

Distributed Constraint Programming with Agents

Carl Christian Rolf and Krzysztof Kuchcinski

Department of Computer Science, Lund University, Sweden
Carl_Christian.Rolf@cs.lth.se, Krzysztof.Kuchcinski@cs.lth.se

Abstract. Many combinatorial optimization problems lend themselves to be modeled as distributed constraint optimization problems (DisCOP). Problems such as job shop scheduling have an intuitive matching between agents and machines. In distributed constraint problems, agents control variables and are connected via constraints. We have equipped these agents with a full constraint solver. This makes it possible to use global constraint and advanced search schemes. By empowering the agents with their own solver, we overcome the low performance that often haunts distributed constraint satisfaction problems (DisCSP). By using global constraints, we achieve far greater pruning than traditional DisCSP models. Hence, we dramatically reduce communication between agents. Our experiments show that both global constraints and advanced search schemes are necessary to optimize job shop schedules using DisCSP.

1 Introduction

In this paper, we discuss distributed constraint programming with agents (DCP). We introduce advanced agents with global constraints, and advanced search to solve distributed constraint satisfaction problems (DisCSP), in particular distributed constraint optimization problems (DisCOP). DCP is a special form of constraint programming (CP), where variables belong to specific agents and can only be modified by their respective agents.

We differentiate DCP from DisCSP since DisCSP has traditionally assumed one variable per agent [18]. In contrast, we study the case where agents can control several variables, making it possible to use global constraints. Such constraints are often needed when solving complex CP problems, such as job shop scheduling problems (JSSP).

Using global constraints in DCP requires that each agent has its own constraint solver. Having a full solver in each agent also makes modeling and communication more efficient. As far as we know, no published work on DisCSP has studied global constraints. In earlier work, these constraints were transformed into equivalent primitive or table constraints. This led to inefficient solving and model representation.

There are two main contributions in this paper:

- We empower the individual agents with a full constraint solver; and
- We introduce an advanced search scheme.

A major advantage of each agent having a full solver is that we can create advanced search methods by adding constraints during search. In this paper, we study JSSP and

search that adds ordering-constraints before the actual assignments, significantly increasing the performance. We are not aware of any previous research on DisCSP that studies this type of search.

Formally, a constraint satisfaction problem (CSP) can be defined as a 3-tuple $P = (X, D, C)$, where X is a set of variables, D is a set of finite domains where D_i is the domain of x_i , and C is a set of primitive or global constraints containing several of the variables in X . Solving a CSP means finding assignments to X such that the value of $x_i \in D_i$, while all the constraints are satisfied. P is referred to as a constraint store.

Finding a valid assignment to a constraint satisfaction problem is usually accomplished by combining backtracking search with consistency checking that prunes inconsistent values. In every node of the search tree, a variable is assigned one of the values from its domain. Due to time-complexity issues, the consistency methods are rarely complete. Hence, the domains will contain values that are locally consistent, i.e., they will not be part of a solution, but we cannot prove this yet.

DisCSP, as used in [17], can be defined similarly to CSP with the 4-tuple $P = (A, X, D, C)$, where A is a set of agents and X is a set of variables so that $x_i \in a_i$. D is a set of finite domains, and C is a set of sets of *binary* constraints. Each variable x_i has a finite domain d_i , and each set of constraints c_{ij} connects two agents a_i and a_j , where $i \neq j$. Furthermore, each variable is controlled by exactly one agent. Lastly, the constraint network builds a connected graph. In other words, each agent is connected to another agent. Hence, there is at least one path from agent a_i to agent a_j .

Our model of DCP extends the DisCSP definition to a higher level. We retain the properties that a variable is controlled by exactly one agent, and that there is a path from any agent a_i to agent a_j . Now, however, X is a set of sets of variables, C is a set of sets of n -ary constraints, and D is a set of sets of finite domains. Every agent a_i has a set of variables x_i and a set of constraints c_i . In [13], a similar definition is introduced, but not expanded upon. In fact, we are not aware of anyone actually using the main advantage of having many variables per agent. The fact that our agents can have global constraints enables us to use the full power of modeling and pruning in CP.

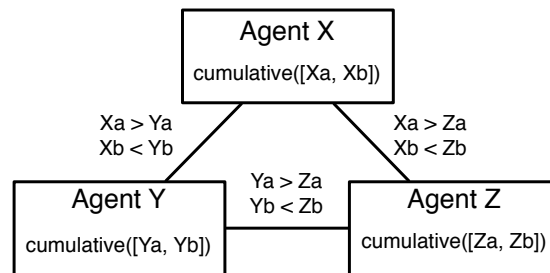


Fig. 1: Model of a distributed JSSP, where each agent holds several variables.

In DCP, we can perform the consistency and search phase asynchronously [20]. First, we let each agent establish consistency internally, then send its prunings to the agents that are connected via constraints. Figure 1 depicts the structure of the constraint

network for a small JSSP. Each agent holds two variables and ensures no overlap between tasks via a cumulative constraint [2]. The constraints between agents are the precedence constraints, stated at the edges. Whenever a variable that is part of a connected constraint changes, the prunings will be propagated to the connected agents. Using our formal model, we, e.g., have $A = \{X, Y, Z\}$, $x_x = \{Xa, Xb\}$, $c_x = \{cumulative([Xa, Xb]), Xa > Ya, Xb < Yb\}$ and $c_{xy} = \{Xa > Ya, Xb < Yb\}$.

The rest of this paper is organized as follows. Section 2 introduces the background and the related work. In Section 3, our model of DCP with global constraints and advanced search is described. Section 4 describes our experiments and results. Finally, Section 5 gathers our conclusions.

2 Background and Related Work

Most work on DisCSP deals with the scenario where each agent holds a single variable and only binary constraints exist between the agents [18]. These problems are typically solved with an asynchronous search, where consistency is enforced between the connected agents [20]. One notable exception is [13]. However, that paper mentions neither global constraints nor advanced search methods.

The model of each agent only controlling one variable and only having binary constraints can technically be used to model any problem. However, even the latest search algorithms need to send a huge amount of messages to other agents [6] to solve such problems. This makes such a limited model less feasible when dealing with large or complex problems. This is especially problematic for optimization problems, since there is a greater need for search than for simple satisfiability problems.

One main difference between our model and previous work, such as [20, 6, 4, 17, 13], is that we can communicate entire domains. When a domain has been received, the prunings it carries are evaluated. This is much more efficient than sending one value from a domain at a time and getting a *Good* or *NoGood* message back.

Privacy is often used to motivate distribution of variables. Previous work, such as [21] shows that perfect privacy is possible for DisCSP. However, in the real world, complex encryption and minimal communication are impractical if they decrease performance too much. Our ultimate goal is to use our work for scheduling in autonomous unmanned aerial vehicles [9]. Hence, we focus more on performance than privacy.

A great limitation of previous work is that the problem model is usually translated into a table form [11]. These tables represent all possible assignments by the cartesian product of the domains in the constraint. For many problems, this representation is unfeasibly large [17]. In scheduling, a single cumulative constraint, ensuring no overlap of tasks [2], would have to be translated to binary constraints for every single time point. Even small scheduling problems would need thousands of constraints.

Many complex optimization problems need global constraints to solve in reasonable time. Some papers on DisCSP build advanced structures of agents. Others add a master-like agent that controls the global limits of the problem [12]. However, as far as we know, no one provides global constraints in each agent.

In order for DCP to solve large problems which are relevant to the real world, like JSSP, we need more advanced agents. Theoretically, one variable per agent is sufficient

to model any DisCSP. However, just as global constraints can be reduced to binary constraints, the decreased pruning makes such an approach unrealistic for large optimization problems. This paper introduces agents with full constraint solvers, in order to make DCP feasible for industry use.

3 Distributed Solvers

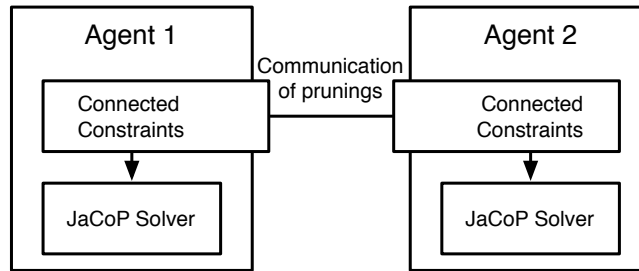


Fig. 2: Our model of DCP, each agent holds a full constraint solver.

Figure 2 depicts our model of DCP. Each agent holds a separate copy of the JaCoP solver [8], and only controls the variables that are needed for that part of the problem model. For instance, in JSSP, each agent holds the variables representing the tasks on the machine that the agent models. The precedence constraints between tasks assigned to different machines are stored in the connected constraints, since they constrain tasks controlled by different agents. This is how prunings are sent between agents.

Figure 3 depicts a simplified view of the distributed constraint evaluation process and the search. All time consuming steps in our solving are parallel. As depicted in Fig. 3, the algorithm evaluates consistency and the agents vote on the next master in parallel. However, in order to guarantee synchronicity, the agents must wait for all prunings to be finished before they can move on to select the next master. Hence, the algorithm moves from synchronous to asynchronous execution of the agents, and back again, with every assignment.

When consistency is evaluated, all prunings are sent directly between the agents that are part of the connected constraint. Hence, the master agent is not controlling communication. It serves only to make an assignment decision and ensure that all agents are synchronized for the next step in the execution. The next step after an assignment may be to backtrack, or locate the next master, or to communicate a solution.

An example of the operations of our model is depicted in Fig. 4, which shows all execution steps. The execution progresses as follows.

1. When the solving is initialized, all agents start to run consistency of their constraints, see Fig. 4(a).
2. If there are changes to a variable that is in a connected constraint, those prunings are sent to the agent holding the other variable of the connected constraint, see

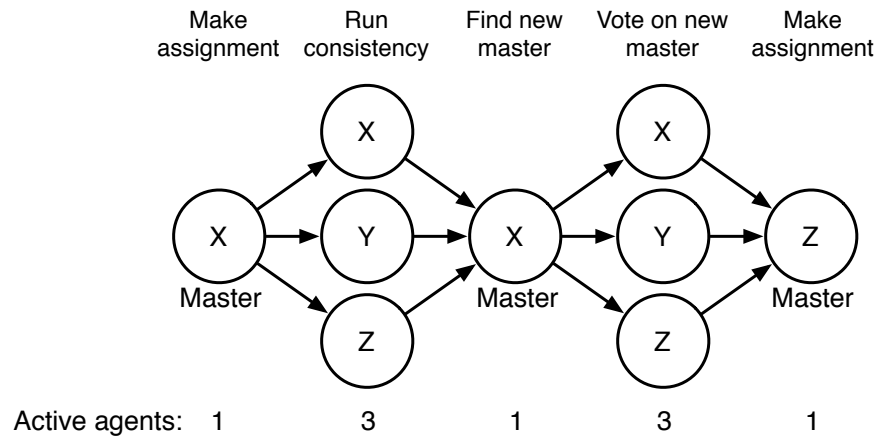


Fig. 3: The progress from assignment to next assignment. X, Y, and Z are agents.

Fig. 4(b). In this paper, we only study the case of binary constraints between agents. Each agent holds a queue of pruning messages, when changes have been received, consistency is again evaluated in the agent. This process continues iteratively until there are no more prunings sent between agents.

3. As soon as the consistency is finished, a negotiation determines which agent will start the search, see Fig. 4(c). This follows the principles of distributed election [5].
4. The agent holding the variable with the highest priority, defined by a user configurable heuristic, gets the master token, see Fig. 4(d)-(e). In this paper, we look at synchronized search. This means that only one agent holds the master token and only this master gets to make the next assignment decision.
5. The master makes an assignment and enforces consistency, see Fig. 4(f).
6. The master sends the prunings to the agents that have connected constraints containing changed variables, see Fig. 4(g).
7. When the agents receive prunings, they automatically run consistency, see Fig. 4(h).
8. When consistency has finished again, we are at the same position as in Fig. 4(c). We renegotiate which agent is to be the new master.

The procedure above continues until all variables have been assigned a value. When a master finds a solution, the cost of the solution can be shared amongst all agents by propagating it to all agents connected to the master. These agents then propagate it further, and so on, until all agents are aware of the solution cost. This is similar to the communication in [3]. Sharing solution costs is necessary in order to use branch and bound search.

If backtracking is necessary, we will undo the assignment leading to the inconsistency. If the current master has run out of possible assignments, it will send a message to the previous master telling it to backtrack. Hence, all agents that have been masters keep track of which agent was master before itself. Furthermore, since agents have several variables, an agent can become master several times in the same search tree branch. Agents therefore also need to keep track of backtracking to themselves.

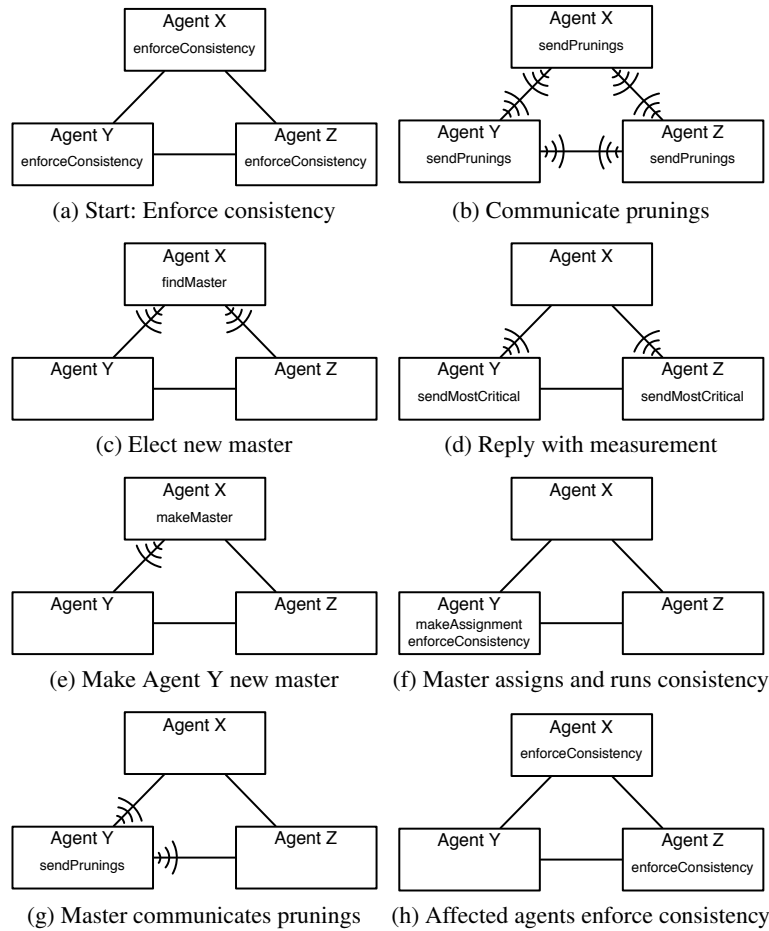


Fig. 4: The operating sequence for consistency and search in our model. The waves along the edges indicate communication

The pseudo-code for our model is shown in Fig. 5. The receive method will be called automatically by the agent whenever a message has been received. Communication between agents are performed by similar syntax to that of [7]. All communication of costs is handled by connected constraints and is therefore controlled by the problem model. This gives great versatility to our model.

The biggest challenge in our distributed model is to detect that all agents are synchronous. For instance, detecting that consistency has reached a fixpoint and it is time to make the next assignment. That detection takes place in the handling of the message `Wait_For_Consistency`. Verifying whether agents are running and consistent can be done as for DisCSP, by using the process of [3].

```

1 // variables controlled by the agent V
2 // actors that participate in the problem A
3
4 receive(message) {
5   switch (message.type) {
6
7     case Make_Master(oldMaster):
8       master = true
9       previousMaster = oldMaster
10      this ! Start_Search
11
12    case Start_Search:
13      v = selectionHeuristic.selectVariable
14      if (v == null)
15        storeSolution
16      this ! Backtrack
17    else
18      k = valueHeuristic.selectValue
19      store.makeAssignment(v, k)
20      this ! Enforce_Consistency
21
22    case Enforce_Consistency:
23      running = true
24      if (store.enforceConsistency)
25        forall (c in connectedConstraints)
26          forall (v in c.remoteVariables)
27            if (v.hasChanged)
28              v.remoteAgent !
29              Pruning(v.name, v.domain)
30    else
31      consistent = false
32      if (master)
33        this ! Wait_For_Consistency
34      running = false
35
36    case Pruning(varName, domain):
37      v = store.findVariable(varName)
38      v.domain = domain
39      this ! Enforce_Consistency

```

(a)

```

1 case Wait_For_Consistency:
2   forall (a in A)
3     if (a.isRunning)
4       this ! Wait_For_Consistency
5     return
6   forall (a in A)
7     if (a.inconsistent)
8       this ! Backtrack
9     return
10  this ! Select_Next_Master
11
12 case Backtrack:
13   store.forbidLastAssignment
14   store.undoLastAssignment
15   if (store.stillInconsistent)
16     previousMaster ! Backtrack
17   else
18     this ! Start_Search
19
20 case Select_Next_Master:
21   forall (a in A)
22     a ! Find_Best_Variable(this)
23
24 case Find_Best_Variable(theMaster):
25   v = selectionHeuristic.selectVariable
26   k = v.fitness
27   theMaster ! Fitness(k, this)
28
29 case Fitness(fitness, actor):
30   if (fitness > bestFitness)
31     bestFitness = fitness
32     bestActor = actor
33   fitnessReplies += 1
34   if (fitnessReplies == A.size)
35     master = false
36     bestActor ! Make_Master(this)
37 }
38 }

```

(b)

Fig. 5: The pseudo-code for the agents. The receive method is called whenever a message arrives. An exclamation mark indicates communication to an agent.

3.1 Advanced Search in Distributed Constraint Programming

In order to solve complex JSSP, we need the more advanced search that is made possible by our model. The algorithm presented in Fig. 5 is somewhat simplified. For JSSPs, we use a sequence of two search methods. The first orders the tasks on each machine by adding precedence constraints. The second assigns actual start times for each task. This is based on the principles described in [1]. While some problems may solve without the ordering, many require an ordering to solve in reasonable time.

Figure 6 depicts the algorithm for the ordering search. During the ordering, the machine with the least slack in the tasks scheduled on it is selected. Then we pick the task, running on that machine, with the smallest start time. Finally, we impose that the selected task has to execute before the other tasks on that machine, and we remove it from the list used to calculate slack. This procedure is repeated recursively.

This type of advanced search is not possible in all DisCSP solvers. Many DisCSP solvers cannot impose constraints during the search. Even solvers that can impose new constraints, are often limited by mostly supporting table constraints [11]. If only table

constraints are supported, the memory use of the solver will increase greatly whenever new constraints are imposed for every time unit of the schedule.

```

1 // M is a vector of vectors representing tasks assigned
2 // to a machine. Each task is specified by its starting
3 // task start time t, task duration d
4
5 boolean Jobshop_Search(M)
6   if store.enforceConsistency
7     if M ≠ ∅
8       m ← selectCriticalMachine(M)
9       sort tasks in m in ascending values of t.min()
10      for each i = 1, ..., n
11        for each j = 1, ..., n
12          if (i ≠ j)
13            impose mi.t + mi.d ≤ mj.t
14            M' ← M \ mi
15            if Jobshop_Search(M')
16              return true
17            else
18              return false
19            return false
20          else
21            store solution
22            return true
23        else
24          return false
25
26 vector selectCriticalMachine(M)
27   for each mi ∈ M
28     min ← min(min(mi.t0), min(mi.t1), ..., min(mi.tn))
29     max ← max(max(mi.t0 + mi.d0), max(mi.t1 + mi.d1), ...,
30             max(mi.tn + mi.dn))
31     supply ← max - min
32     demand ← ∑ mi.di
33     critical ← supply - demand
34   return machine mi with the lowest value of critical

```

Fig. 6: The code for the ordering search.

4 Experimental Evaluation

For our experiments, we used the JaCoP solver [8]. The agent system is written using actors in Scala [14]. The experiments were run on a Mac Pro with two 3.2 GHz quad-core Intel Xeon processors running Mac OS X 10.6.2 with Java 6 and Scala 2.8.1. These two processors have a common cache and memory bus for each of their four cores. The parallel version of our solver is described in detail in [16]. We used a timeout of 30 minutes for all the experiments. All experiments were run 20 times, giving a standard deviation of less than 5%.

We ran several standard benchmark scheduling problems described in [19, 10]. The characteristics of the problems are listed in Table 1. These are all JSSP, where n jobs with m tasks are to be scheduled on m different machines. We study the case of non-preemptive scheduling.

Table 1: Characteristics of the problems for the global constraint model.

Problem	Jobs	Tasks	Variables	Constraints	Optimum
LA01	10	5	61	56	666
LA04	10	5	61	56	590
LA05	10	5	61	56	593
MT06	6	6	43	43	55

We created two DisCOP models of each problem: one for our version of DCP with global constraints, and the other representing the traditional case with only primitive constraints. When using our model, each agent represents one machine. It contains one global cumulative constraint with n tasks [2], to ensure no overlap of tasks.

In the primitive model, each agent represents one variable. For the problems we studied, the primitive constraints are binary in the sense that they only contain two variables. Our primitive constraint models did not use table constraints. Instead, they used the constraint $start_i + duration_i \leq start_j \vee start_j + duration_j \leq start_i$ for every pair of tasks, to ensure no overlap. This constraint is technically a binary constraints, since the duration is a constant. These primitive constraints can replace cumulative for JSSP since we only have one instance of each resource.

Each problem was started with no prior knowledge of the optimal solution. Hence, the domains of the variables representing the start time tasks were $\{0..1000\}$. When using many resources, translating a single cumulative constraint into a table constraint requires primitive constraints in every time point. For many problems, this could result in an excessive number of rows in the table constraint. This is often infeasible due to memory size.

4.1 Experimental Results

The results for finding and proving the the optimal solution are shown in Table 2 and Table 3. Clearly, the primitive representation of the problems rarely found the optimal within the 30 minute timeout. The only exception was MT06, the simplest problem we tested. Still, finding the solution for MT06 took almost 30 times as long as the global model.

Table 2: Execution time in seconds that the global constraint model took to find the optimal solution and the best solution found within the 30 minute timeout.

Problem	Time to find optimum	Time to prove optimum	Solution
LA01, Global	3.8	4.0	666
LA04, Global	10.8	12.1	590
LA05, Global	0.7	0.97	593
MT06, Global	3.0	3.0	55

Table 3: Execution time in seconds that the primitive constraint model took to find the optimal solution and the best solution found within the 30 minute timeout.

Problem	Time to find optimum	Time to prove optimum	Solution
LA01, Primitive	Timeout	Timeout	936
LA04, Primitive	Timeout	Timeout	976
LA05, Primitive	Timeout	Timeout	720
MT06, Primitive	87.7	Timeout	55

Our model of DCP with global constraints in each agent gives superior performance in our experiments. The traditional model with only one variable per agent never managed to prove the optimality within the timeout. This performance increase comes partly from the fact that we can order variables before we start search. When agents control only one variable, this type of ordering is not possible. In this case, adding the ordering constraints will mostly serve to increase the number of pruning messages that need to be sent.

When we turn off the ordering of tasks, the performance drops significantly for the global model. However, even though we could not prove optimality without ordering, we found better solutions within the timeout than the primitive model for almost all problems. Hence, the benefit of our model is not simply in the use of advanced search, but also in the use of global constraints.

Although our search is synchronous, using asynchronous search would probably not benefit the traditional primitive model much. Our model would probably still be better, because in our experiments we use a simulated distribution, thus minimizing the penalty of sending many messages. The primitive model communicates many more messages to reach the consistency fixpoint. When using a network, the communication would be an order of magnitude more time consuming than on a shared-memory multicore machine.

Using asynchronous search would bring benefits to both the global constraint model and the primitive one. However, the search space of CP is exponential with regard to domain size. Parallel search only gives a polynomial speed-up [15]. Hence the performance advantage of the global constraint model is likely to remain, even though the model with one variable per agent allows for more parallelism.

We also created a third model, where each agent control several variables, but have no global constraints. Just as for the global representation, each agent models one machine. However, the cumulative constraint has been replaced by the same kind of constraints as in the primitive model for every pair of tasks.

The performance of this third model, shown in Table 4, was better than that of the single variable per agent model. However, the performance was usually much lower than of the global constraint representation. For the simplest problem it was slightly faster. But for the most difficult problem, it did not find the optimum within the timeout.

The performance benefit of our model of DCP is not simply because of our advanced search. The ordering of tasks on each machine is possible in the model with several variables per agent but without global constraints. However, the pruning is much weaker when there are no global constraints.

Our results for the model in Table 4, compared to the results in Table 3, illustrate the cost of communication. We get much better performance than the scenario of one variable per agent, despite using the same constraints. Hence, the difference between the performance of these two models comes mostly from the communication of prunings.

The cost of communication depends on the agent framework. However, we ran our experiments on a shared-memory machine. Running on a cluster, with network communication, would increase the performance penalty of communication severely. If anything, our experiments over estimate the competitiveness of traditional DisCSP models.

Table 4: Results for multi-variable agents, *without* global constraints, but *with* ordering.

Problem	Time to find optimum	Time to prove optimum	Best Solution
LA01	3.8	6.9	666
LA04	Timeout	Timeout	667
LA05	0.47	9.7	593
MT06	2.5	2.6	55

5 Conclusions

In this paper, we have introduced a completely new model of distributed constraint programming. Unlike any work we are aware of, we equip each agent with a full constraint solver. Our model is the the only one we have seen published that can use global constraints. It also allows advanced search, during which we can order tasks before assigning actual start times of scheduling problems.

By equipping each agent with a full constraint solver, we allow much more efficient modeling of problems. Unlike most work on DisCSP, we do not translate our models into table constraints. This allows us to communicate domains and constraints between agents during the search. Such communication is much more efficient than that of traditional DisCSP. Reducing communication is a major concern in DisCSP solving.

Our main conclusion of this paper is that both global constraints and advanced search are needed in order to solve complex scheduling problems using distributed constraint programming. Traditional work on DisCSP has focused on agents that only control one variable and only have primitive constraints. We conclude that these older models are unlikely to offer good performance for real world use, even when using asynchronous search.

Another conclusion is that using the traditional approach to DisCSP of one variable per agent should be very well motivated. Using one variable per agent may provide better robustness and privacy. However, we show that letting agents control several variables, using global constraints, and using advanced search methods are all important for good performance.

References

1. Baptiste, P., Pape, C.L., Nuijten, W.: Constraint-Based Scheduling. Kluwer Academic Publishers, Norwell, MA, USA (2001)
2. Caseau, Y., Laburthe, F.: Improving branch and bound for jobshop scheduling with constraint propagation. In: *Combinatorics and Computer Science*. pp. 129–149 (1995)
3. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3, 63–75 (1985)
4. Ezzahir, R., Bessiere, C., Belaisaoui, M., Bouyakhf, E.: DisChoco: A platform for distributed constraint programming. In: *Proceedings of the IJCAI'07 Distributed Constraint Reasoning Workshop (DCR'07)*. pp. 16–27 (2007)
5. Gallager, R.G., Humblet, P.A., Spira, P.M.: A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.* 5, 66–77 (1983)
6. Gershman, A., Meisels, A., Zivan, R.: Asynchronous forward bounding for distributed COPs. *Journal of Artificial Intelligence Research* 34, 61–88 (2010)
7. Hoare, C.A.R.: Communicating sequential processes. *Commun. ACM* 21, 666–677 (1978)
8. Kuchcinski, K.: Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 8(3), 355–383 (2003)
9. Kvarnstrom, J., Doherty, P.: Automated planning for collaborative UAV systems. In: *The 11th International Conference on Control Automation Robotics Vision*. pp. 1078–1085 (2010)
10. Lawrence, S.R.: Resource-constrained project scheduling: An experimental investigation of heuristic scheduling techniques. Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh PA (1984)
11. Léauté, T., Ottens, B., Szymanek, R.: FRODO 2.0: An open-source framework for distributed constraint optimization. In: *Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09)*. pp. 160–164 (2009)
12. Meisels, A., Kaplansky, E.: Scheduling agents - distributed timetabling problems (DisTTP). In: *Practice and Theory of Automated Timetabling IV, Lecture Notes in Computer Science*, vol. 2740, pp. 166–177. Springer Berlin / Heidelberg (2003)
13. Meisels, A., Zivan, R.: Asynchronous forward-checking for DisCSPs. *Constraints* 12, 131–150 (2007)
14. Odersky, M., Spoon, L., Venners, B.: *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edn. (2008)
15. Rao, V.N., Kumar, V.: Superlinear speedup in parallel state-space search. In: *Proceedings of the Eighth Conference on Foundations of Software Technology and Theoretical Computer Science*. pp. 161–174. Springer-Verlag, London, UK (1988)
16. Rolf, C.C., Kuchcinski, K.: Load-balancing methods for parallel and distributed constraint solving. *The 10th IEEE International Conference on Cluster Computing* pp. 304–309 (2008)
17. Rossi, F., Beek, P.v., Walsh, T.: *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA (2006)
18. Salido, M.: Distributed CSPs: Why it is assumed a variable per agent? In: Miguel, I., Ruml, W. (eds.) *Abstraction, Reformulation, and Approximation, Lecture Notes in Computer Science*, vol. 4612, pp. 407–408. Springer Berlin / Heidelberg (2007)
19. Thompson, G.L.: *Industrial scheduling / edited by J.F. Muth and G.L. Thompson with the collaboration of P.R. Winters*. Prentice-Hall, Englewood Cliffs, N.J. : (1963)
20. Yokoo, M., Hirayama, K.: Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems* 3(2), 185–207 (2000)
21. Yokoo, M., Suzuki, K., Hirayama, K.: Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In: Van Hentenryck, P. (ed.) *Principles and Practice of Constraint Programming - CP 2002, Lecture Notes in Computer Science*, vol. 2470, pp. 43–66. Springer Berlin / Heidelberg (2006)