Distributed and Dynamic Task Reallocation in Robot Organizations

Wei-Min Shen and **Behnam Salemi**

Information Sciences Institute and Computer Science Department
University of Southern California
4676 Admiralty Way, Marina del Rey, CA 90292

Abstract

Task reallocation in a multi-robot organization is a process that distributes a decomposed global task to individual robots. This process must be distributed and dynamic because it relies on critical information that can only be obtained during mission execution. This paper presents a representation for this challenging problem and proposes an algorithm that allows member robots to trade tasks and responsibilities autonomously. Preliminary results show that such an algorithm can indeed improve the efficiency of organizational performance and construct a locally optimal (hill climbing) task allocation during mission execution.

1. Introduction

In an organization of multi-robots, *task reallocation* is the process of distributing a global mission task, which has been decomposed into subtasks, to the robots in the organization. In contrast to the problem of resource allocation, task reallocation emphasizes the task migration and organizational changes among robots, rather than allocation of resources.

Traditionally, task reallocations are often considered in a centralized and static setting. A single controller robot would gather and examine all the relevant information about the current organization and mission, then decide and allocate tasks for every fellow robot. The weakness of this approach is that the accuracy of this global information is not always possible to obtain and often hard to maintain. The central controller must know the decomposition of the global task and the required capabilities and resources for each subtask. It must also know every robot's available capabilities and resources. This approach also creates a fragile bottleneck in the organization. Any failures to the controller robot will paralyze the entire organization. A distributed task allocation negotiation system is presented in [1], which is based on the contract net protocol. However, this work is different form the proposed approach in this paper, since our approach considers dependencies among subtasks as a criterion for task reallocation.

Many previous approaches for task reallocation also assume a known evaluation function for measuring the quality of task allocation, and this function remains unchanged during the process of problem solving. For example, [2] solves task allocation by analyzing the evaluation function and making all necessary decisions before the problem solving starts. Similarly, [3] assumes that the values of coalitions are computable before the execution of the organization.

In real-world applications, a solution to task allocation must consider the dynamic aspects of the environment and unexpected changes in robot behaviors because a given evaluation function might be inaccurate and the capabilities and resources of robots may change. Robots in an organization must be able to negotiate without any fixed leaders and find a satisficing solution for task allocation.

Task reallocation is closely related to the problem of self-organization, where the main objective is to decide who does what and how to collaborate with others. However, self-organization is a very diverse natural phenomenon, and a coherent and general definition is still in debate. For example, [4] defines an organization as a set of problem solvers with "information and control relationships." [5] describes an organization as a set of production systems with shared variables.[6] describes an organization as a set of "routines". [2] models an organization as a task dependent structure that includes the task units to be done, the participating (universally capable) robots, an assignment of the tasks to the robots, and a workflow structure dictates the task distribution and result assembly. Most of these definitions, although they provide valuable case studies, are not operational for task reallocation during problem solving.

This paper proposes a new approach to the problem of distributed and dynamic task reallocation based on the principles of self-organization. The approach deals with the dynamic changes in the robots and in the environment by reallocating tasks based on the performance of robots. Such reallocations are made by individual robots themselves and require no central controller robot to know the global knowledge of the organization and task progress. To focus our attention on the organizational aspects of the problem, we simplify the measurement of performance by considering the costs of communication only. Our approach is similar to the bottom-up approaches for self-organization. [5]. However, we do not assume that robots have universal capabilities (i.e., every robot can handle every subtask) and the population of robots can be changed arbitrarily.

In this paper, we define a result of task reallocation as assignment of robots to a *role-graph,* and define the process of task reallocation as the optimization of these assignments, with respects to the dynamic cost function. In a role-graph, a *role* node is a set of responsibilities (or subtasks) for the given task, and a *role-relationship* edge is a commitment between roles to communicate certain types of information (such as subtasks, solutions, actions, or data). This representation separates the role requirement from the robots' capabilities, and makes the assignment of robots to roles an essential task in organization. In this paper, we assume that an initial role-graph is given to the robots, but the robots are allowed to modify the role-graph at will. As we will see later, these modifications include trading tasks and responsibilities among robots, and such modifications can affect the structure of an organization. With this representation, task reallocation is a team-learning process for adapting a role-graph and searching an optimal robot assignment to the role-graph based on performance results during problem solving.

The rest of the paper is organized as follows. Section 2 gives a formal definition for the problem of distributed and dynamic task reallocation. Section 3 presents a solution approach based on local search with two novel heuristics for task-trading and responsibly-trading among robots. Section 4 describes the SOLO algorithm that implements the above approach. Section 5 presents the experimental results of the SOLO algorithm. Section 7 concludes the paper with future research directions.

2. Distributed and Dynamic Task Reallocation

To study the problem of distributed and dynamic task reallocation in a domain-independent fashion, it is necessary to ground the research on a rigorous computational foundation that is domain-independent and decomposable among multiple robots. Examples of such foundations include Distributed Constraint Satisfaction Problems (DCSP), Distributed Bayesian Networks, Contract Nets, and Graphical Models [7]. In this paper, we shall focus on DCSP to investigate the feasibility of the approach.

A Constraint Satisfaction Problem (CSP) is commonly defined as assigning values to a list of variables V from a respective list of domains D such that a set of constraints C over the variables is satisfied. For example, we can define an example CSP$_1$ as follows: V=[x_1, x_2, x_3], D=[{1,2}, {2}, {1,2}], and C=[($x_1 \neq x_3$), ($x_2 \neq x_3$)]. Then a solution for CSP$_1$ is (x_1,x_2,x_3)=(2, 2, 1). A distributed CSP is a CSP in which V, D, and C are distributed among multiple robots. A DCSP is solved if each robot solves its local portion of the CSP and the collection of all local solutions is a solution to the CSP. For instance, we can partition the above example into two parts: V_1=[x_1,x_2], D_1=[{1,2},{2}], C_1=[($x_1 \neq x_3$)] and V_2=[x_3], D_2=[{1,2}], C_2=[($x_2 \neq x_3$)], and assign them to two robots respectively. Then, a solution to the DCSP is (x_1,x_2)=(2,2), and (x_3)=(1). In the standard DCSP, however, task reallocation is not an issue because the assignment between robots and variables are given and static.

To generalize DCSP to address the problem of distributed and dynamic task reallocation, we map tasks to variables, task dependencies to constraints between variables. A task is solved if the corresponding variable is assigned a value that does not violate any involved constraints. We further introduce that (1) every task/variable has a set of required capabilities, (2) every task dependency/constraint has two responsibilities: the *supervisor* and the *subordinator*; and (3) there is a set of heterogeneous robots that collectively possess all the required capabilities. For task dependency that links two task variables, the responsibility of the supervisor is to select a value for its variable and pass the value to the subordinator. The responsibility of the subordinator is to adjust the value of its variable so that it satisfies the constraint with the supervisor's value.

Formally, the problem of distributed and dynamic task reallocation can be defined as a tuple (V, R, D, C, A), where V is a list of task/variables, R a list of required capabilities by the task/variables, D a list of value domains for the task/variables, C a set constraints, and A a set of robots with heterogeneous capabilities. The goal of this problem is to find an assignment $A \Leftrightarrow (V,C)$ that is both *complete* and *optimal*. An assignment is complete if every task and every responsibility is assigned to a *qualified* robot and no single capability is assigned to more than one task simultaneously. A robot is qualified for a task if the robot possesses the necessary capabilities required by the task. An assignment is optimal if it enables a solution to the given global problem to be found with the minimal cost. The cost of a global solution can be measured in a user-specified way. For example, it could be the total number of messages sent between robots, or the sum of computational time consumed by the participating robots. In this paper, however, we will only consider the total number of messages for communication.

To illustrate the above definitions, consider a task reallocation problem TR$_1$ extended from CSP$_1$ as follows: V=[x_1, x_2, x_3], R=[{c_1, c_4}, {c_2}, {c_3}], D=[{1,2}, {2}, {1,2}], C=[($x_1 \neq x_3$), ($x_2 \neq x_3$)], and A=[A_1={c_1, c_2, c_4}, A_2={c_2, c_3, c_4}]. In this problem, the required capabilities for task x_1 are {c_1, c_4}, task x_2 {c_2}, task x_3 {c_3}, respectively. The robot A_1 has capabilities {c_1,c_2, c_4} and is qualified for x_1 and x_2, and the robot A_2 has {c_2,c_3,c_4} and is qualified for x_2 and x_3. Without loss of generality, we can simplify the problem by assuming that each task requires a unique capability so that the capability requirements for tasks can be embedded in the task variables. For each task that requires more than one capability, such as x_1 requires {c_1, c_4}, we create a new capability called $c_{1\&4}$ so that x_1 can be handled by $c_{1\&4}$. With this convention, TR$_1$ can be simplified as: V=[x_1, x_2, x_3], R=[{$c_{1\&4}$}, {c_2}, {c_3}], D=[{1,2}, {2}, {1,2}], C=[($x_1 \neq x_3$), ($x_2 \neq x_3$)], and A=[A_1={$c_{1\&4}$, c_2}, A_2={c_2, c_3, c_4}]. Thus we can replace robots' capabilities by task variables as follows: R=[{x_1},{x_2},{x_3}] and A=[A_1={x_1, x_2}, A_2={x_2, x_3}].

With the above definition, we can enumerate all possible task allocations for a given problem by matching the qualification of robots with task variables and task dependency constraints. In our current example, there are eight possible task allocations listed in Table 1. For example, in the solution O_1, $A_1:(x_1,x_2)$ indicates that the robot A_1 is assigned tasks x_1 and x_2, and $A_2:(x_3)$ indicates that the robot A_2 is assigned x_3. The responsibility assignment $[x_1 \rightarrow x_3]$ means that for the task dependency between x_1 and x_3, the robot A_1 (who is assigned to x_1) is the supervisor while A_2 (who is assigned to x_3) is the subordinator. Similarly, for the task dependency between x_2 and x_3, $[x_3 \rightarrow x_2]$ indicates that A_2 is the supervisor and A_1 is the subordinator.

Table 1: All possible task allocations for TR_1

	Task Assignment	Responsibility Assignment
O_1	$A_1:(x_1,x_2)$, $A_2:(x_3)$	$[x_1 \rightarrow x_3 \rightarrow x_2]$
O_2	$A_1:(x_1,x_2),A_2:(x_3)$	$[x_1 \rightarrow x_3 \leftarrow x_2]$
O_3	$A_1:(x_1,x_2),A_2:(x_3)$	$[x_1 \leftarrow x_3 \rightarrow x_2]$
O_4	$A_1:(x_1,x_2),A_2:(x_3)$	$[x_1 \leftarrow x_3 \leftarrow x_2]$
O_5	$A_1:(x_1),A_2:(x_2,x_3)$	$[x_1 \rightarrow x_3 \rightarrow x_2]$
O_6	$A_1:(x_1),A_2:(x_2,x_3)$	$[x_1 \rightarrow x_3 \leftarrow x_2]$
O_7	$A_1:(x_1),A_2:(x_2,x_3)$	$[x_1 \leftarrow x_3 \rightarrow x_2]$
O_8	$A_1:(x_1),A_2:(x_2,x_3)$	$[x_1 \leftarrow x_3 \leftarrow x_2]$

The above task allocations can be graphically represented in a Distributed Organizational Task Network (DOTN). In a DOTN, the nodes are the task variables with required capabilities, the edges are the task dependency constraints, and the direction of an edge represents the responsibilities of the involved robots in the corresponding task dependency constraint. A DOTN is complete and optimal if the robot assignment to the elements in the DOTN (nodes and edge directions) is complete and optimal. To illustrate the representation of DOTN, Figure 1(a), 1(b), and 1(c) shows three task allocations corresponding to O_1, O_2, and O_8, respectively. Note that O_1 and O_2 have the same task assignments $[A_1:(x_1,x_2),A_2:(x_3)]$, but different responsibility assignment for the dependency constrain $[x_2 \neq x_3]$. In O_1, A_1 is the subordinator and A_2 the supervisor, while in O_2, A_1 is the supervisor and A_2 the subordinator. In Figure 1(c), O_8 has a totally different task assignment: A_1 is assigned to x_1, and A_2 to (x_2, x_3).

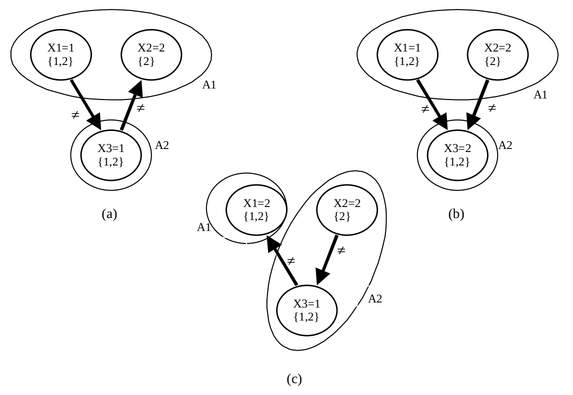

Figure 1: DOTN examples for O_1, O_2, and O_8.

Intuitively speaking, the solution O8 is the optimal task allocation because it is most likely for the robots to find a coherent solution for the global problem faced by the organization. This is because the tasks x_1 and x_2 are free from any dependency constraints so they are best to be assigned to different robots. Furthermore, for the only dependency constraint $[x_2 \neq x_3]$ across the two robots, x_2 has a more restricted domain than x_3, so its assignee A_2 should be given the responsibility of supervisor. In this simple illustrative example, the objective of task reallocation is to find this optimal solution O_8 during the process of solving the global DCS problem.

3. Task Reallocation by Heuristic Search

The above definition of task allocation implies that solving the problem requires a search in an enormous space for possible assignments between robots and DOTN elements (nodes and edge directions). This search can be done either exhaustively or heuristically.

The basic idea for exhaustive search is quite simple. One can find the best task allocation by going through all possible task allocations and returning the one that has the best performance. This search is complete because it guarantees to find the optimal task allocation. But the time complexity of this search makes it impractical to use. When the environment is changed, the entire search process must be started over again.

To find a practical solution for distributed and dynamic task reallocation, we may use local search approaches to find an approximation of the best task allocation. Local search proceeds by incrementally improving the current task allocation based on the feedback of solving the global problem. To do so, we have to answer two questions: how to modify a task allocation, and when to apply these modifications so that they result in improvement. By definition, a task allocation can be modified by two actions: trade tasks among robots, and trade responsibilities among robots. We now discuss them in detail.

Trading Tasks among Robots

In a task allocation process, tasks with dependency constraints are partitioned into groups, and each group is then assigned to a qualified robot. In this context, we can classify a dependency constraint between two tasks as either *remote* (across robots) or *local* (within a robot). An optimal task allocation is the one that minimizes the remote dependencies between robots so that each robot can solve its own tasks in a relatively independent way.

How do we measure the dependencies between two tasks or two robots? The most straightforward way is to simply count the number of dependency relationships between tasks or robots. This is a static estimation and the best task allocation based on this measurement can be accomplished by analyzing the role-graph at the outset and partition the roles into groups to minimize the total number of dependency relationships between groups. However, this static approach does not consider the likelihood how a dependency can be satisfied. Such likelihood would depend on how easy to find solutions for the tasks involved, how wide the communication bandwidth is between robots, and many other facts. Such information is not available until the problem solving process starts.

As a first attempt for dynamic task reallocation, we estimate the dependency between two tasks A and B by the number of messages exchanged between the tasks:

dependency(A,B) ≅ the_num_of_messages(A,B).

When a task *x* is traded from robot A to robot B, some local dependencies of x may become remote, while some remote dependencies of x may become local. The purpose of such task trading is to reduce the total amount of remote dependencies among all robots.

To illustrate this point, consider the task allocation in Figure 2, where five tasks t_1, t_2, t_3, t_4, and t_5 are allocated to three robots A, B, C, and the dependencies on the links are as shown. Assume that based on the qualification of the robots, only t_2 and t_3 can migrate from A to B or C.

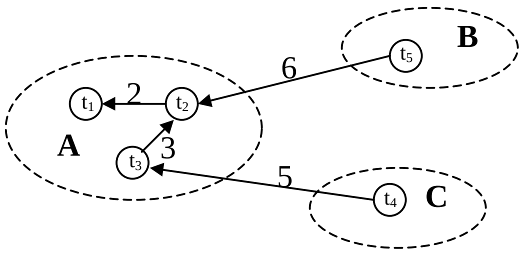

Figure 2: Trading tasks in a task allocation.

Given the above facts, the changes of remote dependencies caused by the four possible task trades can be computed as follows:

Trade t_2 from A to B: -6+3+2 = -1
Trade t_2 from A to C: 3+2 = 5
Trade t_3 from A to B: +3
Trade t_3 from A to C: -5 +3 = -2

For example, when t_2 is moved from A to B, it eliminates a remote dependency of value 6, but introduces two new remote dependencies of value 2 and 3. Among these four possibilities, the trade t3 from A to C is the most profitable modification for the current task allocation.

In general, we trade a task *x* from a robot A to another robot B if the total remote dependencies between x and tasks in B is higher than the total local dependencies of x in A. This trade of task will reduce the total remote dependencies in the entire task allocation. To facilitate this decision-making, each robot is required to record the number of messages sent and received on each task dependency.

Trading Responsibilities among Robots

Trading responsibilities between robots is another way to modify a task allocation. The basic idea is to switch the supervisor and subordinator responsibility whenever the subordinator cannot find any possible solution to its task. The motivation of this action is that by becoming a supervisor, a robot may have more freedom to choose solutions for its task, therefore more likely to find a global solution.

In DOTN, this action switches the direction of an edge and alters the direction of information flow between roles. Such changes can affect the performance of an organization because it has been demonstrated in [8] that changes in the priority order among variables can influence the rate of problem solving in DCSP. Switching responsibilities is a special case of changing priorities.

The simple protocol of switching responsibilities between two robots, however, is too limited to be effective and can run into deadlocks. Consider a situation where both the supervisor and the subordinator cannot find solutions to their tasks, then no matter how many times they switch responsibilities; the problem can never be solved because they are constrained by other neighbors.

To overcome this problem, we extend the scope of trading responsibilities from two robots to the entire neighborhood. Inspired by the priority schema in [9], we assume that every task is assigned a global priority. If two tasks are linked by a dependency constraint, then the task with the higher priority is the supervisor. Whenever a robot fails to find a solution for its task, it will switch the priority of that task with a neighboring task that has the highest priority in the neighborhood. Different from the schema in [9], this protocol does not introduce any new priorities for the tasks yet it can avoid any loop creation in DOTN.

To illustrate this protocol of trading responsibilities, assume that the tasks [t_1,t_2,t_3,t_4,t_5] in Figure 2 have the priorities [1,2,3,4,5]. When t_2 fails to find a solution, it will switch its priority with t_5 because t_5 is the neighbor task that has the highest priority. Notice that after this switch, t_2 becomes the supervisor of all its neighbors, which includes t_5, t_3, and t_1.

4. The SOLO Algorithm

The process of task reallocation described above has been implemented as a new algorithm called SOLO, which is an extension of the Asynchronous Backtracking Algorithm [10] and the version that deals with multiple variables per agent [9]. Given a DCSP, the SOLO algorithm allows robots to reallocate the task variables assigned to them initially. The output of SOLO is a solution to the given DCSP and a near optimal task allocation among robots.

Four types of messages are used. (1) An *ok?* message is sent when a robot is proposing a new solution to its local task. When a robot *Ai* sends out an *ok?* message for a local task x_i, it also indicates if it is willing to give away (*giveAway?*) the task to the receiver. (2) A *nogood* message is sent when a subordinator robot finds it is impossible to find a solution for a task to satisfy all neighboring supervisors. (3) An *interested* message is sent by a qualified robot as a reply to an *ok?* message to indicate the willingness to accept the offered task. (4) A *release* message is sent by a robot to transfer a local task to an interested receiver. Among all the “interested”

```
when received (ok?, (Aj, xj, dj, priority, giveAway?)) do
  add (Aj, xj, dj, priority) to robotView;
  if (giveAway? & qualifedfor(xj) & profitToAccept(xj)) then
      send(interested, Ai, Aj, xj);
  when robotView and currentAssignments are inconsistent
      checkRobotView;
end do;

when received (nogood ,xj ,nogood) do
  add nogood to the nogoodList
  when (xk, dk, priority) is contained in the nogood
      where xk is not in the neighbors do
      add xk to neighbors,
      add (Ak, xk, dk, prority) to robotView; end do;
  checkRobotView;
end do;

when received (interested, Aj, xi) do
  if xi is in the possessedVariablesList then
      delete xi from possessedVariablesList ;
      send (release, (xi, di, priority)); end if
end do;

when received (release, (xi, di, priority)) do
  add xi to possessedVariablesList;
  annouce the new ownership of xi to the neighbors;
end do;

procedure checkRobotView
  if robotView and currentAssignments are consistent then
      for each xi that has a new value d do
          for each robot Ak that has a constraint with xi do
          giveAway?=CommCost(xi,Ak)>localCommCost(xi));
          send (ok?, (Ai, xi, d, currentPriority(xi), giveAway?));
  else select $x_k$ from possessedVariablesList, which has the
          highest priority and violating some constraint
          with higher priority variables;
          if no value in $D_k$ is consistent with
              robotView and currentAssignments then
              record and communicate a nogood, i.e., the subset
                  of robotView and currentAssignments where
                  $x_k$ has no consistent value;
              when the obtained  nogood is new do
                  switch the priorities between $x_k$ and the
inconsistent variable that has the highest priority in the nogood;
                  $x_k = d;$ where $d \in D_k$ and $d$ minimizes the
number of violations with lower priority variables;
                  checkRobotView; end do;
          else $x_k = d;$ where $d \in D_k$ and $d$ is consistent with
              robotView and currentAssignments and minimizes
              the number of violations with lower priority
              variables;
              checkRobotView;
end if; end if;
```

Figure 3: The SOLO Algorithm

receivers, the offering robot will select the one that has the highest remote dependency with respect to the offered task. Since communication among robots are asynchronous, the offering robot cannot guarantee to wait for all interested parties so it is permissible to release the task to the first interested receiver.

Figure 3 illustrates the procedures for receiving messages such as *ok?*, *nogood*, *interested,* and *release*, and for checking local robot views. The algorithm starts with an initial task allocation determined randomly. Each robot assigns values to its local variables, and sends *ok?* messages to all related subordinator robots. After that, robots wait and respond to incoming messages. When an *ok?* message about a variable *x* is received, the receiver robot *A* will update its local view and send back an *interested* message if it is qualified to possess *x* and also possessing *x* is profitable for *A*. When an *interested* message for a variable *x* is received, the receiver robot will relinquish the variable (by deleting *x* from its local variable list) and replies with a *release* message. The receiver of the *release* message will add the variable to it local variables list and announce the new ownership by a set of *ok?* messages.

To illustrate the SOLO algorithm in detail, let us consider our TR1 example again. We assume that the initial task allocation is Figure 1(a). Each robot communicates these initial values via *ok?* messages. When A_1 sends an *ok?* message to A_2 for x_1's new value, it also indicates the willingness to give x_1 away because x_1 has a higher remote dependency than its local dependency. When A_2 receives this *ok?* message, it updates the *robotView* but is not interested in x_1 because the lack of qualification. After A_2 assigns a new value 2 to its local variable x_3, it sends an *ok?* message to A_1 (for A_1 is the subordinator of the constraint $x_3 \neq x_2$). This time A_1 fails to find a consistent value x_2 to satisfy the constraint of $x_3 \neq x_2$, so it performs the following actions. A_1 sends a *nogood* message $\{(x_3=2)\}$ to A_2, switches the priority value of x_2 with x_3, selects a new value 2 for x_2, sends an *ok?* message to A_2 to inform the priority switch and its willingness to give x_2 away. At this point, the task allocation becomes Figure 1(b). In this new task allocation, A_2 sends out two messages: an *interested* message to A_1 for taking x_2, and a *nogood* $\{(x_1=1),(x_2=2)\}$ message to A_1 because it fails to find a consistent value for x_3. After these messages, A_1 releases x_2 to A_2 and changes the value of x_1 to 2. At this point, all tasks are solved and the task allocation is Figure 1(c).

5. Experimental Results

We have applied the SOLO algorithm to a distributed 3-color problem. Given *n* variables, we first generate a random 3-color problem with 2.7*n* links (to ensure the difficulty of the problems). We then generate a set of *m* robots by randomly partitioning the capabilities (variables) into *m* even subsets. If the *n*/*m* is not an integer, then the remaining capabilities are assigned to the last robot. To make sure that robots have overlapping capabilities, we then expend each robot's capabilities by adding extra *p%*, randomly selected different capabilities. Notice that when *p*=0, every robot has unique capabilities and there is no room for trading tasks. If p=100, then all robots can trade all tasks.

Table 2 lists the results of running SOLO with different number of variables (*n*), robots (*m*), and capability overlapping (*p*). Each data point in the table is the average for 50 randomly generated problem instances. The initial values of the variables in these trails are determined randomly. To show the effects of task trading, we have recorded the number remote and local messages, the cycles needed to solve the problem, and the number of task trading.

Table 2: The effects of task trading

n/*m*	*p*=0	*p*=30	*p*=60	*p*=90
	# of remote messages			
10/4	119.0	79.3	68.1	62.0
10/8	265.1	186.7	137.4	200.8
20/8	1004.7	2394.5	1111.7	593.2
20/12	5729.7	3652.9	3038.3	1132.6
30/10	2584.0	2269.4	2490.3	2343.2
30/20	5969.4	7007.3	4236.4	348.9
50/10	2948.9	3152.1	3446.1	3310.4
100/20	6041.7	5920.6	5766.5	5718.1
n/*m*	*# of local messages*			
10/4	29.8	19.1	17.6	21.6
10/8	17.8	25.3	19.6	35.4
20/8	142.2	333.3	189.6	89.6
20/12	412.4	379.8	409.2	143.9
30/10	269.4	239.6	264.7	257.1
30/20	176.7	329.6	293.6	19.4
50/10	324.5	353.0	396.4	395.4
100/20	311.5	329.1	333.3	350.3
n/*m*	*# of cycles for solving DCSP*			
10/4	29.8	19.1	17.6	21.6
10/8	5.8	5.8	4.1	8.2
20/8	20.8	44.6	33.5	16.5
20/12	47.0	47.0	60.2	21.2
30/10	26.7	27.4	31.8	32.1
30/20	21.3	32.4	37.6	3.7
50/10	34.0	39.7	39.8	41.9
100/20	22.8	24.7	26.1	27.0
n/*m*	*# of task trading*			
10/4	0.0	0.7	1.1	0.5
10/8	0.0	0.0	2.5	2.0
20/8	0.0	0.1	2.8	3.9
20/12	0.0	3.7	5.0	4.9
30/10	0.0	1.3	3.6	5.8
30/20	0.0	7.0	8.0	6.6
50/10	0.0	7.0	8.7	7.4
100/20	0.0	7.2	9.9	10.8

As we can see from the results, as the overlapping capability increases, more tasks are traded between robots, less communication is needed between robots, and the rate of converge is faster (less cycles). In general, when robots have choices for what they do, task reallocation allows them to solve the problem much more quickly than fixed task allocation. We notice that communication does not reduce monotonically with the capability overlapping. In the case m/n=20/8, we see an increase of communication at 30% of overlapping, before it goes down again. Further investigation is required to determine the causes of this phenomenon.

6. Conclusion

This paper presents an approach to distributed and dynamic task reallocation in multi-robot systems. This problem is motivated by the fact that most critical information for organizational performance must be obtained during problem solving and static criteria for task allocation cannot take the dynamic information into account. The paper identifies two important heuristics for improving task reallocation and presents an initial implementation of the SOLO algorithm.

The research reported here also suggests a number of future research directions in task reallocation. In particular, more factors other than the number of messages must be considered to better estimate the dependencies between tasks and robots. Applications of the approach to real-world problems that involve robots are also necessary. An even more challenging problem is to deal with the changes in the environment where solutions to tasks are non-stationary.

7. References

1. Sandholm, T.W. *Contract Types for Satisficing Task Allocation: I Theoretical Results*. in *AAAI Spring Symposium Series*. 1998.
2. So, Y.-P., E.H. Durfee. *An organizational self-design model for organizational change*. in *National Conference on Artificial Intelligence*. 1993.
3. Shehory, O., S. Kraus. *Formation of overlapping coalitions for precedence-ordered task-execution among autonomous agents*. in *International Conference on Multiple Agent Systems*. 1996.
4. Durfee, E.H., V.R. Lesser, D.D. Corkill, *Trends in Cooperative Distributed Problem Solving.* IEEE Transactions on Knowledge and Data Engineering, 1989. **1**(1): p. 63-83.
5. Ishida, T., L. Gasser, M. Yokoo, *Organization Self-Design of Distributed Production Systems.* IEEE Transactions on Knowledge and Data Engineering, 1992. **4**(2): p. 123-134.
6. Levitt, B., J.G. March, *Organizational Learning.* Annual Review of Sociology, 1988. **14**: p. 319-340.
7. Jordan, M., *Learning in Graphical Medels*. 1998: MIT Press.
8. Armstrong, A., E. Durfee. *Dynamic prioritization of complex agents in distributed constraint satisfaction problems*. in *Proceedings of the International Joint Conference on Artificial Intelligence*. 1997.
9. Yokoo, M., E.H. Durfee, T. Ishida, K. Kuwabara, *The Distributed Constraint Satisfaction Problem: Formalization and Algorithms.* IEEE Transactions on Knowledge and Data Engineering, 1998. **10**(5): p. 673-685.
10. Yokoo, M., K. Hirayama. *Distributed Constraint Satisfaction Algorithm for Complex Local Problems*. in *International Conference for Multiple Agent Systems*. 1998.