Asynchronous Partitioning Framework

Vitaliy Freidovich\textsuperscript{1} and Amnon Meisels\textsuperscript{2},

\textsuperscript{1} Department of Mathematics and Computer Science,  
The Open University of Israel, Raanana, Israel  
f_vitaliy@yahoo.com  
\textsuperscript{2} Department of Computer Science,  
Ben-Gurion University of the Negev, Beer-Sheva, Israel  
am@cs.bgu.ac.il

Abstract. A new general framework for agent cooperation and coordination in solving distributed constraint satisfaction problems (DCSPs) is presented. The Asynchronous Partitioning Framework (APF) first partitions agents into groups of agents, based on some heuristic, prior to any search being conducted. During the partitioning one of the agents in each group is assigned the role of a group leader. Next, two distinct types of searches among the agents are performed in parallel. The first type of search is conducted within each group, in parallel and asynchronously to all searches in other groups. The second type of search, the global search, is conducted between the groups, and treats each group as if it is a single agent represented by its group leader. The structure of the groups remains static throughout the search processes. Two distinct algorithms implementing APF and a correctness proof are presented, and the advantages of APF are evaluated experimentally.

Keywords: Distributed Constraints Satisfaction; Distributed Search; Distributed Problem Solving; Distributed Partitioning; Distributed Framework.

1 Introduction

Distributed Constraint Satisfaction Problems (DCSPs) are composed of a set of agents. Each agent holds a set of variables, has a local representation of the constraint network, and is connected to other agents’ variables by some constraints. The role of an agent is to assign values to its variables, attempting to generate a locally consistent assignment. Consistency checks with external constraints are performed by exchanging messages with constraining agents to check consistency against their proposed local assignments. All agents cooperate in order to find a globally consistent assignment.

Several different algorithms for solving DCSPs have been proposed. The pioneering algorithm is Asynchronous Backtracking (ABT) [9]. In ABT agents are totally ordered and perform assignments to values in a completely asynchronous manner. Agents send new assignments to lower priority agents (e.g., ordered after them), and backtracking inconsistent partial assignments (Nogoods) to higher priority agents.
A more coordinated approach was proposed in the form of the Asynchronous Partial Overlay (APO) [3] algorithm. APO uses the concept of a mediator to centralize the search procedure in parts of the problem. Each mediator tries to solve a sub-problem which is centralized around it. When an agent is in conflict with other agents, it tries to assume the role of a mediator. If successful, the agent links to other agents, effectively extending the size of the sub-problem to be solved. In each step a mediator tries to extend the mediation session, to neighbors of agents that are inside the mediation session. Thus higher priority mediation sessions engulf lower priority mediation sessions during the search [1, 3].

The present paper proposes a new approach for solving DCSPs, based on the actual structure of a given DCSP as represented by its constraints graph. The Asynchronous Partitioning Framework (APF) adheres to the fail first principle [2] by concentrating computational efforts first on the most constrained regions of the constraints graph. This is achieved by partitioning the DCSP into groups, each representing such a region, and conducting local searches inside each group in parallel to other groups. The results of the local searches are combined into a complete solution by performing a global search between the groups.

APF performs backtracking synchronously inside each group. This is in contrast to ABT that performs backtracking between the agents in a completely asynchronous manner. However, backtracking against agents in different groups is asynchronous in APF. This makes APF a partially coordinated and partially asynchronous algorithm.

APF may seem similar to the APO algorithm. The linking step of APO, which is performed in a mediation session, actually generates a group, using the terminology of APF. However, the major difference is that in APO the partitioning mechanism generates dynamic groups which grow during the search process, while APF uses a static partition, where the groups remain without change during its search phase.

## 2 Asynchronous Partitioning Framework

The Asynchronous Partitioning Framework (APF) is presented in Figure 1. It is composed of the following 4 distinct components:

1. The group partitioning algorithm, denoted by GroupPartition, is responsible for partitioning a given set of agents into disjoint groups, each group having a single group leader.
2. The local search algorithm, LocalSearch, conducts a search inside each group.
3. The global search algorithm, GlobalSearch, conducts a global search among all the groups.
4. The coordination engine is the heart of APF, and coordinates the LocalSearch and the GlobalSearch algorithms.

### 2.1 Detailed Description

APF starts by partitioning the agents of a given DCSP into disjoint groups (line 1) and then initiates search for the solution in lines 2 and 18. Synchronization of search
1. GroupPartition(agents)
2. inside each group do:
3. LocalSearch(advance token via group leader; ignore constraints with lower priority groups)
4. if during LocalSearch an agent \(a_i\) changes its assignment
5. notify group leaders of all agents in lower priority groups that are connected to \(a_i\)
6. if during LocalSearch, or when in consistent or inconsistent state, a group leader receives notification regarding a change of value of an agent in a higher priority group, one of its agents, \(a_j\), is connected to
7. record the connection to the higher priority group
8. notify agent \(a_j\) regarding the new assignment
9. if the token has already passed agent \(a_j\)
10. regenerate the token by the agent view of \(a_j\)
11. reset assignments of all agents before \(a_j\) in accordance with the regenerated token
12. reset domains of all agents in the group
13. restart the search at agent \(a_j\)
14. if during LocalSearch a group leader is notified that the last agent in the group had found a consistent assignment
15. enter the consistent state
16. if during LocalSearch a group leader is notified that the first agent in the group cannot find a consistent assignment
17. enter the inconsistent state
18. in parallel to step (2) do:
19. GlobalSearch between the group leaders(use a global token; order the search by the priorities of the groups from highest to lowest; move global token only after the current group had determined its state)

**Fig. 1.** Asynchronous Partitioning Framework (APF)
which the original change of value from agent \(a_i\) was intended. Next, the group leader of \(a_i\) must act upon the location of the token within the group. If the token has never reached agent \(a_j\) during the local search, the group leader doesn't have to do anything, as the local search was not affected yet by the constraint with \(a_i\), since \(a_j\) didn't have the chance to remove any value from its domain due to \(a_i\)'s previous assignment. However, if the token did reach agent \(a_j\), and was then either advanced to the next agent, or backtracked to the previous agent, by the order of LocalSearch, the search must be restarted at \(a_j\).

Restarting the search at \(a_j\) allows APF to keep local consistency within the group. To restart the search at \(a_j\), the group leader first regenerates a new token (line 10), based on \(a_j\)'s agent view. Each agent has an agent view which records its local knowledge regarding assignments of other agents, and which is updated by the LocalSearch algorithm each time \(a_j\) receives the local token with the assignments that appear in the token. In line 11 the assignments of all agents preceding \(a_j\) by the order of LocalSearch, must be reset in accordance with \(a_j\)'s agent view, so that they would be consistent with the newly regenerated token. Then, in line 12 the domains of all the agents must be reset, as some values might have been deleted from them due to \(a_j\)'s previous assignment which has now changed. Finally, in line 13 the search is restarted at agent \(a_j\).

If the group leader discovers that the last agent by the order of LocalSearch, had managed to find a consistent assignment to its variable, all the intra-group constraints had been satisfied, and since LocalSearch is kept consistent with inter-group constraints as well, a partial assignment which is consistent with both intra and inter group constraints had been found, and the group leader can enter the consistent state at lines 14-15. In the opposite case, when the first agent by the order of LocalSearch had exhausted all possible assignments, the group leader enters the inconsistent state (lines 16-17).

Concurrently to the local searches performed inside each group, a global search, using the GlobalSearch algorithm, between the different groups is being conducted in lines 18-19 of APF. For this purpose a global token is used, which moves between the group leaders. The groups are ordered by the priorities assigned to them by the GroupPartition algorithm. Upon receiving a global token, each group leader advances or backtracks it, accordingly to the state of its group - consistent or inconsistent. Otherwise, if a local search is in progress, the group leader waits for it to enter one of these two states, before proceeding.

### 2.2 Required Conditions

Most of the component of APF - GroupPartition, LocalSearch and GlobalSearch – can be replaced by alternative algorithms. For the APF to be correct, these components must possess the following properties:

- The GroupPartition algorithm must:
  1. Form disjoint groups, such that each agent will belong to exactly one group.
  2. Generate a partition in which each group contains an agent which is connected to all other agents in that group.
  3. Elect exactly one agent in each group which is connected to all other agents in
the group, to be the group leader.
4. Assign a priority to each group, so that all the group priorities are injective.
5. Order the agents inside each group.

- The LocalSearch and the GlobalSearch algorithms has to be:
  1. Correct.
  2. Synchronous. Exactly one consistent partial assignment, extending the token, is performed at any given time.
- In addition, the LocalSearch algorithm must:
  3. Keep an agent view for each agent, recording assignments performed by constraining agents.
  4. Update agent views each time an agent receives a token, with the assignments that appear in the token.

2.3 Correctness Proof Summary

The following claims can be proven by examining the coordination engine of APF:
1. APF is sound. This claim follows from the facts that a consistent state of any LocalSearch implies that all agents inside the group are assigned consistently with the constraints between agents in the group and with agents in higher priority groups, and that the global token is advanced only after the current group has entered the consistent state, in accordance with the order of the groups.
2. APF is complete. This claim can be proven by examining (in negation) a consistent assignment which APF had failed to find. The claim is proven by examining the first group which failed to extend the appropriate part of the assignment, in contradiction to the completeness of the LocalSearch algorithm running inside that group.
3. APF terminates in a finite number of steps. This claim can be shown by examining the scenarios in which APF might fail to terminate. Since both LocalSearch and GlobalSearch are correct, this can occur only due to an infinite number of times APF restarts the search inside some group, at some agent \( a_i \). This situation can only be caused by an infinite number of messages received from higher priority agents. As the number of such agents is finite, some agent \( a_{i+1} \) belonging to a higher priority group, must send an infinite number of messages to \( a_i \). By simple induction it can be concluded that an infinite number of such agents must exist, each belonging to a higher priority group, in contradiction to the injectiveness of group priorities, as stated by property 4 of the GroupPartition algorithm.
4. The correctness of APF follows from the above 3 claims.

3 The AGP Algorithm

The Asynchronous Group Partitioning (AGP) algorithm is an implementation of the APF framework. It is composed of 4 phases:
1. Priority computation
2. Group partitioning
3. Group ordering
4. Search for a solution
Phases 1-3 are the implementation of the GroupPartition algorithm, and phase 4 implements the coordination engine of APF. The implementation of AGP synchronizes all agents to enter the next phase of the algorithm only after all the agents had completed the current phase.

### 3.1 Priority Computation Phase

The priorities of agents are calculated using a simple heuristic, which determines the priority of an agent based on the density of the agent's neighborhood in the constraints graph. The heuristic measures the density of the neighborhood of an agent $a_i$ by employing the formula:

$$P_{a_i} = k_i + \sum_{j=1}^{k_i} (\text{CommonNeighbors}(a_i, a_j))$$

where $P_{a_i}$ is $a_i$’s priority, $k_i$ is the number of $a_i$’s neighbors, and $\text{CommonNeighbors}(a_i, a_j)$ returns the number of agents which are neighbors of both $a_i$ and of $a_j$.

To compute the priorities in a distributed manner, an agent $a_i$ sends each neighbor $a_j$ a message which instructs it to check how many of the neighbors of $a_i$ are also its own neighbors in the constraints graph. $a_i$ then sums the results returned by each neighbor $a_j$, adds to the result $k_i$, and notifies all the agents of its computed priority.

Figure 2 (a) presents the results of the priority computation phase on an exemplary instance of a DCSP problem, the computed priority appears below each agent's id.

### 3.2 Group Partitioning Phase

Figure 3 presents the group partitioning phase of agents into disjoint groups, based on the priorities computed in the priority computation phase. A group leader is assigned
Fig. 3. Group partitioning phase

to each group.

**Detailed Description.** The group partitioning phase starts by running `find_group_leader`, which tries to find the appropriate agent in each agent’s neighborhood for being its group leader. The search is conducted according to the priorities of the different neighbors, which are sorted initially into the `sortedListOfNeighbors` list, equal priorities being resolved lexicographically. During this process each neighbor is fetched from `sortedListOfNeighbors` in its turn, and sent a JOIN message which requests it to add the current agent to its group. If it denies the request, by sending a NOT_JOINED message, `find_group_leader` is called again by AGP, and continues the search to the next agent in the list. When the agent
encountered on the top of sortedListOfNeighbors is the searching agent itself, the
search after a group leader is terminated, and the agent assigns itself as its own group
leader (set_self_as_group_leader).

set_self_as_group_leader marks the current agent as its own group leader, and
sends a JOINED message to each agent waiting for its response in waitingToJoinList.
This message informs each such agent that the current agent has indeed become its
group leader. waitingToJoinList contains all the agents that had asked the current
agent whether it is ready to become their group leader. Once the role of the current
agent had been determined, it can inform them that it accepts their request. In
addition, it informs all other agents that it had determined its group leader to be self
by sending a SET_MY_GROUP_LEADER message.

When a JOIN message is received, AGP calls the join method. join checks whether
the current agent is a group leader. If so it sends a JOINED message that informs
the agent that it is in its group now, and the agent is added to the group list. Otherwise, if
the current agent is not a group leader, it sends a NOT_JOINED message that informs
the requesting agent to continue its search with some other agent. A third possibility is
that the agent hadn't determined yet if it is a group leader or not (still waiting for a
response from some other agent). In such a case the requesting agent is added to the
waitingToJoinList, and is sent a response in a later stage, when the current agent had
determined if it is, or is not, a group leader.

If a JOINED message is received, AGP calls the joined method to handle this
situation. joined marks current agent as not being a group leader, and records the
agent that is. Next, it informs all agents waiting for a response in waitingToJoinList
that it cannot accept their request, by sending them a NOT_JOINED message. It then
informs all agents that it had determined its group leader

Correctness Proof Summary. A number of claims can be proven by analyzing the
algorithmic structure of the group partitioning phase.
1. After a finite number of steps, each agent ai satisfies either isGroupLeader=true or
   isNotGroupLeader=true. This claim follows from the fact that the size of the
   waitingToJoinList is finite for all agents at any given point of time, since agents
   send join requests only to higher priority agents.
2. There exists no agent ai, for which both isGroupLeader=true and
   isNotGroupLeader=true.
3. For each agent ai, it holds that (isGroupLeader=true and isNotGroupLeader=false)
   or (isGroupLeader=false and isNotGroupLeader=true). This claim follows directly
   from claims 1 and 2.
4. It follows that each agent ai will execute one method out of the
   set_self_as_group_leader and joined methods.
5. Exactly one method out of the methods join or set_self_as_group_leader is
   executed exactly once, for any agent ai.
6. Each agent ai has exactly one group leader.
7. Each agent ai belongs to exactly one group. This claim can be shown by first
   proving that each agent belongs to at least one group based on claim 6, and that
   each agent belongs to at most one group by looking on, from the moment of time
   in which ai had been first added to some group.
8. Every group leader belongs to a single group. This follows directly from 7 and from set_self_as_group_leader.
9. Every agent \( a_i \) belongs to the same group as its group leader.
10. To each group belong agents, having the same group leader.
11. The priority of a group is the priority of its group leader. If two or more groups were calculated to have the same priority, their order is resolved lexicographically (see 7 and 9).
12. The group partitioning phase terminates after a finite number of steps (1, 5 and 6 above).

**Example.** Let us demonstrate the group partitioning phase by examining its execution on the instance of figure 2 (a), for agent \( a_1 \). All the agents start their execution in parallel, and we will demonstrate the run on some random execution order.

\( a_1 \) enters the find_group_leader method and finds \( a_3 \) at the top of its sortedListofNeighbors, as it has the highest priority out of all its neighbors. \( a_1 \) then sends it a JOIN message. \( a_3 \) receives the message and calls the join method. There \( a_1 \) is added to waitingToJoinList of \( a_3 \). Now \( a_3 \) starts its execution and calls find_group_leader. It finds \( a_6 \) on top of its sortedListofNeighbors, so it sends \( a_6 \) a JOIN message. \( a_6 \) receives the message and calls the join method. Since its state has not been determined yet, it adds \( a_3 \) to its waitingToJoinList. Similarly, \( a_6 \) sends a JOIN message to \( a_5 \), and is added to its waitingToJoinList. Now \( a_5 \) starts its execution. It's on the top of its sortedListofNeighbors, so it calls set_self_as_group_leader. There it sets itself as a group leader, sends a JOINED message to \( a_6 \) and a SET_MY_GROUP_LEADER to all the agents. \( a_6 \) receives the JOINED message from \( a_5 \), marks it as its group leader, and sends a NOT_JOINED message to \( a_3 \). \( a_3 \) receives the NOT_JOINED, and calls find_group_leader again. Since \( a_1 \) is now on the top of its sortedListofNeighbors, it calls set_self_as_group_leader. There it sets itself as a group leader, and sends a JOINED message back to \( a_1 \), that finally marks \( a_3 \) as its group leader. The results of the partitioning for all agents are shown in figure 2 (b).

3.3 Group Ordering Phase

The group ordering phase orders the agents within each group, in accordance with the Inner Group Priority (IGP), which is different from the priority calculated for the agent in the priority calculation phase. IGP is calculated by using a simple heuristic, which determines the IGP based on the size of groups which are connected to the current agent by some constraint. This heuristic uses the following formula:

\[
IGP_{a_i} = PG_{a_i} + \sum PCG_{ij}
\]

where \( IGPA_i \) is the inner group priority of agent \( a_i \), \( PGA_i \) denotes the priority of the group of agent \( a_i \), and \( PCG_{ij} \) is the priority of the \( j^{th} \) connected group to one of its agents, \( a_i \) is connected. If the agent \( a_i \) is connected to a number of agents in the same group, that group will appear accordingly a number of times in the summation. In this way, an agent which is connected to a greater number of bigger groups will have a
higher IGP.

To calculate its IGP, each agent \( a_i \) adds to the priority of its group leader, the sum of the priorities of the group leaders of all the groups to which \( a_i \) is connected, and then notifies all the agents in its group of its IGP. Each agent then sorts the agents belonging to its group by their IGPs, resolving equal priorities lexicographically, thus defining the search order within the group.

### 3.4 AGP's GroupPartition Correctness

We have thus completed the presentation of AGP's implementation of the GroupPartition algorithm, and are now ready to refer to its correctness, in the following claim:

- The priority computation, group partitioning, and group ordering phases of AGP implement correctly the GroupPartition algorithm, and preserve all the properties required by APF. The proof of this claim is based on verifying that the GroupPartition algorithm is implemented correctly by these phases, as had been partially shown by the correctness proof summary in section 3.2, and that they meet all APF's requirements for the GroupPartition algorithm.

### 3.5 Search for Solution Phase

AGP's search for solution phase is directly based on the implementation of APF's search, as depicted in figure 1. AGP replaces APF's LocalSearch algorithm with the Synchronous Backtracking (SBT) [10] algorithm, and GlobalSearch algorithm with the Conflict based BackJumping (CBJ) [8] algorithm. It is easy to see that both SBT and CBJ meet the requirements of APF.

We have thus completed the presentation of the AGP algorithm. The correctness of AGP follows from the correctness of APF shown in section 2.3, the correctness of AGP's implementation of the GroupPartition algorithm, shown in section 3.4, and from the correctness of its search for solution phase shown in this section.

### 4 The AGP-CBJ Algorithm

As mentioned before, APF is a general framework for agent cooperation and coordination for solving DCSPs. This property is emphasized by the fact that both the search inside each group and the search between the groups can be implemented by any algorithm preserving certain properties. To demonstrate this idea we have implemented the Asynchronous Group Partitioning Conflict Based BackJumping (AGP-CBJ) algorithm, producing our second implementation of APF.

AGP-CBJ is a natural evolution of the AGP algorithm. It is identical to the AGP algorithm, besides the fact that inside each group, instead of using for the LocalSearch algorithm an SBT algorithm, a CBJ algorithm is used instead. Thus, in AGP-CBJ both the local search and the global use the CBJ algorithm. The correctness of AGP-CBJ
follows from the correctness of AGP, as well as from the fact that CBJ meets all APF’s requirements for the LocalSearch algorithm.

5 Experimental Evaluation

To evaluate the performance of AGP, a number of experiments had been conducted, which we describe below.

5.1 Experimental Setup

All experiments were conducted on an asynchronous simulator. In this simulator, agents are simulated by threads, which can only communicate by sending messages to each other. In each experiment, the network of the constraints had been randomly generated by selecting the probability \( p_1 \) of a constraint among any pair of variables, and the probability \( p_2 \), of a violation of a constraint, by assignments of values to a pair of constrained variables. Uniform random networks of constraints are specified by the number of variables in the network - \( n \), the number of values in the domain of each variable – \( k \), the constraints density parameter – \( p_1 \), and the tightness parameter - \( p_2 \), and are commonly used in experimental evaluation of DCSP algorithms [1,4,6,7].

Three sets of problems had been generated with the parameters \( (n=10, k=10) \), \( (n=15, k=10) \) and \( (n=20, k=10) \). For each set the values of \( p_1 \) and \( p_2 \) were varied between 0.1 and 0.9. For each combination of the parameters \( (n, k, p_1, p_2) \), 10 different instances of problems had been generated. Thus, for each combination of \( (n, k) \), 810 different problems were generated, or 2430 problems in total. Our results are presented for the larger instances of 20 variables.

To evaluate the performance of distributed algorithms, two independent measures of performance are commonly used [1,4,6,7] - run time in the form of Non-Concurrent Constraint Checks (NCCCs) [5], and communication load in the form of total number of messages sent.

In all graphs which compare 2 algorithms the number of non-concurrent constraint checks had been averaged for all problems with the given number of \( p_1 \), and all values of \( p_2 \). That is, every point in the graphs represents the average run results of two algorithms over 90 problems.

5.2 Impact of the LocalSearch Algorithm

In figure 4 we present the comparison of the computational effort and the network load of AGP and AGP-CBJ. It is apparent that AGP-CBJ outperforms AGP, both by the computational effort performed and by network load, by an order of magnitude, for all values of \( p_1 \). What differentiates AGP-CBJ from AGP is the search technique used inside each group. These results indicate that the most significant computational effort is performed in APF algorithms, during the search inside each group. Therefore, the search technique used inside the groups will probably have a crucial impact on the performance of the algorithm.
5.3 Partitioning Characteristics

An interesting question which arises upon examining AGPs’ behavior concerns the group partitioning strategy. One might wonder whether the partitioning of agents into groups is in fact a helpful strategy which reduces the computational effort required to solve a DCSP problem. To address this question, a number of experiments had been conducted. The experiments summarized in figure 5 had been conducted for all values of \( p_1 \) and \( p_2 \) between 0.1 and 0.9. That is, each graph summarizes the partitioning of 810 problems into groups.

The first set of experiments tries to find a correlation between the number of groups generated for each problem and the computational effort exhibited by the algorithms for that problem. Figure 5 (a) plots the average number of NCCCs performed by the AGP-CBJ algorithm against the number of groups that problems were partitioned into. Figure 5 (a) shows a strong correlation between the number of the groups, problems were partitioned into, and the computational effort needed to solve these problems. As the number of the groups a problem was partitioned into increases, the computational effort performed by the AGP-CBJ algorithm decreases.

An exception to this relation are problems partitioned into a single group. To better understand this exception we need to reflect on the nature of such problems. From Claim 10 in the correctness proof summary of section 3.2 it follows that if all agents were partitioned into the same group, they are all connected to their group leader. Statistically, there is probably more than one agent which is connected to all other agents in its group. These problems are dense, representing cliques or almost cliques, which are most commonly insoluble, and due to the high density, a no solution can be found quite quickly on them. This explains the low computational effort needed to solve single-group problems.

Similar results had been observed both for the AGP algorithm, and for drilldowns performed for specific values of \( p_1 \).

The next question to be investigated is, for what kind of problems the partitioning strategy will be most efficient. To answer this question, figure 5 (b) presents for each value of \( p_1 \) the average number of groups, problems having that value of \( p_1 \) were partitioned to. There is a clear correlation between the number of the groups...
generated, and the problem density parameter, p. As the density of a problem increases, the number of the groups generated decreases. Since the algorithms were found to be most efficient for problems partitioned into greater number of groups, we will expect the AGP and AGP-CBJ algorithms to be more efficient for problems with lower values of p.

### 5.4 Comparison to APO & ABT

For the comparison of AGP-CBJ and APO algorithms, one needs to use the correct APO version - CompAPO [1]. It is also the best performing version of APO [1]. Figures 6 (a)-(b) present the comparison of the AGP-CBJ and CompAPO algorithms. It is apparent that AGP-CBJ outperforms CompAPO with respect to the computational effort performed. However, CompAPO sends a smaller number of messages than AGP-CBJ, due to its centralized nature.

Figures 6 (c)-(d) present the comparison of the AGP-CBJ and ABT algorithms. We can see that AGP-CBJ generally outperforms ABT with respect to the computational effort. This is true for instances with both low constraint density (p ≤ 0.4) and with the highest constraint density (p ≥ 0.8). However, AGP-CBJ is outperformed by ABT for instances of problems with density 0.5 ≤ p ≤ 0.7. Regarding the network load, AGP-CBJ outperforms ABT, for all p, except for 0.5 ≤ p ≤ 0.6.

### 5.6 Experimental Evaluation Summary

The importance of the search technique used inside each group was found to be great. AGP-CBJ outperformed AGP by an order of magnitude with respect to all measures.

The benefit of the group partitioning strategy was found experimentally to be positive. As the number of groups a problem was partitioned into increases, the computational effort drops. Problems that were partitioned into a larger number of groups needed a lower computational effort to find a solution. A direct correlation was found between the number of groups a problem was partitioned into and the problem density parameter p. As the density of a problem increases, the number of
generated groups decreases. Thus, the AGP and AGP-CBJ algorithms are expected to be more efficient for problems with lower values of $p_1$. Further experiments indicated that AGP-CBJ performs well for values of $p_1$ that are less than 0.5. Surprisingly, AGP-CBJ is efficient as well for values of $p_1$ which are greater than 0.8. The static group partitioning strategy of AGP-CBJ was compared to the dynamic group partitioning strategy of APO. The static strategy of AGP-CBJ was found to be superior by an order of magnitude than the dynamic strategy of APO. This applies to computational effort. However, the APO algorithm exchanges much less messages than AGP-CBJ.

The experimental evaluation included also a comparison of the proposed AGP-CBJ algorithm to the fully asynchronous ABT algorithm. AGP-CBJ outperforms ABT for most values of problem density $p_1$, except for the range $0.5 \leq p_1 \leq 0.7$.

6 Discussion

A new framework, the Asynchronous Partitioning Framework (APF), for agent cooperation and coordination in solving DCSPs was presented. APF is driven by the structure of the underlying constraints graph of a given DCSP, focusing the computational efforts on the more difficult regions of the problem first. APF exploits
the benefits of the fail first principle by partitioning the problem into groups that follow the structure of the underlying constraints graph.

APF allows the replacement of most of its components - the GroupPartition, the LocalSearch and the GlobalSearch algorithms - by any other algorithm or strategy which preserve certain properties. The required properties for each component were defined and the correctness of APF proven.

To demonstrate the generality of APF, two distinct algorithms which implement it were presented – the AGP and the AGP-CBJ algorithms. The framework was experimentally evaluated by conducting a series of experiments on the AGP and AGP-CBJ algorithms. The most significant result is that the group partitioning strategy of APF generated a strongly improved performance.

The presented framework, as well as its implementations, the AGP and AGP-CBJ algorithms, can be improved in a number of ways. First, the LocalSearch and GlobalSearch can be replaced by more advanced DCSP algorithms, such as AFC [6]. Second, some of APF's requirements can be relaxed, to allow a greater degree of versatility to the implementing components. Third, the heuristics of the GroupPartition algorithm can be upgraded. Since the number of groups a problem is partitioned into was found to have a great impact on the performance of APF, a more sophisticated group partitioning strategy can limit the number of agents in a group or split larger groups into smaller ones.

References