the first solution of Zebra against 120 orders**

Processors	1	5	15
NC	2752	3140	3599
NM	2791	2904	3698
Time[s]	7.844	2.869	5.030
S	1	2.73	1.56

Table 7: **Scalability of Algorithms on SEQUENT**

Methods	**One Agent**	**Distributed-agent**		**Parallel-agent**		**Function-agent**	
Algorithms	**ALG-DO**	**HY-ALG-DO**		**ALG-DO**		**ALG-DO**	
Processors	1	2	3	2	3	2	3
Zebra (first solution) against 120 orders							
NC	268	646	1022	686	816	270	270
NM	n/a	50	154	n/a	n/a	n/a	n/a
Time[s]	17	49.48	74	36.66	30	23.26	20
S	1	0.34	0.23	0.46	0.57	0.73	0.85
Zebra (all solutions) against 120 orders							
NC	2322	7386	8942	3920	4059	2363	2363
NM	n/a	593	1542	n/a	n/a	n/a	n/a
Time[s]	207	506.74	549	217.33	160	222.92	221
S	1	0.41	0.38	0.95	1.29	0.93	0.94
8-queens (one solution)							
NC	795	1097	1138	280	420	795	795
NM	n/a	58	158	n/a	n/a	n/a	n/a
Time[s]	0.6	1.34	1.35	0.8	0.89	1.30	1.18
S	1	0.45	0.44	0.75	0.67	0.46	0.51
8-queens (all solutions)							
NC	12702	18195	18341	12702	13184	12702	12702
NM	n/a	878	2542	n/a	n/a	n/a	n/a
Time[s]	4.9	10.73	11.10	3.73	2.8	6.9	6.9
S	1	0.46	0.44	1.31	1.75	0.71	0.71

For search algorithms, the amount of work done by a parallel formulation is often different from that done by the sequential formulation, because the space searched by them may be different.

The amount of search in CSPs or DCSPs can be observed, without loss of generality, by the number of consistency checks performed by search algorithms, although consistency checks are only a part of the total search work. In this section, the amount of work refers to the number of consistency checks.

Let the amount of work done by a sequential processor be W. Now, if the total amount of work done by P distributed processors is W_p, then the search overhead S_o of the parallel system is defined as the ratio of the work done by the parallel formulation to that done by the sequential algorithm:

$$S_o = W_p/W$$

From Table 4, we can see DAB methods suffer higher communication overheads and S_o than the other two methods. Messages among processors and the search overheads of DAB algorithms effectively waste a great deal of CPU time.

Figure 4 shows the load balance curves of a DAB algorithm and a PAB algorithm to solve the Zebra and 8-queens problems. When solving these problems, a DAB algorithm causes a large load imbalance among processors. While the higher order processors may have too much work (consistency checks) to do, the lower order processors have almost nothing to do.

Table 8: **Scalability of PAB ALG-DO on SEQUENT for larger problems**

Processors	1	2	3
12-queens (one solution)			
NC	2113	2540	1782
Time[s]	1.65	1.27	1.20
S	1	1.3	1.38
12-queens (all solutions)			
NC	5311326	5311364	5315931
Time[s]	2458.33	1231.46	827.28
S	1	2	2.97

The above may further explain why, in most cases, PAB algorithms and FAB algorithms out perform DAB algorithms. They have less parallel processing and search overheads.

Parallel search involves the classical communication versus computation trade-off. As we discussed above, search overhead in many distributed and parallel algorithms is greater than 1, implying that the parallel form of the program does more work than the sequential form. We can reduce the search overhead for such formulations at the cost of increased communication, and vice-versa. For instance, if a search space is statically divided among different processors which independently search them without communicating, then they together will often perform much more search than one processor. This search overhead can often be reduced if processors are allowed to communicate, such as passing *dead ends* among processors.

In both the Zebra and n-queens problems, the consistency checks are relatively simple. However, in the real-world, the consistency checking can be complicated, such as solving differential equations. The definition of a constraint varies enormously, as pointed in [20]. It has been taken to mean a Boolean predicate, a fuzzy (or ill-defined) relation, a continuous figure of merit analogous to energy, an algebraic equation, an inequality, a Horn-clause in PROLOG, and various other arbitrarily complex relationships. Thus, the constraint check is one of the key factors that may impact on the performance of algorithms. How big is this factor? In Table 5, the complicated checks refer to 1000 operations of $a = b * c$ to be added when the algorithm performs each relation check. The test results show that for the problems with more complicated checks, algorithms for multi-processor platforms have a better chance of out performing their sequential equivalents. In this case, the communication overheads become less significant, when the computation demands are heavy due to the complicated checks.

3.3 Scalability

The scalability S_c of a parallel algorithm is determined by the ratio of the different parallel runtimes T_{p_j}, where P_j is the number of the parallel processors used, for a fixed problem size.

$$S_c = T_{p_j}/T_{p_i} \quad where \;\; P_j > P_i$$

If S_c is greater than one, then the algorithm is a scalable parallel algorithm from P_i to P_j for that fixed problem. In fact, it may be easy to decide the scalability by way of observing the speedups against the different numbers of processors used. For a scalable algorithm, the speedup should become greater as more processors are used.

Table 6 presents the time to find the first solution for different versions of DEFC-CBJ on a MEIKO Computing Surface. The time to solution includes all overheads such as the initial propagation of constraints. The speedup shows the improvement over running the algorithm on one processor. The speedup for 5 processors is consistently higher than for 15 processors. This is due to the natural topology of the problem. With 5 processors it is easy to put one group on each processor. This has the advantage that most communication takes place within groups with only a small percentage

being across groups. Therefore, most communication takes place within each processor whereas, for 15 processors, there is more inter-processor communication. These results show that the algorithm gives a speedup in excess of 2.5 on 5 processors and 1.56 on 15 processors.

Table 7 shows that DAB methods have poor scalability for these two problem types (Zebra and 8-queens). However, from Table 8, PAB algorithms may have a (nearly) linear scalability when the problem is hard enough and it needs to find more than one solution. Table 7 shows that for small domain problems, such as 8-queens and Zebra, the scalability of the FAB algorithms may be quite poor. When the problem domain size becomes larger, such as the 25 queens problem in Table 3, the scalability of this method improves.

4 CONCLUSION

Table 9: **Characteristics of Basic DPS Strategies**

Characteristics	Distributed-agent		Parallel-agent		Function-agent
Prefered Problems	naturally distributed		tightly coupled		tightly coupled
Algorithm Design	specially designed		any sequential		any sequential
Search Space Type	shared	shared	complete	disjoint	single
Cooperation	necessary	necessary	beneficial	beneficial	n/a
Control Type	central	negotiation	n/a	n/a	n/a
Memory type	both	both	both	both	shared
Communication Cost	medium	high	lower	lower	n/a
Load Balancing	poor	fair	good	good	good
Heuristic scope	local	local	global	global	global
Dynamic Change	easy	easy	easy	difficult	easy
Scalability	poor	poor	fair	good	reasonable
Termination Detection	difficult	difficult	easy	easy	easy
Find a solution	poor	poor	fair	good	good
Find more solutions	poor	poor	good	excellent	fair

Table 9 summaries the properties the methods described in this paper.

From the table it can be seen that PAB strategy is very versatile. However it suffers from not coping easily with dynamic changes to the problem if one agent has only a disjoint search space. This is quite important as many real-world problems naturally require reactions to dynamic change.

DAB algorithms are generally slower than other methods but, if given the correct problem type and fast communication channels, may perform better. For some naturally distributed problems they may be the only algorithms that could be used.

Acknowledgement: This work is partially supported by the U.K. Science and Engineering Research Council (GR/F82733).

REFERENCES

[1] A Bond and L Gasser. *Readings in Distributed Artificial Intelligence.* Morgan, Kaufmann, 1988.

[2] Bernard Burg. Parallel forward checking parts 1 and 2. Technical Report TR-594, TR-595, Institute for New Generation Computer Technology, Japan, 1990.

[3] S H Clearwater, B A Huberman, and T Hogg. Cooperative solution of constraint satisfaction problems. *Science*, 254:1181–1183, 1991.

[4] Zeev Collin and Rina Dechter. A distributed solution to the network consistency problem. Technical report, Israel Institute of Technology, 1990.

[5] S E Conry, K Kuwabara, and V R Lesser. Multistage negotiation for distributed constraint satisfaction. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1462–1477, 1991.

[6] R Dechter. Learning while searching in constraint-satisfaction problems. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 178–183, Menlo Park, Calif., 1986.

[7] R Dechter. Constraint processing incorporating, backjumping, learning, and custer-decomposition. In *Proceedings of CAIA-88*, pages 312–319, 1988.

[8] E C Freuder and M J Quinn. Parallelism in algorithms that take advantage of stable sets of variables to solve constraint satisfaction problems. Technical Report Tech Rep 85-21, University of New Hampshire, Durham, New Hampshire, 1985.

[9] Anath Y Grama and Vipin Kumar. Parallel processing of discrete optimization problems: A survey. Technical report, Dept of Computer Science, Univ of Minnesota, Minneapolis, November 1992.

[10] D G Griffiths and C Whitney. Fundamentals of distributed artificial intelligence. *Br Telecom Technol J*, 9(3):88–96, July 1991.

[11] M Huhns. *Distributed Artificial Intelligence*, volume I and II. Morgan, Kaufmann, 1987.

[12] L M Kirousis. Fast parallel constraint satisfaction. *Artificial Intelligence*, 64:147–160, 1993.

[13] V Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, pages 32–44, Spring 1992.

[14] Q Y Luo, P G Hendry, and J T Buchanan. Extending algorithms for constraint satisfaction problems. Technical Report KEG-5-92, Dept of Computer Science, Univ of Strathclyde, Scotland, June 1992.

[15] Q Y Luo, P G Hendry, and J T Buchanan. A hybrid algorithm for distributed constraint satisfaction problems. In *Parallel Computing: From Theory to Sound Practice*, pages 164–175, Barcelona, Spain, March 1992. IOS Press.

[16] Q Y Luo, P G Hendry, and J T Buchanan. A new algorithm for dynamic distributed constraint satisfaction problems. In *Proceedings of the Fifth Florida Artificial Intelligence Research Symposium 92*, pages 52–56, Florida, USA, April 1992.

[17] Q Y Luo, P G Hendry, and J T Buchanan. Comparison of different approaches for solving distributed constraint satisfaction problems. In *Proceedings of the AAAI 1993 Spring Symposium: Innovative Applications of Massive Parallelism*, pages 150–159, Stanford University, USA, March 1993.

[18] Q Y Luo, P G Hendry, and J T Buchanan. Heuristic search for distributed constraint satisfaction problems. Technical Report KEG-6-93, Dept of Computer Science, Univ of Strathclyde, Scotland, February 1993.

[19] Q Y Luo, P G Hendry, and J T Buchanan. Methods used to implement an integrated distributed scheduler. In *Proceedings of the CKBS-SIG workshop 1993*, pages 47–60, Unversity of Keele, UK, September 1993.

[20] W Meyer. *Expert Systems in Factory Management Knowledge-based CIM*. Ellis Horwood, 1990.

[21] Y Nishibe, K Kuwabara, and T Ishida. Effects of heuristics in distributed constraint satisfaction: Towards satisficing algorithms. In *Proceedings of the 11th International Workshop for DAI*, 1992.

[22] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. Technical Report AISL-46-91, Dept of Computer Science, University of Strathclyde, Scotland, 1991. Also in Computational Intelligence, 9(3):268-299, 1993.

[23] V N Rao and V Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, 1993.

[24] K Sycara, S Roth, N Sadeh, and M Fox. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1446–1461, 1991.

[25] M Yamaguchi, K Hagiwara, and N Tokura. On upper bounds of communication complexities and ideal time complexities of distributed algorithm. In *COMP86-83*, pages 17–28, 1987.

[26] M Yokoo and E Durfee. Distributed search formalisms for distributed problem solving: Overview. In *Proceedings of the Eleventh International Workshop on Distributed Artificial Intelligence*, pages 371–390, Glen Arbor, MI, February 1992.

[27] Makoto Yokoo. Constraint relaxation in distributed constraint satisfaction problems. In *5th IEEE International Conference on Tools with Artificial Intelligence*, 1993.

[28] Makoto Yokoo. Dynamic variable/value ordering heuristics for solving large-scale distributed constraint satisfaction problems. In *12th International Workshop on Distributed Artificial Intelligence*, 1993.

[29] Makoto Yokoo, Toru Ishida, and K Kuwabara. Distributed constraint satisfaction for dai. In *Proceedings of the 10th International Workshop for DAI*, 1990.