Concurrent Backjumping for Distributed CSPs

Roie Zivan and Amnon Meisels *

Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel
{zivanr,am}@cs.bgu.ac.il

Abstract

*A distributed concurrent search algorithm for solving distributed constraint satisfaction problems (*DisCSPs*) is presented. Concurrent search algorithms are composed of multiple search processes (*SPs*) that operate concurrently and scan non-intersecting parts of the global search space. Each* SP *is represented by a unique data structure, containing a current partial assignment (CPA), that is circulated among the different agents. Search processes are generated* dynamically, *starting by the initializing agent, and also by any number of agents during search.*

In the proposed algorithm, all search processes perform conflict-based backjumping concurrently. This results in two advantages:

- *The search on each sub-space is more efficient*
- *Backjumping in a single search space can detect multiple unsolvable search spaces, thus sharing information among search processes*

Experimental evaluation of the $ConcCBJ$ algorithm, on randomly generated DisCSPs*, is presented. $ConcCBJ$ outperforms both its simpler (backtracking) version and asynchronous backtracking (ABT) [Yokoo2000].*

1. Introduction

Distributed constraint satisfaction problems (*DisCSP*s) are composed of agents, each holding its local constraints network, that are connected by constraints among variables of different agents. Agents assign values to variables, attempting to generate a locally consistent assignment that is also consistent with all constraints between agents (cf. [Yokoo2000, Solotorevsky et. al.1996]). To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages with other agents, to check consistency of their proposed assignments against constraints with variables owned by different agents. Following common practice, it is assumed that an agent can send messages to any one of the other agents [Yokoo2000, Meseguer and Jimenez2000, Bessiere et. al.2001].

Distributed CSPs are an elegant model for many every day combinatorial problems that are distributed by nature. Take for example a large hospital that is composed of many wards. Each ward constructs a weekly timetable assigning its nurses to shifts. The construction of a weekly timetable involves solving a constraint satisfaction problem for each ward. Some of the nurses in every ward are qualified to work in the *Emergency Room*. Hospital regulations require a certain number of qualified nurses (e.g. for Emergency Room) in each shift. This imposes constraints among the timetables of wards and generates a complex Distributed CSP [Solotorevsky et. al.1996].

A search procedure for a consistent assignment of all agents in a $DisCSP$, is a distributed algorithm. An intuitive way to make the distributed search process on DisCSPs efficient is to enable agents to compute concurrently. One method for achieving concurrency in search on DisCSPs is to enable agents to search asynchronously. In asynchronous search all agents compute assignments to their variables concurrently (asynchronously) and cooperate in a single search process over the DisCSP.

Several asynchronous search algorithms for *DisCSP*s have been proposed in the last decade [Yokoo et. al.1998, Yokoo2000, Bessiere et. al.2001]. The common feature of all asynchronous search algorithms is that agents process their assignments asynchronously, even if their local assignments are not consistent with other agents' assignments. In order to make asynchronous backtracking correct and complete asynchronous search algorithms keep data structures for nogoods that are discovered during search (cf. [Bessiere et. al.2001]).

A different way of achieving concurrency for search is to enable the agents to run multiple search processes concurrently on a DisCSP [Zivan and Meisels2002]. The concurrent backtracking algorithm (ConcBT), which we presented in a previous paper, distributes among agents a dynamically changing number of search processes [Zivan and Meisels2004]. All agents participate in each search process, since each agent holds some variables of the search space. Agents generate and terminate search processes dynamically during the run of the $ConcBT$ algorithm [Zivan and Meisels2004]. Each of the search processes of $ConcBT$ circulates among all agents a form of message that includes a consistent partial assignment of multiple agents. When such a message includes a complete assignment, to all variables of all agents, the search process stops and the solution is reported.

The present paper proposes an algorithm that *runs multiple backjumping search processes* on DisCSPs. The proposed $ConcCBJ$ algorithm performs conflict-based backjumping (CBJ) on each of its concurrent sub-search spaces. There are two major advantages to concurrent CBJ search. First, the efficiency of each sub search process is improved. Second, the performance of backjumping in one search space can indicate that other search spaces are unsolvable enabling early termination of the search processes exploring them (Section 3). The concurrency of the $ConcCBJ$ algorithm is achieved by the circulation of multiple Current Partial Assignments(*CPA*s) among all agents. Each *CPA* represents one search process (SP) and each search process scans a different part of the global search space. The search space is split dynamically at different points on the path of the search process by agents generating additional CPAs. This changes dynamically the degree of concurrency during search and enables automatic load balancing. The splitting and re-splitting of the search space is performed independently by agents and is thus a distributed process.

In order to describe the $ConcCBJ$ algorithm we first introduce concurrent search in its simplest form in section 2. In section 3 we present the changes to simple concurrent search, to enable it to perform Conflict-based Backjumping (CBJ). The method for terminating SPs which are shown to be unsolvable is also presented in section 3. A correctness and completeness proof for concurrent search is outlined in section 4. Section 5 presents experimental evaluations, which demonstrate the advantage of $ConcCBJ$ over the original $ConcBT$ algorithm. A comparison of $ConcCBJ$ to asynchronous backtracking (ABT) [Yokoo2000, Bessiere et. al.2001] is also presented in Section 5. *ConcCBJ* outperforms *ABT* by a large factor and its advantage is more pronounced for harder problem instances. Our conclusions on the advantages of using multiple CBJ search processes in DisCSP search are summarized in section 6.

2. Concurrent Search

A concurrent search algorithm performs multiple concurrent backtrack search processes on disjoint parts of the *DisCSP* search-space. Each search space includes all variables and therefore involves all agents. Each agent holds a set of data structures, one for each search process. These data structures, which we term SPs, include all the relevant data for the state of the agent on each of the search processes. Agents in concurrent search algorithms pass their assignments to other agents on a *CPA* (Current Partial Assignment) data structure. Each *CPA* represents one search process, and holds the agents' current assignments in the corresponding search process. An agent that receives a *CPA* tries to assign its local variables with values that are not conflicting with the assignments already on the *CPA*, using only the current domains in the *SP* that is related to the received *CPA*. The uniqueness of the *CPA* for every search space ensures that assignments are not done concurrently in a single sub-search-space.

An agent can generate a set of CPAs that split the search space of a single CPA that passed through that agent, by splitting the domain of one of its variables. Agents can perform splits independently and keep the resulting data structures (*SP*s) privately. All other agents need not be aware of the split, they process all $CPAs$ in exactly the same manner (see section 2.2). *CPA*s are created either by the Initializing Agent (*IA*) at the beginning of the algorithm run, or dynamically by any agent that splits an active search-space during the algorithm run. A simple heuristic of counting the number of times agents pass the *CPA* in a sub-search-space (without finding a solution), is used to determine the need for re-splitting of search-spaces. This generates a nice mechanism of load balancing, creating more search processes on heavily backtracked search spaces.

A backtrack operation is performed by an agent which fails to find a consistent assignment in the search-space corresponding to the partial assignment on the *CPA*.

2.1. Main objects of Concurrent search

The main data structure that is used and passed between the agents is a *current partial assignment (CPA)*. A *CPA* contains an ordered list of triplets $< A_i, X_j, val >$ where A_i is the agent that owns the variable X_j and val is a value, from the domain of X_j, assigned to X_j. This list of triplets starts empty, with the agent that initializes the search process, and includes more assignments as it is passed among

the agents. Each agent adds to a *CPA* that passes through it, a set of assignments to its local variables that is consistent with all former assignments on the *CPA*. If successful, it passes the *CPA* to the next agent. If not, it *backtracks*, by sending the *CPA* to the agent from which it was received. Splitting the search space on some variable divides the values in the domain of this variable into several groups. Each sub-domain defines a unique sub-search-space and a unique *CPA* traverses this search space (for a detailed example of dynamic splitting see [Zivan and Meisels2004]).

Every agent that receives a *CPA* for the first time, creates a local data structure which we call a *search process (SP)*. This is true also for the initializing agent (*IA*), for each created *CPA*. The *SP* holds all data on current domains for the variables of the agent, such as the remaining and removed values during the path of the *CPA*.

The structure of the *ID* of a *CPA* and its corresponding *SP* is a pair $< A, j >$, where A is the ID of the agent that created the CPA and j is the number of CPAs this agent created so far. The *ID* of CPAs enables all agents to create *CPA*s independently, with a unique *ID*. This is the basis for dynamic splitting of the search space. When a split is performed during search, all CPAs generated by the agent that performs the split have a unique ID and carry the ID of the CPA from which they were split.

2.2. Algorithm description

In order to present concurrent search algorithms we use the following terminology:

- $CPA\_generator$: Every *CPA* carries the *ID* of the agent that created it.
- $steps\_limit$: the number of steps (from one agent to the next) that will trigger a split, if the *CPA* does not find a solution, or does not return to its generator.
- $split\_set$: the set of SP-IDs, stored in each SP after a split has been performed, including the IDs of the active SPs that were split from the SP by the agent holding it.
- $origin\_SP$: an agent that performs a dynamic split, holds in each of the new *SPs* the *ID* of the *SP* it was split from (i.e. of $origin\_SP$). An analogous definition holds for $origin\_CPA$. The $origin\_SP$ of an *SP* that was not created in a dynamic split operation is its own *ID*.

The messages exchanged by agents in concurrent search are the following:

- **CPA** - a regular CPA message.
- **backtrack_msg** - a CPA sent in a backtrack operation.

- **ConcBT**:
  1. done ← false
  2. **if**(IA) **then** $initialize\_SPs$
  3. **while**(**not** done)
  4. **switch** msg.type
  5. $split$: perform_split
  6. $stop$: done ← true
  7. CPA: receive_CPA
  8. $backtrack$: receive_CPA
- **initialize_SPs**:
  1. **for** i ← 1 to $domain\_size$
  2. create_SP(i)
  3. domain_SP[i] ← first_val[i]
  4. CPA ← create_CPA(i)
  5. $assign\_CPA$
- **receive_CPA**:
  1. CPA ← $msg.CPA$
  2. **if**(first_received(CPA_ID))
  3. create_SP(CPA_ID)
  4. **if**($CPA\_generator$ = ID)
  5. CPA_steps ← 0
  6. **else**
  7. CPA_steps ++
  8. **if**(CPA_steps = $steps\_limit$)
  9. send($split\_msg$, $CPA\_generator$)
  10. **if**(msg.type = $backtrack\_msg$)
  11. remove_last_assignment
  12. $assign\_CPA$
- **assign_CPA**:
  1. CPA ← assign_local
  2. **if**(is_consistent(CPA))
  3. **if**(is_full(CPA))
  4. $report\_solution$
  5. stop
  6. **else**
  7. send(CPA, next_agent)
  8. **else**
  9. $backtrack$

Figure 1. Main and Assign parts of ConcBT

- **stop** - a message indicating the end of the search.
- **split** - a message that is sent in order to trigger a split operation. Contains the *ID* of the *SP* to be split.
- **unsolvable** - a message used by the ConcCBJ algorithm, to indicate an unsolvable sub search space.

The simplest concurrent search algorithm is Concurrent Backtracking ($ConcBT$) [Zivan and Meisels2004]. The main function of the algorithm and functions that perform assignments on the CPA when it moves forward are presented in (Figure 1).

- The main function **ConcBT**, initializes the search if it is run by the *initializing agent (IA)*. It initializes the algorithm by creating multiple *SPs*, assigning each SP with one of the first variable's values. After initialization, it loops forever, waiting for messages to arrive.

- **receive_CPA** first checks if the agent holds a *SP* with the *ID* of the *current_CPA* and if not, creates a new *SP*. If the CPA is received by its generator, it changes the value of the steps counter (CPA_steps) to zero. This prevents unnecessary splitting. Otherwise, it checks whether the *CPA* has reached the $steps\_limit$ and a split must be initialized (lines 7-9). Before assigning the CPA a check is made whether the CPA was received in a $backtrack\_msg$, if so the previous assignment of the agent which is the last assignment made on the CPA is removed, before $assign\_CPA$ is called (lines 10-11).
- **assign_CPA** tries to find an assignment for the local variables of the agent, which is consistent with the assignments on the $current\_CPA$. If it succeeds, the agent sends the *CPA* to the selected $next\_agent$ (line 7). If not, it calls the *backtrack* method (line 9).

The rest of the functions of the ConcBT algorithm are presented in Figure 2.

- The **backtrack** method is called when a consistent assignment cannot be found in a *SP*. Since a split might have been performed by the current agent, a check is made, whether all the *CPA*s that were split from the $current\_CPA$ have also failed (line 2). When all split CPAs have returned unsuccessfully, a backtrack message is sent carrying the ID of the $origin\_CPA$. In case of an *IA*, the *SP* and the corresponding *origin_CPA* are marked as a failure (lines 3-4). If all other *CPA*s are marked as failures, the search is ended unsuccessfully (line 6).
- The **perform_split** method tries to find in the *SP* specified in the $split\_message$, a variable with a non-empty current_domain. It first checks that the CPA to be split has not been sent back already, in a backtrack message (line 1). If it does not find a variable for splitting, it sends a split_message to $next\_agent$ (lines 8-9). If it finds a variable to split, it creates a new *SP* and CPA, and calls *assign_CPA* to initialize the new search (lines 3-5). The *ID* of the generated CPA is added to the split set of the divided SPs $origin\_SP$ (line 6).

The algorithm ends unsuccessfully, when all *CPA*s return for backtrack to the IA and the domain of their first variable is empty. The algorithm ends successfully if *one CPA contains a complete assignment*, a value for every variable in the DisCSP.

3. Conflict-based Backjumpig

The method of backjumping that is used in the proposed algorithm is based on $Dynamic\ Backtrack$

- **backtrack**:
  1. delete(current_CPA from $origin\_split\_set$)
  2. **if**($origin\_split\_set$ is_empty)
  3. **if**(IA)
  4. CPA ← no_solution
  5. **if**(no_active_CPAs)
  6. report_no_solution
  7. stop
  8. **else**
  9. send($backtrack\_msg$, $last\_assignee$)
  10. **else**
  11. $mark\_fail(current\_CPA)$
- **perform_split**:
  1. **if**(**not_backtracked**(CPA))
  2. var ← $select\_split\_var$
  3. **if**(var is_not null)
  4. create_split_SP(var)
  5. create_split_CPA(SP_ID)
  6. add(CPA_ID to $origin\_split\_set$)
  7. $assign\_CPA$
  8. **else**
  9. send($split\_msg$, $next\_agent$)
- **stop**:
  1. send(stop, all_other_agents)
  2. done ← true

Figure 2. Backtrack and Split for ConcBT

[Ginsberg1993, Bessiere et. al.2001]. Each agent that removes a value from its current domain stores the partial assignment that caused the removal of the value. When the current domain of an agent empties, the agent constructs a backtrack message from the union of all assignments in its stored removal explanations. The backtrack message is sent to the agent which is the owner of the lowest priority variable in the inconsistent partial assignment.

In every sub search space all tuples of assignments share the head of the assignment. Thus for every sub-search-space we define:

Definition 3.1 *A* Common Header *(CH) is the maximal prefix of assignments which is included in all partial assignments in a sub-search-space.*

In a concurrent backtracking algorithm the inconsistent partial assignments are composed of all assignments prior to the current empty domain. When backjumping is performed, an inconsistent subset of former assignments is constructed, which forms a nogood (or an explanation) [Ginsberg1993]. A short nogood can belong to multiple search spaces, all of which contain no solution and are thus unsolvable. In order to terminate the corresponding search processes, an agent that receives a backtrack message performs the following procedure:

- detect the SP to which the received CPA either belongs or was split from.
- check if the SP was split.
- if it was:
  - send an $unsolvable$ message which carries the ID of the SP found not to contain a solution. Send the message to the agent to whom the related CPA was last sent.
  - choose a new unique ID for the CPA received and its related SP.
  - check if there are other SPs which contain the inconsistent partial assignment received, and send corresponding $unsolvable$ messages.
  - continue the search using the SP and CPA with the new ID.

The change of ID makes the process independent of whether the backtrack message included the *original CPA* or one of its split offsprings.

An agent that receives an $unsolvable$ message performs the following operations for the unsolvable SP and each of the SPs split from it:

- mark the SP as unsolvable.
- send an $unsolvable$ message which carries the ID of the SP to the agent to whom the related CPA was last sent.

Agents that receive a CPA first check if the related SP was not marked $unsolvable$. If so they terminate the CPA and its related SP.

Figure 3 presents the main method ConcCBJ, the methods that were changed from the $ConcBT$ algorithm in Figure 1and two additional methods. Line 9 in the main method calls procedure $mark\_unsolvable$ when an $unsolvable$ message is received.

In method $receive\_CPA$ a check is made in lines 3,4 if the SP related to the received CPA is marked unsolvable. In such a case the CPA is not assigned and the related SP is terminated. For a backtracking CPA (lines 13-18) a check is made whether the SP was split by agents who received the CPA after this agent (line 15). If so, the termination of the unsolvable SP is initiated by sending an $unsolvable$ message. A new ID is assigned to the received CPA and its related SP (line 17). Before calling $assign\_CPA$, a check is made whether there are other SPs which can be declared unsolvable. This can happen when the head of their partial assignment (their CH) contains the received inconsistent partial assignment.

- **ConcCBJ**:
  1. done ← false
  2. **if**(IA) **then** $initialize\_SPs$
  3. **while**(**not** done)
  4. **switch** msg.type
  5. $split$: perform_split
  6. $stop$: done ← true
  7. CPA: receive_CPA
  8. $backtrack$: receive_CPA
  9. $unsolvable$: mark_unsolvable
- **receive_CPA**:
  1. CPA ← $msg.CPA$
  2. **if**(unsolvable SP)
  3. terminate CPA
  4. else

  ..
  ..
  ..

  13. **if**(msg.type = $backtrack\_msg$)
  14. remove_last_assignment
  15. **if**(SP_split_ahead)
  16. send(unsolvable, SP.next_agent)
  17. rename_SP
  18. check_SPs(CPA.inconsistent_assignment)
  19. $assign\_CPA$
- **backtrack**:
  1. delete(current_CPA from $origin\_split\_set$)
  2. **if**($origin\_split\_set$ is_empty)
  3. **if**(IA)
  4. CPA ← no_solution
  5. **if**(no_active_CPAs)
  6. report_no_solution
  7. stop
  8. **else**
  9. $backtrack\_msg \leftarrow inconsistent\_assignment$
  10. send($backtrack\_msg$, $lowest\_priority\_assignee$)
  11. **else**
  12. $mark\_fail(current\_CPA)$
- **mark_unsolvable**
  1. mark msg.SP unsolvable
  2. send(unsolvable, next_agent)
  3. **for each** split_SP
  4. mark split_SP unsolvable
  5. send(unsolvable, next_agent)
- **check_SPs**(inconsistent_assignment)
  1. **for each** of the agent's SPs
  2. **if**(SP.contains(inconsistent_assignment))
  3. send(unsolvable, SP.next_agent)

Figure 3. Methods for CBJ

In method $backtrack$, the agent inserts the culprit inconsistent partial assignment into the backtrack message (line 9) before sending it back in line 10.

Method $mark\_unsolvable$ is part of the mechanism for

terminating SPs on unsolvable search spaces. The agent marks the SP related to the message received as unsolvable, and sends unsolvable messages to the agents to whom the CPA of this SP, and any other CPA split from it, were sent.

4. Correctness of ConcCBJ

A central fact that can be established immediately is that agents send forward only consistent partial assignments. This fact can be seen at lines 1, 2 and 7 of procedure *assign_CPA* (Figure 1). This implies that agents process, in procedures *receive_CPA* and *assign_CPA*, only consistent CPAs. Since the processing of CPAs in these procedures are the only means for extending partial assignments, the following lemma holds:

Lemma 4.1 *ConcCBJ extends only consistent partial assignments. The partial assignments are received via a CPA and extended and sent forward by the receiving agent.*

The correctness of ConcCBJ includes soundness and completeness. The soundness of ConcCBJ follows immediately from Lemma 4.1. The only lines of the algorithm that report a solution are lines 3, 4 of procedure *assign_CPA*. These lines follow a consistent extension of the partial assignment on a received CPA. It follows that a solution is reported *iff* a CPA includes a complete and consistent assignment.

In order to prove the completeness of the ConcCBJ algorithm we first outline the proof for the simpler concurrent backtrack version and then show that adding conflict based backjumping does not affect the completeness of the algorithm.

The main points of the completeness proof are the following:

- Completeness for the case of a single CPA, is equivalent to the proof of completeness for centralized backtrack by Kondrak and vanBeek [Kondrak and vanBeek1997].
- For several CPAs generated by the IA, the only difference from the $1-CPA$ case is in the data structures of the IA.
- Finally, it is shown that a dynamic split operation does not interfere with the correctness of the algorithm.

The reader is encouraged to look up the full version of the paper [Zivan and Meisels2004] for the full completeness proof, as well as for a detailed example of dynamic splitting.

For the completeness of ConcCBJ one continues as follows.

Lemma 4.2 *A sub-search-space whose CH includes an inconsistent subset of assignments does not include a solution to the DisCSP.*

The proof of lemma 4.2 derives from the method of constructing an inconsistent assignment in dynamic backtrack [Ginsberg1993, Bessiere et. al.2001]. A partial assignment is declared inconsistent only if it causes an empty domain in one of the variables. This implies that this partial assignment cannot be part of a solution. From definition 3.1 we derive that if a CH includes an inconsistent partial assignment it must be included in all the assignments in its related sub-search-space which means that none of these assignments is a solution to the $DisCSP$.

Lemma 4.3 *ConcCBJ does not terminate search-processes which lead to a solution.*

Only SPs that have a CH that is an extension of the CH that was found inconsistent are marked unsolvable. The search on these SPs is later terminated. Lemma 4.2 implies the proof for lemma 4.3.

It is immediately clear from lemma 4.3 that all partial assignments that lead to a solution will be extended and therefore the completeness of the ConcBT algorithm will not be affected by the addition of Conflict-Based Backjumping.

5. Experimental Evaluation

Experiments were conducted on random networks of constraints. The network of constraints, in each of the experiments, is generated randomly by selecting the probability p_1 of a constraint among any pair of variables and the probability p_2, for the occurrence of a violation among two assignments of values to a constrained pair of variables. Such uniform random constraints networks of n variables, k values in each domain, a constraints density of p_1 and tightness p_2 are commonly used in experimental evaluations of CSP algorithms (cf. [Prosser1996, Smith1996]). Experiments were conducted on networks with 10 agents ($n = 10$) and 10 values for each agent's variable ($k = 10$). In all of our experiments $p1 = 0.7$ and the value of the tightness p_2 is varied between 0.1 and 0.9, to cover all ranges of problem difficulty.

The common approach in evaluating the performance of distributed algorithms is to compare two independent measures of performance - time, in the form of steps of computation [Lynch1997, Yokoo2000], and communication load, in the form of the total number of messages sent [Lynch1997].

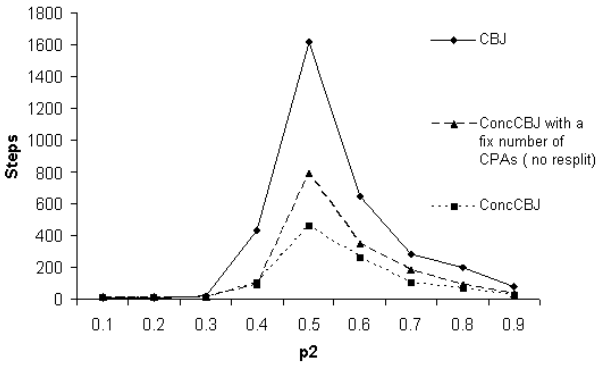

Figure 4. Number of steps in different versions of ConcCBJ, either 1-CPA, 5-CPAs, or dynamic number of CPAs

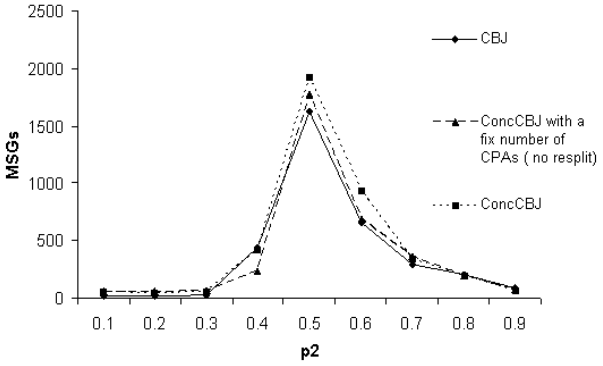

Figure 5. Total number of messages sent by the different versions of ConcCBJ

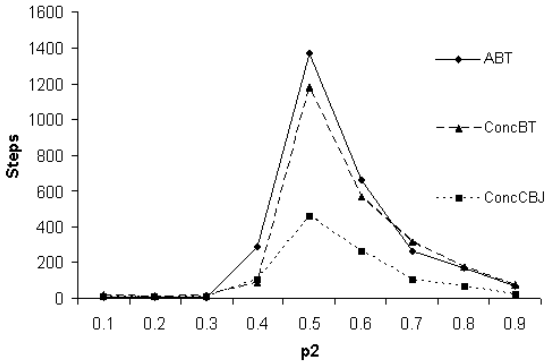

Figure 6. Steps performed by *ConcBT*, *ConcCBJ* and *ABT*.

5.1. Evaluation of concurrency

To investigate the effect of conflict based backjumping on concurrent search, one needs to compare the performance of ConcCBJ with the simpler algorithm ConcBT. It is also important to asses the impact of dynamic splitting of search spaces. To this end, the $ConcCBJ$ algorithm was run in a 1-*CPA* version, 5-*CPA* version and a 5-*CPA* version with dynamic re-splitting, using a step limit of 20. The 1-*CPA* version is completely sequential and serves as the baseline for comparison to the concurrent versions.

Figure 4 shows the computational effort in number of concurrent steps of computation to a solution, for all three versions. It is easy to see that concurrency improves the search efficiency and that dynamic resplitting improves it further. Figure 5 shows the results in total number of messages sent. ConcCBJ with dynamic splitting increases the number of traversing CPAs during search, but the effect on the total number of messages is minor.

5.2. Comparing to Asynchronous Backtracking

The performance of *ConcCBJ* can be compared to an asynchronous algorithm for solving DisCSPs, Asynchronous BackTracking (ABT) [Yokoo2000]. In the ABT algorithm agents assign their variables asynchronously, and send their assignments in $ok?$ messages to other agents to check against constraints. A fixed priority order among agents is used to break conflicts. Agents inform higher priority agents of their inconsistent assignment by sending them the inconsistent partial assignment in a $Nogood$ message. In our implementation of ABT, the $Nogoods$ are resolved and stored according to the method presented in [Bessiere et. al.2001]. Based on Yokoo's suggestions [Yokoo2000] the agents read, in every step, all messages received before performing computation.

Figure 6 presents the comparison of the number of steps performed by $ConcBT$, $ConcCBJ$ and ABT. Both concurrent algorithms use dynamic re-splitting with step-limit of 20. For the harder problem instances, $ConcBT$ performs slightly better than ABT. *ConcCBJ* outperforms ABT on these instances by a factor of 3.

With respect to the number of messages, $ConcCBJ$'s use of the network is very economic compared to ABT. Figure 7 presents the comparison of the of the total number of messages sent by the three algorithms. For the harder problem instances, ConcCBJ sends a factor of 5 fewer messages than ABT.

6. Discussion

The concurrent backjumping search algorithm ($ConcCBJ$) provides an efficient method for sev-

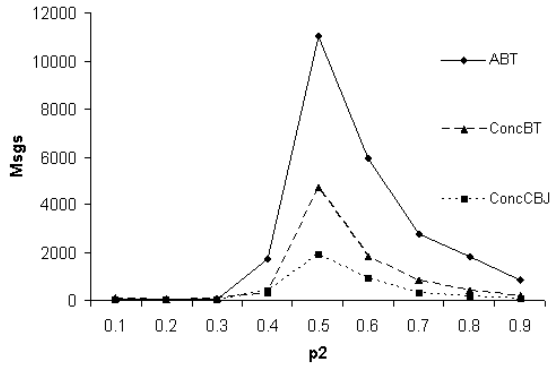

Figure 7. Number of messages sent by *ConcBT*, *ConcCBJ* and *ABT*.

eral search processes to search concurrently a DisCSP. The addition of Conflict Based Backjumping to concurrent search, enables early termination of search processes on sub-spaces which do not lead to a solution. An inconsistent subset can be found in one sub-space that rules out other sub-spaces as unsolvable.

Concurrent Backjumping is a distributed search algorithm that maintains multiple search processes concurrently. Its experimental behavior on random DisCSPs clearly indicates its efficiency, compared to algorithms of a single search process like ABT.

A dual model of concurrent search on distributed CSP is to think of every search process as an agent. Search process agents move from one DisCSP agent to another assigning the local variables of each DisCSP agent. The assignments are kept compatible with all former assignments. Every CPA is an active agent in this model. CPA agents traverse their sub search spaces on the DisCSP and can generate additional CPA agents dynamically. CPA agents that discover themselves unsolvable, terminate. This is a *Multi Search Agent* model for DisCSPs

Concurrent backtracking, as proposed in the present paper and in [Zivan and Meisels2004], may seem similar to former approaches of parallelism. Splitting the search space at the first agent and running several search processes for each of the values of the first agents' domain is part of interleaved search in [Hamadi2001]. There is, however, a major difference between ConcCBJ and IDIBT [Hamadi2002]. The interleaved parallel search algorithm runs multiple processes of an asynchronous search algorithm and its multiplicity is fixed at the start of its run [Hamadi2002]. The protocol of ConcCBJ enables it to perform dynamic splitting of the search space. Our experimental study shows that dynamic splitting of the search space improves the search by a meaningful factor, in contrast to $IDIBT$, where performance deteriorates for more than 2 contexts [Hamadi2002].

References

[Bessiere et. al.2001] C. Bessiere, A. Maestre and P. Messeguer. Distributed Dynamic Backtracking. *Proc. Workshop on Distributed Constraints, IJCAI-01*, Seattle, 2001.

[Fernandez et. al.2002] C. Fernandez, R. Bejar, B. Krishnamachari, K. Gomes Communication and Computation in Distributed CSP Algorithms. *Proc. Principles and Practice of Constraint Programming, CP-2002*, pages 664-679, Ithaca NY USA, July, 2002.

[Ginsberg1993] M. L. Ginsberg Dynamic Backtracking. *Artificial Intelligence Research*,1: 25-46, 1993

[Hamadi2001] Y. Hamadi. Distributed, Interleaved, Parallel and Cooperative Search in Constraint Satisfaction Networks. In *Proc. IAT-01*, Singappore, 2001.

[Hamadi2002] Y. Hamadi *Interleaved Backtracking in Distributed Constraint Networks*, *Intern. Jou. AI Tools*, 11: 167-188, 2002.

[Kondrak and vanBeek1997] G. Kondrak and P. vanBeek, *A Theoretical Evaluation of Selected Backtracking Algorithms*, *Artificial Intelligence*, 89: 365-87, 1997.

[Lynch1997] N. A. Lynch. Distributed Algorithms. Morgan Kaufmann Series, 1997.

[Meisels et. al.2002] A. Meisels et. al. *Comparing performance of Distributed Constraints Processing Algorithms*, *Proc. AAMAS-2002 Workshop on Distributed Constraint Satisfaction*, Bologna, July, 2002.

[Meseguer and Jimenez2000] Proc. CP-2000 Workshop on Distributed Constraint Satisfaction, Singapore, 22 September, 2000.

[Prosser1993] P. Prosser Hybrid Algorithms for the Constraint Satisfaction Problem *Computational Intelligence*, 9:268-199, 1993.

[Prosser1996] P. Prosser An empirical study of phase transition in binary constraint satisfaction problems *Artificial Intelligence*, 81:81-109, 1996.

[Smith1996] B. M. Smith. Locating the phase transition in binary constraint satisfaction problems. In *Artificial Intelligence*, 81:155-181, 1996.

[Solotorevsky et. al.1996] G. Solotorevsky, E. Gudes and A. Meisels. Modeling and Solving Distributed Constraint Satisfaction Problems (DCSPs). *Constraint Processing-96*, New Hamphshire, October 1996.

[Yokoo et. al.1998] M. Yokoo, E. H. Durfee, T. Ishida, K. Kuwabara. Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. on Data and Kn. Eng.*, 10(5): 673-685, 1998.

[Yokoo2000] M. Yokoo. Distributed Constraint Satisfaction Problems. Springer Verlag, 2000.

[Yokoo2000] M. Yokoo. Algorithms for Distributed Constraint Satisfaction: A Review. *Autonomous Agents & Multi-Agent Sys.*, 3(2): 198-212, 2000.

[Zivan and Meisels2002] A. Meisels et. al. Parallel Backtrack search on DisCSPs Proc. AAMAS-2002 Workshop on Distributed Constraint Satisfaction, Bologna, July, 2002.

[Zivan and Meisels2004] R. Zivan and A. Meisels. Concurrent Backtrack search on DisCSPs. to apear in *FLAIRS 2004*, May 2004. (full version http://www.cs.bgu.ac.il/˜zivanr)