A Distributed Merge-and-Split Mechanism for Dynamic Virtual Organization Formation in Grids

Lena Mashayekhy
Department of Computer Science
Wayne State University
Detroit, MI 48202, USA
Email: mlena@wayne.edu

Daniel Grosu
Department of Computer Science
Wayne State University
Detroit, MI 48202, USA
Email: dgrosu@wayne.edu

***Abstract*—We model the Virtual Organization (VO) formation problem in grids using concepts from coalitional game theory and design a distributed mechanism for solving it. The proposed distributed mechanism enables the formation of VOs guaranteeing the maximum profit for their participating Grid Service Providers (GSPs). We show that the proposed mechanism produces stable VOs, that is, the GSPs do not have incentives to break away from the current VO and join some other VO. We perform extensive simulation experiments using real workload traces to characterize the properties of the proposed distributed mechanism. The results show that the proposed distributed mechanism not only produces VOs that are stable yielding high revenue for the participating GSPs, but also decides the structure of the VOs in a reasonable amount of time.**

***Keywords*-grid computing; VO formation; coalitional games;**

I. Introduction

Grid computing systems are composed of heterogeneous resources (CPUs, storage, etc.) owned by autonomous organizations. These systems provide essential resources for conducting cutting-edge science and engineering research. The resource management in such open distributed environments is a very complex problem. One important aspect of resource management in grids is how Grid Service Providers (GSPs) pool their resources together to execute large scale applications. These GSPs collaborate and form Virtual Organizations (VOs) [1].

The life-cycle of a VO consists of four phases: (i) identification, in which the possible partners and the VO's objectives are identified; (ii) formation, in which the potential partners (GSPs) negotiate the exact terms, the goal, and the duration of collaboration; (iii) operation, in which the members of the VO collaborate in solving a specific task; and (iv) dissolution, in which the VO is dismantled. This paper focuses on designing mechanisms for the second phase, the formation of VOs. We model the VO formation as a coalitional game where GSPs decide to form VOs in such a way that each GSP maximizes its own profit, the difference between revenue and costs. A GSP will choose to participate in a VO if its profit is not negative. The VOs provide the composite resource needed to execute applications. A VO is traditionally conceived for the sharing of resources, but it can also represent a business model [1]. In this work, a VO is a coalition of GSPs which desire to maximize their individual profits and are largely indifferent about the global welfare. We design a distributed VO formation mechanism based on concepts from coalitional game theory [2]. The model that we consider consists of a set of GSPs and a grid user that submits a program and a specification consisting of a deadline and payment. A subset of GSPs will form a VO to execute the program before its deadline. The objective of each GSP is to form a VO that maximizes its own profit.

Related Work. Several economic-based models and systems for resource management in open distributed systems have been proposed in the literature [3], [4]. They do not explicitly address the problem of VO formation and management which is one of the key issues that need to be solved in large scale computing systems in order to facilitate collaboration among participating organizations. Globus Toolkit [5] supports the operation and management of VOs by providing basic middleware for VO policy specification and enforcement, resource management, provisioning, and discovery. The toolkit does not provide mechanisms for VO formation and tools for VO management and analysis. The VO formation problem can be viewed as a coalition formation problem. Research on coalition formation has been conducted in the multi-agent systems community for problems such as allocating a task [6] and service composition [7]. Sandholm et al. [8] proposed a method for finding the optimal coalition structure where the optimal coalition structure is defined in terms of maximizing the social welfare. Their method does not take into account the incentives of the players to form coalitions. In addition, they did not consider the task assignment and the payoff division when finding the optimal coalition structure.

To the best of our knowledge, the closest work to ours is by Patel et al. [9] which describes the mechanisms for VO formation used in the CONOISE-G project. The mechanisms used in CONOISE-G are based on Constraint Satisfaction Programming (CSP) techniques [10] while our proposed mechanism is based on coalitional game models and techniques. CSP techniques do not facilitate the stability and robustness analysis of the VO formation process, while this is intrinsic and represents one of the strengths of coalitional game theory. Furthermore, the CONOISE-G project does not address the issues of scheduling the applications within the VO, while our proposed framework addresses this issue and provides a mechanism for application scheduling within VOs. Carroll and

Grosu [11], developed a VO formation framework based on extensive form games. The approach we use here is based on coalitional games and merge-and-split operations which, unlike the one used in [11], guarantees the stability of the VOs formed by the proposed mechanism and is more computationally efficient. In our previous work [12] we proposed a centralized mechanism for VO formation in grids. The mechanism, called MSVOF, is executed by a trusted party that also facilitates the communication among VOs/GSPs. The distributed mechanism we propose in this paper addresses the drawbacks (single point of failure, scalability, etc.) of the MSVOF mechanism.

Our Contribution. We address the problem of VO formation in grids by designing a distributed mechanism that allows the GSPs to make their own decisions to participate in VOs. In this mechanism, instead of having a centralized party deciding the formation of VOs, the GSPs themselves decide to form VOs. The mechanism produces a stable VO structure, that is, none of the GSPs has incentives to merge to another VO or split from a VO to form another VO. The mechanism determines the mapping of the tasks to each of the VOs that minimizes the cost of execution by using a branch-and-bound method. We analyze the properties of our proposed VO formation mechanism and perform extensive simulation experiments using real workload traces from the Parallel Workloads Archive [13]. The results show that the proposed distributed mechanism determines not only a stable VO that maximizes the individual payoffs of the participating GSPs, but also determines the VOs faster than MSVOF, a centralized mechanism proposed in [12].

Organization. The rest of the paper is organized as follows. In Section II, we describe the VO formation problem and the game theoretic framework used to design the proposed VO formation mechanism. In Section III, we present the proposed distributed mechanism, and characterize its properties. In Section IV, we evaluate the mechanism by extensive simulation experiments. In Section V, we summarize our results and present possible directions for future research.

II. Virtual Organization Formation Framework

In this section, we model the VO formation in grids as a coalitional game and introduce the coalition formation framework used in the design of the proposed mechanism.

A. VO Formation as a Coalitional Game

The system model assumes that a user wants to execute a large-scale application program $\mathcal{T}$ consisting of n independent tasks $\{T_1, T_2, \ldots, T_n\}$ on the available set of grid service providers (GSPs) by a given deadline d. Application programs consisting of several independent tasks are representative for a wide range of problems in science and engineering [14]. Each task $T \in \mathcal{T}$ composing the application program is characterized by its workload $w(T)$, which can be defined as the amount of floating-point operations required to execute the task. Executing the application program $\mathcal{T}$ requires a large number of resources which cannot be provided by a single GSP. Thus, several GSPs pool their resources together to execute the application. We consider that a set of m GSPs, $\mathcal{G} = \{G_1, G_2, \ldots, G_m\}$, are available and are willing to provide resources for executing programs. Here, we assume that the GSPs are driven by incentives in the sense that they will execute a task only if they make some profit out of it. More specifically, the GSPs are assumed to be self-interested and welfare-maximizing entities. Each service provider $G \in \mathcal{G}$ owns several computational resources which are abstracted as a single machine with speed $s(G)$. The speed $s(G)$ gives the number of floating-point operations per second that can be executed by GSP G. Therefore, the execution time of task T at GSP G is given by the *execution time function* $t : \mathcal{T} \times \mathcal{G} \rightarrow \mathbb{R}^+$, where $t(T, G) = \frac{w(T)}{s(G)}$. We also assume that once a task is assigned to a GSP, the task is neither preempted nor migrated.

A GSP incurs cost for executing a task. The cost incurred by GSP $G \in \mathcal{G}$ when executing task $T \in \mathcal{T}$ is given by $c(T, G)$, where $c : \mathcal{T} \times \mathcal{G} \rightarrow \mathbb{R}^+$ is the *cost function*. Furthermore we assume that a GSP has zero fixed costs and its variable costs are given by the function c. A user is willing to pay a price P less than her available budget B if the program is executed to completion by deadline d. If the program execution exceeds d, the user is not willing to pay any amount that is, $P = 0$. Since a single GSP does not have the required resources for executing a program, GSPs form VOs in order to have the necessary resources to execute the program and more importantly, to maximize their profits. The profit is simply defined as the difference between the payment received by a GSP and its execution costs. If the profit is negative (*i.e.*, a loss), the GSP will choose not to participate.

We model the VO formation problem as a coalitional game. A *coalitional game* [2] is defined by the pair $(\mathcal{G}, v)$, where $\mathcal{G}$ is the set of players and v is a real-valued function called the *characteristic function*, defined on $S \subseteq \mathcal{G}$ such that $v : S \rightarrow \mathbb{R}^+$ and $v(\emptyset) = 0$. In our model, the players are the GSPs that form VOs which are coalitions of GSPs. In this work, we use the terms VO and coalition interchangeably. Each subset $S \subseteq \mathcal{G}$ is a *coalition*. If all the players form a coalition, it is called the *grand coalition*. A coalition has a *value* given by the characteristic function $v(S)$ representing the profit obtained when the members of a coalition work as a group. For each coalition of GSPs $S \subseteq \mathcal{G}$, there exists a mapping $\pi_S : \mathcal{T} \rightarrow S$, which assigns task $T \in \mathcal{T}$ to service provider $G \in S$. To maximize the profit obtained by a VO, we need to find an optimal mapping of all the tasks on the members of VO in such a way that the mapping minimizes the execution cost. We call this task assignment problem the MIN-COST-ASSIGN problem.

MIN-COST-ASSIGN finds a mapping of n tasks to k GSPs in VO S, where $k = |S|$. The goal is to minimize the cost of execution. We consider the following decision variables:

$$\sigma_S(T, G) = \begin{cases} 1 & \text{if } \pi_S(T) = G, \\ 0 & \text{if } \pi_S(T) \neq G. \end{cases} \tag{1}$$

We formulate the MIN-COST-ASSIGN problem as an integer program (IP) as follows:

$$\text{Minimize } C(\mathcal{T}, S) = \sum_{T \in \mathcal{T}} \sum_{G \in S} \sigma_S(T,G) c(T,G), \tag{2}$$

$$\text{Subject to: } \sum_{T \in \mathcal{T}} \sigma_S(T,G) t(T,G) \leq d, \ (\forall G \in S), \tag{3}$$

$$\sum_{G \in S} \sigma_S(T,G) = 1, \ (\forall T \in \mathcal{T}), \tag{4}$$

$$\sum_{T \in \mathcal{T}} \sigma_S(T,G) \geq 1, \ (\forall G \in S), \tag{5}$$

$$\sigma_S(T,G) \in \{0,1\}, \ (\forall G \in S \text{ and } \forall T \in \mathcal{T}). \tag{6}$$

The objective function (2) represents the costs incurred for executing the program $\mathcal{T}$ on S under the mapping. Constraints (3) ensure that each GSP can execute its assigned tasks by the deadline. Constraints (4) guarantee that each task $T \in \mathcal{T}$ is assigned to exactly one GSP. Constraints (5) ensure that each GSP $G \in S$ is assigned at least one task. Constraints (6) represent the integrality requirements for the decision variables.

We define the following characteristic function for our proposed VO formation game:

$$v(S) = \begin{cases} 0 & \text{if } |S| = 0 \text{ or IP is not feasible,} \\ P - C(\mathcal{T}, S) & \text{if } |S| > 0 \text{ and IP is feasible,} \end{cases} \tag{7}$$

where $|S|$ is the cardinality of S. Note that $v(S)$ can be negative if $C(\mathcal{T}, S) > P$. That means GSPs in S incur cost. The objective of each GSP is to determine the membership in a coalition that gives the highest share of profit. Here, we adopt the equal sharing of the profit as the payoff division rule. That is, each GSP in coalition S receives the share $v(S)/|S|$.

Due to their welfare-maximizing behavior, the GSPs prefer to form a low profit coalition if their profit divisions are higher than those obtained by participating in a high profit coalition. Therefore, a service provider G determines its preferred coalition S, where $G \in S$ by solving:

$$\max_{(S)} \frac{P - C(\mathcal{T}, S)}{|S|} \tag{8}$$

The GSPs' goal is to maximize their own profit by solving the optimization problem given in equation (8). Therefore, minimizing the cost $C(\mathcal{T}, S)$ by solving the MIN-COST-ASSIGN problem maximizes the profit $P - C(\mathcal{T}, S)$ earned by a VO. A VO obtains profit and then the profit is divided among participating GSPs. As a result, a GSP prefers a VO that provides the highest profit among all possible VOs.

The *payoff* or the *share* of GSP G part of coalition S, denoted by $x_G(S)$ is given by $x_G(S) = \frac{v(S)}{|S|}$. Thus, the payoff vector $(x_{G_1}(S), \cdots, x_{G_r}(S))$ gives the payoff divisions of the r GSPs that are part of coalition S. In this paper, we assume that if a GSP does not execute a task it receives a payoff of 0. If there are some GSPs that do not participate in executing any task of the program, they are not considered members of the VO executing the application.

B. Coalition Formation Framework

Coalition formation [15] is the partitioning of the players into disjoint sets. A coalition structure $\mathcal{CS} = \{S_1, S_2, \ldots, S_h\}$ forms a partition of the set of GSPs $\mathcal{G}$ such that each player is a member of exactly one coalition, *i.e.*, $S_i \cap S_j = \emptyset$ for all i and j, where $i \neq j$ and $\bigcup_{S_i \in \mathcal{CS}} S_i = \mathcal{G}$. The problem of finding the optimal coalition structure is NP-complete [8]. Enumerating all coalition structures to find the optimal coalition structure is not feasible since the possible number of coalition structures is B_m, the m-th Bell number [16] which gives the number of partitions of a set of size m, where $m = |\mathcal{G}|$.

In the VO formation game defined in the previous section only one of the coalitions in the coalition structure is selected to execute the application program. The selected coalition is the one that yields the highest individual payoff for all of its members. The coalitions that cannot complete the program within the deadline will not be considered since the payoff for such coalitions is zero.

The following concepts are used in the design of the VO formation mechanism.

Definition 1 (Collection): A *collection* in $\mathcal{G}$, is defined as the set $\mathcal{C} = \{S_1, \cdots, S_k\}$ of mutually disjoint coalitions. If $\cup_{j=1}^{k} S_j = \mathcal{G}$, the collection $\mathcal{C}$ is called a *partition* of $\mathcal{G}$.

Definition 2 (Comparison): A *collection comparison* $\triangleright$ is defined to compare two collections A and B that are partitions of the same subset $S \subseteq \mathcal{G}$. $A \triangleright B$ implies that the way A partitions S is preferred to the way B partitions S.

In the proposed VO formation game, a welfare-maximizing GSP will determine its coalition by considering the profit it earns and not the coalition value. Thus, comparison relations among collections are defined based on the GSPs' individual payoffs. These comparison relations correspond to the merge and split rules which will be defined later in this section. We consider two collections $\hat{S} = \{\cup_{j=1}^{k} S_j\}$ and $\{S_1, \cdots, S_k\}$ from the same subset. We define two comparison relations, the *merge comparison* $\triangleright_m$ and the *split comparison* $\triangleright_s$, based on the individual payoffs as follows:

$$\begin{aligned} \hat{S} \triangleright_m \{S_1, \cdots, S_k\} \quad & \Longleftrightarrow \{\forall j \in \{1, \ldots, k\}, \forall G_i \in \hat{S} \cap S_j; \\ & x_i(\hat{S}) \geq x_i(S_j) \text{ and } \exists j \in \{1, \ldots, k\}, \\ & \exists G_r \in S_j \, ; x_r(\hat{S}) > x_r(S_j)\} \end{aligned} \tag{9}$$

$$\begin{aligned} \{S_1, \cdots, S_k\} \triangleright_s \hat{S} \quad & \Longleftrightarrow \{\exists j \in \{1, \ldots, k\}, \forall G_i \in \hat{S} \cap S_j; \\ & x_i(S_j) \geq x_i(\hat{S}) \text{ and} \\ & \exists G_r \in S_j \, ; x_r(S_j) > x_r(\hat{S})\} \end{aligned} \tag{10}$$

Equation (9) implies that collection $\hat{S}$ composed of only one coalition $\{\cup_{j=1}^{k} S_j\}$ is preferred over $\{S_1, \cdots, S_k\}$, if at least one player G_r is able to improve its payoff without decreasing other players' payoffs. Equation (10) implies that collection $\{S_1, \cdots, S_k\}$ is preferred over $\hat{S}$, if at least one coalition S_j is able to keep the payoffs of its members while at least one of its members, G_r, is able to improve its payoff regardless of the effect on the other players outside of S_j.

Using the defined comparison relations, we propose a VO formation mechanism involving two types of rules as follows [15]:

Merge Rule: Merge any set of coalitions $\{S_1, \cdots, S_k\}$, where $\{\cup_{j=1}^{k} S_j\} \triangleright_m \{S_1, \cdots, S_k\}$.

Split Rule: Split any coalition $\hat{S} = \{\cup_{j=1}^{k} S_j\}$, where $\{S_1, \cdots, S_k\} \triangleright_s \{\cup_{j=1}^{k} S_j\}$.

Coalitions decide to merge only if at least one GSP is able to strictly improve its individual payoff through the merge rule without decreasing the other GSPs' payoffs. Therefore, the merge rule is an agreement among the GSPs to operate together if it is beneficial for them.

As we mentioned before, one of the formed coalitions, the final coalition, executes the program, thus, the formation of the rest of the coalitions is not important. The reason for this is that the rest of the GSPs which are not in the final coalition can participate again in another coalition formation process for executing another application program. Therefore, a coalition decides to split only if there is at least one sub-coalition that strictly improves the individual payoffs of its constituent GSPs. Under the split rule, the individual payoffs of the other sub-coalitions may decrease. The split rule can be seen as the implementation of a *selfish* decision by a coalition, which does not take into account the effect of the split on the other coalitions.

Two coalitions S_i and S_j decide to merge based on the merge comparison defined by (9) where all of GSPs in $S_i \cup S_j$ are able to keep or improve their individual payoffs. A GSP individual payoff is computed based on (8) while satisfying the deadline constraint. As a result, the merge occurs if the following two inequalities are satisfied where at least one of them must be strict.

$$\frac{P - C(\mathcal{T}, S_i \cup S_j)}{|S_i \cup S_j|} \geq \frac{P - C(\mathcal{T}, S_i)}{|S_i|} \tag{11}$$

$$\frac{P - C(\mathcal{T}, S_i \cup S_j)}{|S_i \cup S_j|} \geq \frac{P - C(\mathcal{T}, S_j)}{|S_j|} \tag{12}$$

Since $|S_i \cup S_j| > |S_i|$ and $|S_i \cup S_j| > |S_j|$, in order for a GSP in S_i to keep or improve its payoff, $P - C(\mathcal{T}, S_i \cup S_j) \geq P - C(\mathcal{T}, S_i)$, and it should be the same for a GSP in S_j. Thus, $C(\mathcal{T}, S_i \cup S_j) \leq C(\mathcal{T}, S_i)$ and $C(\mathcal{T}, S_i \cup S_j) \leq C(\mathcal{T}, S_j)$. That means that coalitions can only merge when the cost of the formed coalition by merge is less than their cost.

For the split rule, a coalition $\hat{S}$ decides to split into two coalitions S_i and S_j based on the split comparison defined by (10), where all GSPs in S_i, S_j, or both are able to keep or improve their individual payoffs. Thus, $\hat{S}$ splits if at least one of the following inequalities is satisfied.

$$\frac{P - C(\mathcal{T}, \hat{S})}{|\hat{S}|} < \frac{P - C(\mathcal{T}, S_i)}{|S_i|} \tag{13}$$

$$\frac{P - C(\mathcal{T}, \hat{S})}{|\hat{S}|} < \frac{P - C(\mathcal{T}, S_j)}{|S_j|} \tag{14}$$

That means that the individual payoff of each GSP in at least one of the splitted coalitions, S_i or S_j, should be higher than its individual payoff in $\hat{S}$.

The stability of the resulting coalition structure is characterized using the concept of defection function $\mathbb{D}$ [15].

Definition 3 (Defection function): A *defection function* $\mathbb{D}$ is a function which associates with each partition $\mathcal{P}$ of $\mathcal{G}$ a group of collections in $\mathcal{G}$.
A partition $\mathcal{P}$ is $\mathbb{D}$-stable if no group of players is interested in leaving $\mathcal{P}$. Thus, the players can only form the collections allowed by $\mathbb{D}$. A defection function $\mathbb{D}_P$ which allows the formation of all partitions of the grand coalition was proposed by Apt and Witzel [15]. $\mathbb{D}_P$-stability is defined based on this function. $\mathbb{D}_P$ allows any group of players to leave the partition $\mathcal{P}$ of $\mathcal{G}$ through merge-and-split rules to create another partition in $\mathcal{G}$. Therefore, $\mathbb{D}_P$-stability means that no coalition has an incentive to merge or split.

III. Distributed Merge-and-Split VO Formation Mechanism (DMSVOF)

The proposed distributed merge-and-split VO formation mechanism (DMSVOF) is given in Algorithm 1. In DMSVOF, the merge and split decisions are made in a distributed fashion. For each VO, DMSVOF selects a GSP as a decision maker, D, that is responsible for making the merge and split decisions. Each VO in the coalition structure $\mathcal{CS}$ has its own decision maker. We denote by D the decision maker of a VO $S \in \mathcal{CS}$, and by $\mathcal{D}$ the set of all available decision makers.

In the description of DMSVOF we denote by B&B-MIN-COST-ASSIGN(S) the function that implements the branch-and-bound method [17] for solving the MIN-COST-ASSIGN problem for a VO S.

DMSVOF starts with a coalition structure $\mathcal{CS}$ consisting of every singleton $G \in \mathcal{G}$ as a VO S in $\mathcal{CS}$. For each VO $S \in \mathcal{CS}$, its only member becomes a decision maker D (i.e., $D = \{G\}$). Each decision maker D uses a vector $visited$ to keep track of all decision makers in $\mathcal{D}$ that are visited for merging. Initially, each D sets all the entries of $visited$ to `FALSE`. That means that all VOs should be considered for merging. Each decision maker D maintains another vector, called $status$, to keep track of the status of all decision makers in $\mathcal{D}$. Initially, D sets the $status$ of all decision makers to `UNKNOWN`. All decision makers must know the status of others.

A GSP G becomes a decision maker D by either receiving a request from a user or by receiving a request from another decision maker in the system. If a user sends a request for executing a job then every singleton $G \in \mathcal{G}$ becomes a VO S. If another decision maker, D', sends a request to G, it also sends the set of GSPs as a VO S. By receiving this request, G becomes a decision maker for S. Then, G changes the $visited$ entry corresponding to D' to `TRUE`. That means, VO S and S' cannot merge, where D and D' are the decision makers of S and S', respectively. D checks if all tasks can be executed by the GSPs participating in its associated VO (line 17). It also computes $v(S)$ based on the allocation. If a GSP is a decision maker of a VO, then, it enters the merge-and-split process (lines 18-48). First, the MERGE procedure is invoked, and then, all decision makers synchronize with each other (line 23). That is, they all wait for every decision maker to find a stable VO, that does not want to merge with any other VOs. Then, if D is still a decision maker, it starts the split

Algorithm 1 Distributed Merge-and-Split VO Formation Mechanism (DMSVOF)

```
1: Every GSP $G$ executes:
2: $\mathcal{D} = \emptyset$
3: for all $G' \in \mathcal{G}, G \neq G'$ do
4:     $D' \leftarrow \{G'\}$
5:     $visited[D'] \leftarrow$ FALSE; $status[D'] \leftarrow$ UNKNOWN
6:     $\mathcal{D} = \mathcal{D} \cup D'$
7: end for
8: if Receive $\langle \text{user}, \mathcal{T} \rangle$ then
9:     $S = \{G\}$
10:    $D \leftarrow \{G\}$
11:    $\mathcal{D} = \mathcal{D} \cup D$
12: end if
13: if Receive $\langle D', S, \mathcal{D}, split \rangle$ then
14:    $visited[D'] \leftarrow$ TRUE
15:    $D \leftarrow \{G\}$
16: end if
17: Map program $\mathcal{T}$ on $S \in \mathcal{CS}$
18: while $D \in \mathcal{D}$ do
19:    $stop =$ MERGE( )
20:    if $stop$ then
21:        $\mathcal{D} = \mathcal{D} \setminus D$, $D$ is not a decision maker anymore
           (Exit from DMSVOF)
22:    end if
23:    Synchronize with the other decision makers
24:    Receive $\langle D', done \rangle$
25:    $status[D'] \leftarrow$ DONE
26:    $[stop, D''] =$ SPLIT( )
27:    if $stop$ then
28:        Broadcast $\langle D, done \rangle$ to all other decision makers in $\mathcal{D}$
29:        if $status[D'] =$ DONE$, \forall D' \in \mathcal{D}$ then
30:            break
31:        end if
32:        Sleep
33:        Receive $\langle D', D'' \rangle$
34:        $visited[D'] \leftarrow$ FALSE
35:        $visited[D''] \leftarrow$ FALSE
36:        $status[D'] \leftarrow$ UNKNOWN
37:        $status[D''] \leftarrow$ UNKNOWN
38:    else
39:        Wakeup
40:        Broadcast $\langle D, D'' \rangle$
41:    end if
42: end while
43: if $D \in \mathcal{D}$ then
44:    Broadcast $\langle D, \langle v(S)/|S| \rangle$ to all other decision makers
45:    Receive the individual values from all other decision makers
46:    Find $D' \in \mathcal{D}$ where $S' = \max_{S \in \mathcal{CS}} \{v(S)/|S|\}$
47:    $D'$ notifies the user to execute program $\mathcal{T}$ on VO $S'$
48: end if
```

process by invoking the SPLIT procedure, otherwise it exits the mechanism. This GSP may become a decision maker again if it receives a request from another decision maker (lines 13-16). In the split process, if the splitting does not occur, *stop* flag is set to TRUE and D broadcasts message DONE to all decision makers. That means, its VO is stable with respect to splitting. If the status of all the decision makers is DONE, for all S in the system, D exits from the merge-and-split while-loop. That means, none of the VOs want to merge or split any more. However, even if one of the decision makers does not finish its decision on merge or split, D goes to sleep.

If the splitting occurs, the *stop* flag is set to FALSE and D sends a wakeup message to all decision makers. All the decision makers that were on sleep condition start the merging process again. Also, they change the $status$ of two new

Algorithm 2 MERGE( ): Distributed Merge

```
1: repeat
2:     Randomly select $D' \in \mathcal{D}$ for which
           $visited[D'] =$ FALSE and
           $status[D'] =$ UNKNOWN$, rank[D] < rank[D']$
3:     Send a Merge request $\langle D, S, merge \rangle$ to $D'$ of $S'$
4:     if Receive $\langle D', ACK \rangle$ then
5:         $visited[D'] \leftarrow$ TRUE
6:         $D'$ becomes a decision maker for $S \cup S'$
7:         return TRUE      {Exit from the Merge process}
8:     end if
9:     if Receive $\langle D', NACK \rangle$ then
10:        $visited[D'] \leftarrow$ TRUE
11:    end if
12:    if Receive $\langle D', remove \rangle$ then
13:        $status[D'] \leftarrow$ REMOVED
14:        $\mathcal{D} = \mathcal{D} \setminus D'$
15:        $\mathcal{CS} = \mathcal{CS} \setminus S'$
16:    end if
17:    Receive a Merge request $\langle D', S', merge \rangle$
18:    $visited[D'] \leftarrow$ TRUE
19:    B&B-MIN-COST-ASSIGN($S \cup S'$)
           {Map program $\mathcal{T}$ on $S \cup S'$}
20:    if $S \cup S' \triangleright_m \{S, S'\}$ then
21:        $S \leftarrow S \cup S'$ {merge $S$ and $S'$}
22:        $S' \leftarrow \emptyset$ {$S'$ is removed from $\mathcal{CS}$}
23:        Send $\langle D, ACK \rangle$
24:        {$D$ is the decision maker of the new VO $S \cup S'$}
25:        $\mathcal{D} = \mathcal{D} \setminus D'$
26:        for all $D'' \in \mathcal{D}, D'' \neq D$ do
27:            Send $\langle D', remove \rangle$ to $D''$
28:            $visited[D''] \leftarrow$ FALSE
29:        end for
30:    else
31:        Send $\langle D, NACK \rangle$
32:    end if
33: until $visited[D'] =$ TRUE$, \forall D' \in \mathcal{D}, D \neq D'$
34: return FALSE
```

decision makers D' and D'' to UNKNOWN, and the $visited$ to FALSE. This is needed since these decision makers are the decision makers of the new VOs. As a result, multiple successive merge-and-split operations are repeated by each D. The mechanism terminates if there are no choices for merge or split for all existing VOs in $\mathcal{CS}$, and thus, the status of all of them is DONE.

If D breaks the while-loop, it means that all VOs terminate from the merge-and-split decisions. Each decision maker broadcasts its individual value to all other decision makers (line 44), it also finds a VO that yields the highest individual value. A decision maker for the VO with the highest individual value sends a message to the user, and it executes the program $\mathcal{T}$.

The merge process (described in Algorithm 2) is a pairwise negotiation between two VOs based on the merge comparison ($\triangleright_m$). The decision maker D of VO S starts the negotiation by choosing a non-visited decision maker in $\mathcal{D}$ randomly and sending a request to merge (lines 1-3). The non-visited decision maker is selected such that the status of that decision maker is UNKNOWN and its rank is higher than that of the current decision maker. The rank of a GSP G, denoted by $rank[G]$, is an integer drawn randomly from the interval $[1, m]$, before the VO formation process starts. We use the ranking to impose an order on the merge requests. The reason of using an order for choosing a decision maker for merge is to prevent deadlocks in the system due to cycles of requests. Decision maker D also sends its VO S to D' as part of the request message. If a decision maker D sends a merge request to D', the decision maker that receives the request, D', analyzes the possibility of merging their VOs, S and S'. Depending on the result of the merge comparison, D' may respond with an acceptance (denoted by ACK) or rejection (denoted by $NACK$) of the merge. The decision maker of a VO responding with an acceptance message becomes a decision maker for the merged VOs, $S \cup S'$ (lines 4-8). D is not a decision maker anymore and it exits the mechanism. If D' responds with a reject message, D changes $visited[D']$ to TRUE, so it will not be selected for the merge (lines 9-11). If D receives a message that D' is not available any more (it has been merged with another VO), D updates the status of D' and the coalition structure.

If D receives a merge request from D', it needs to check whether it can merge with D'. D sets the $visited$ entry corresponding to D' to TRUE and calls B&B-MIN-COST-ASSIGN to find an optimal allocation for the application program $\mathcal{T}$ on $S \cup S'$. If $S \cup S' \triangleright_m \{S, S'\}$, i.e., all the members receive higher profit by merging, D sends an ACK message to D'. $S \cup S'$ is saved in S, and S' is removed from $\mathcal{CS}$. D becomes a decision maker of the merged VO. It also sends a message to all decision makers to remove D' from the set of decision makers $\mathcal{D}$. Since S is changed, it can visit all the existing VOs in the next merge steps. Thus, $visited[D'']$ for all $D'' \in \mathcal{D}$, $D'' \neq D$ is set to FALSE (lines 20-29).

If the merge comparison is not optimal, one or more GSPs would receive less individual payoff in the merged VOs than in their current VO. As a result, D sends a reject message ($NACK$) to D' (line 31). D tries to find another non-visited decision maker suitable for merging. If all the VOs are tested and a merge does not occur ($visited$ is TRUE for all the existing decision makers) D exits from the merge process.

The VO obtained by the merge process is then subject to split. In the split process (presented in Algorithm 3), a VO that has more than one member is subject to splitting. D tries to split S into two disjoint VOs S' and S'', where $S' \cup S'' = S$. D calls B&B-MIN-COST-ASSIGN twice to find an optimal allocation on S' and an optimal allocation on S'' for application $\mathcal{T}$. Since the split is a selfish decision, the splitting occurs even if only one of the members of coalition S' or S'' can improve its individual value. D remains a decision maker for the VO that it belongs to (here, we assume $D \in S$), and it selects a GSP $G'' \in S''$ randomly as a decision maker of S''. D adds D'' to the set of decision makers and sends S'' to D''.

If a VO splits, then the merging process starts again. The merge or split decisions are made in a distributed manner, i.e., each decision maker makes its own decisions. Multiple successive merge-and-split operations are repeated until the mechanism terminates. That means that there are no choices

Algorithm 3 SPLIT( ): Distributed Split

```
1: if $|S| > 1$ then
2:     for all partitions $\{S', S''\}$ of $S$,
           where $S = S' \cup S'', S' \cap S'' = \emptyset$ do
3:         B&B-MIN-COST-ASSIGN($S'$)
               {Map program $\mathcal{T}$ on $S'$}
4:         B&B-MIN-COST-ASSIGN($S''$)
               {Map program $\mathcal{T}$ on $S''$}
5:         if $\{S', S''\} \triangleright_s S$ then
6:             $S \leftarrow S'$ {that is $\mathcal{CS} = \mathcal{CS} \bigcup S' \setminus S$}
7:             $\mathcal{CS} = \mathcal{CS} \bigcup S''$
8:             Select $G''$ where $G'' \in S''$and $\{G''\} \notin \mathcal{D}$ as a
                   decision maker of $S''$
9:             $D'' \leftarrow \{G''\}$
10:            $\mathcal{D} = \mathcal{D} \cup D''$
11:            Send $\langle D, S'', \mathcal{D}, split \rangle$ to $D''$
12:            $visited[D''] \leftarrow$ TRUE
13:            for all $D' \in \mathcal{D}, D' \neq D, D' \neq D''$ do
14:                $visited[D'] \leftarrow$ FALSE
15:            end for
16:            return [FALSE, $D''$]
17:        end if
18:    end for
19: end if
20: return [TRUE, NULL]
```

for merge or split for all existing coalitions in $\mathcal{CS}$.

DMSVOF Properties. We now investigate the properties of DMSVOF. We will first show that DMSVOF produces stable VOs and then investigate its complexity.

Theorem 1: Every partition of $\mathcal{G}$ determined by DMSVOF is $\mathbb{D}_P$-stable.

Proof: (Sketch) There is no cycle of partitions in any sequence of merge and split operations. This is due to the fact that if two VO merge, in the split process, these VOs cannot split because of the individual values of the component GSPs cannot increase. Using the comparison relations $\triangleright_m$ and $\triangleright_s$, the resulting partition after each merge or split is more preferred than a previous partition. Existing partitions do not appear again, and every sub-sequence of merge-and-split operations is acyclic. Since the number of different partitions is finite, the merge-and-split iterations terminate. The final partition cannot be subject to any further merge or split. As a result, the final partition is $\mathbb{D}_P$-stable. ■

Next, we investigate the computational and communication complexity of DMSVOF. The computational complexity of the mechanism is determined by the number of attempts for merge and split. In the worst case scenario, each decision maker needs to make a merge attempt with all the other decision makers in $\mathcal{D}$. The number of decision makers is the same as the number of VOs in $\mathcal{CS}$. In each iteration, the total number of merge attempts for a decision maker is equal to the number of VOs, that is, $O(|\mathcal{CS}|)$. However, the merge process requires a significantly less number of attempts. This is due to the fact that if a VO is found for merge, the merge occurs. In the worst case scenario, splitting a VO S is in $O(2^{|S|})$ which involves finding all the possible partitions of size two of the participating GSPs in that VO. The coalitions in $\mathcal{CS}$ are small sets especially since we apply selfish split decisions that keep the size of the coalitions as small as possible. As a result, the split is reasonable in terms of complexity. In addition, once a coalition decides to split, the search for further splits is not needed.

The communication complexity is defined in terms of number of messages that are exchanged in the mechanism. In the merge process, the total number of messages that a decision maker exchanges is in $O(|\mathcal{CS}|)$. In the split process, if a VO splits, it needs $O(1)$ messages for communication. However, whether it splits or not, the VO needs to broadcast either its status or the id of a new decision maker, thus requiring $O(|\mathcal{CS}|)$ messages.

IV. Experimental Results

Experimental Setup. For our experiments we consider 16 GSPs which is a reasonable estimation of the number of GSPs in real grids. The number of GSPs is small since each GSP is a provider and not a single machine. We use real workloads from the Parallel Workloads Archive [13] to drive our simulation experiments. More specifically, we use the logs from the Atlas cluster at Lawrence Livermore National Laboratory (LLNL). This log consists of recently collected traces (from November 2006 to Jun 2007) that contain a good range of job sizes, from 8 to 8832. We used the cleaned log LLNL-Atlas-2006-2.1-cln.swf which has 43,778 jobs. We selected 21,915 jobs that completed successfully out of all the jobs in the log. About 13% of the total completed jobs are large jobs having runtimes greater than 7200 seconds.

We selected six different sizes (i.e., number of tasks) of the application program from the Atlas log, ranging from 256 to 8192 tasks. For each program, the number of allocated processors the job uses gives the number of tasks, while the average CPU time used gives the average runtime of a task. We used the peak performance of a processor to convert the runtime to workload for each task. We generated the values of the other parameters based on the extracted data from the Atlas log. The parameters and their values are listed in Table I. The values for deadline and payment were generated in such a way that there exists a feasible solution in each experiment.

Each task has a workload expressed in Giga Floating-point Operation (GFLOP). To generate a workload, we extract the runtime of a job (in seconds) from the logs, and multiply that by the performance of a processor in the Atlas system (4.91 GFLOPS). This number gives the maximum amount of giga floating-point operations for a task. We assume that the workload of each task is within $[0.5, 1.0]$ of the maximum GFLOP of the job. The workload vector, w, contains the workload of each task of the application program.

The speed vector s is generated relative to the Atlas system. Each GSP has a speed chosen within the range $4.91 \times [16, 128]$ GFLOPS. This is because each GSP can have several processors capable of performing 4.91 GFLOPS. The reason that we chose this range is that the number of processors of the Atlas cluster is 9,216. If all 16 GSPs have the highest performance of 128×4.91, we would have 2048 processors that is 22.2 percent of the power of the Atlas system. As a result the generated deadline is at most 16 times greater than the runtime. This guarantees that there is a feasible solution for the task allocation. The execution time of each task T_j on each GSP G_i is obtained using the speed vector and the workload vector.

TABLE I: Simulation Parameters

Param.	Description	Value(s)
m	Number of GSPs	16
n	Number of tasks	$[8, 8832]$
s	GSP's speeds ($m \times 1$ vector)	4.91 $\times [16, 128]$ GFLOPS
w	Tasks' workload ($n \times 1$ vector)	$[17676, 1682922.14]$ GFLOP
t	Execution time matrix ($m \times n$)	$\frac{w}{s}$ seconds
c	Cost matrix ($m \times n$)	$[1, \phi_b \times \phi_r]$
d	Deadline	$[0.3, 2.0]$ $\times$ Runtime $\times n/1000$ seconds
P	Payment	$[0.2, 0.4] \times \ max_c \ \times \ n$ units
ϕ_b	Maximum baseline value	100
ϕ_r	Maximum row multiplier	10
Runtime	Runtime of a job from log	$\geq$ 7200 seconds
max_c	Maximum cost	$\phi_b \times \phi_r$

Each cost matrix c is generated using the method described by Braun et al. [18]. First, a baseline vector of size n is generated where each element is a random uniform number within $[1, \phi_b]$. Then, the rows of the cost matrix are generated based on the baseline vector. Each element j in row i of the matrix, $c(i, j)$, is obtained by multiplying the element i of the baseline vector with a uniform random number within $[1, \phi_r]$, a row multiplier. Therefore, one row requires m different row multipliers. As a result, each element in the cost matrix is within the range $[1, \phi_b \times \phi_r]$.

We use the SimGrid toolkit [19] to simulate the grid system. SimGrid is a toolkit for the simulation of distributed applications in heterogeneous distributed environments. We consider that each GSP is a host, and we use a complete graph to set the routes among GSPs. That means, each GSP has a direct link with any other GSPs to send and receive messages. We set the latency of the links to 0.015 milliseconds. We also use the CPLEX solver provided by IBM ILOG CPLEX Optimization Studio for Academics Initiative [20] for solving the MIN-COST-ASSIGN problem.

Analysis of Results. We compare the performance of our distributed VO formation mechanism, DMSVOF, with that of the centralized VO formation mechanism, MSVOF, proposed in [12]. Both mechanisms use merge-and-split operations to find a VO to execute the program. Also, the mechanisms use the branch-and-bound method for solving MIN-COST-ASSIGN and finding the mapping of the tasks to GSPs in a VO. We perform a series of ten experiments in each case, and we present the average of the obtained results.

In Fig. 1a, we show the performance of DMSVOF and MSVOF in terms of individual GSP's payoffs in the final VO as a function of the number of tasks. The figure shows that in some cases DMSVOF provides higher individual payoff for GSPs in the final VO, while in some other cases MSVOF provides higher individual payoff for GSPs in the final VO. The reason is that both mechanisms find stable partitions which may be different. This is expected since for a given problem several stable partitions may exist. However, the results show that there is no significant differences between

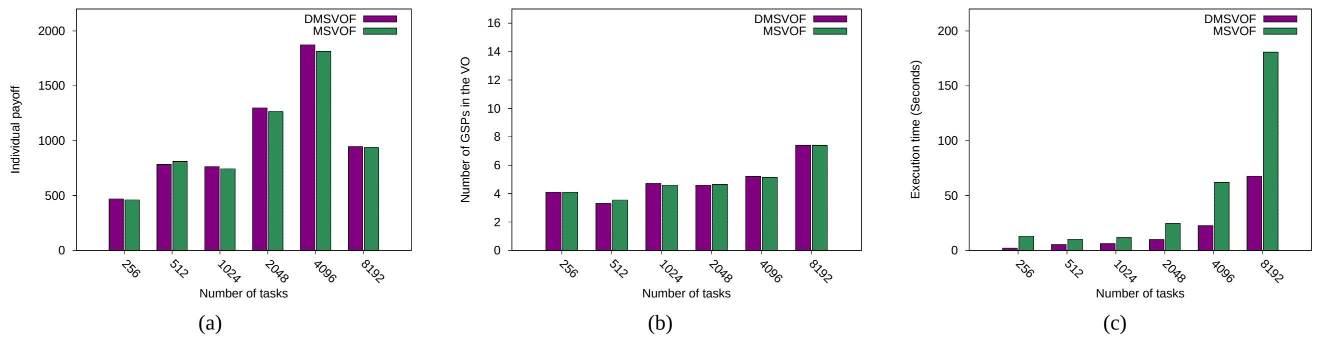

Fig. 1: DMSVOF vs. MSVOF: (a) GSPs Individual Payoff; (b) Size of Final VO; (c) Execution Time.

the individual payoff for GSPs provided by the DMSVOF and the MSVOF. Note that the individual profit may decrease with the increase in the number of tasks. This can happen when an almost the same total profit is divided among the GSPs that are part of a larger VO (e.g., the case for 4096 and 8192 tasks).

In Fig. 1b, we compare the size of the final VO obtained by DMSVOF and MSVOF. This figure shows that as the number of tasks increases the size of the VO obtained by both mechanisms increases. This means that the more tasks, the more GSPs pool their resources to form a VO in order to execute the program. The results show that both mechanisms try to find a small final VO in order to provide higher individual payoff for its GSPs.

Fig. 1c shows the execution time of DMSVOF and MSVOF. These results were obtained on a 3.00GHz Intel quad-core PC with 8GB of memory. The results show that the proposed DMSVOF mechanism is able to reduce the execution time. This reduction is mostly due to the concurrency in the split process, since each decision maker decides for its split regardless of the decisions of the other decision makers. As a result, each merge-and-split iteration of DMSVOF takes less time than an iteration of MSVOF. The reason for getting higher execution times for 4096 and 8192 tasks is that the VOs explored by the mechanism are larger in size. As a result, the split operation takes more time to test the possible cases.

From the above results, we conclude that the proposed distributed VO formation mechanism is not only able to form stable VOs with the highest individual payoff for the GSPs, but also to find the final VO in reasonable amount of time.

V. Conclusion

We modeled the VO formation problem as a coalitional game and designed a distributed VO formation mechanism based on merge-and-split operations (called DMSVOF). We performed extensive experiments with data extracted from real workload traces to investigate DMSVOF's properties. We showed that the sizes of the final VOs obtained by DMSVOF and MSVOF (a centralized mechanism) are equal in most of the cases. We also show that the two mechanisms produce VO's for which the individual GSP's profits are similar. The advantage of DMSVOF is that it determines the final VO much faster than MSVOF. As future work, we would like to consider the task dependencies in our VO formation model and design new distributed mechanisms for VO formation.

Acknowledgment. This research was supported in part by NSF grants DGE-0654014 and CNS-1116787.

References

[1] I. Foster and C. Kesselman, *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 2004.
[2] M. J. Osborne, *An Introduction to Game Theory*. New York, NY, USA: Oxford University Press, 2004.
[3] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger, "Economic models for resource allocation and scheduling in grid computing," *Concur. & Comp.: Practice and Exp.*, vol. 14, no. 13-15, pp. 1507–1542, 2002.
[4] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan, "Analyzing market-based resource allocation strategies for the computational grid," *Intl. J. of High Perf. Comp. Applications*, vol. 15, no. 3, pp. 258–281, Aug. 2001.
[5] "Globus." [Online]. Available: http://www.globus.org
[6] O. Shehory and S. Kraus, "Task allocation via coalition formation among autonomous agents," in *Proc. of Intl. Joint Conf. on Artificial Intelligence*, vol. 14, 1995, pp. 655–661.
[7] I. Müller, R. Kowalczyk, and P. Braun, "Towards agent-based coalition formation for service composition," in *Proc. of the IEEE/WIC/ACM Intl. Conf. on Intelligent Agent Technology*, Dec. 2006, pp. 73–80.
[8] T. Sandholm, K. Larson, M. Andersson, O. Shehory, and F. Tohme, "Coalition structure generation with worst case guarantees," *Artificial Intelligence*, vol. 111, pp. 209–238, 1999.
[9] J. Patel, W. T. L. Teacy, N. R. Jennings, M. Luck, S. Chalmers, N. Oren, T. J. Norman, A. Preece, P. M. D. Gray, G. Shercliff, P. J. Stockreisser, J. Shao, W. A. Gray, N. J. Fiddian, and S. Thompson, "Agent-based virtual organisations for the grid," *Multiagent Grid Syst.*, vol. 1, no. 4, pp. 237–249, 2005.
[10] K. Apt, *Principles of Constraint Programming*. New York, USA: Cambridge University Press, 2003.
[11] T. E. Carroll and D. Grosu, "Formation of virtual organizations in grids: A game-theoretic approach," *Concur. & Comp.: Practice and Exp.*, vol. 22, no. 14, pp. 1972–1989, 2010.
[12] L. Mashayekhy and D. Grosu, "A merge-and-split mechanism for dynamic virtual organization formation in grids," in *Proc. of the 30th IEEE Intl. Perf. Comp. and Comm. Conf.*, 2011, pp. 1–8.
[13] "Parallel workloads archive." [Online]. Available: "http://www.cs.huji.ac.il/labs/parallel/workload/"
[14] C. Weng and X. Lu, "Heuristic scheduling for bag-of-tasks applications in combination with QoS in the computational grid," *Future Generation Computer Systems*, vol. 21, no. 2, pp. 271–280, 2005.
[15] K. Apt and A. Witzel, "A generic approach to coalition formation," *International Game Theory Review*, vol. 11, no. 3, pp. 347–367, 2009.
[16] D. Knuth, *The Art of Computer Programming, Volume 4, Combinatorial Algorithms, Part 1*. Addison-Wesley, 2011.
[17] E. Lawler and D. Wood, "Branch-and-bound methods: A survey," *Operations research*, pp. 699–719, 1966.
[18] T. Braun *et al.*, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel and Distr. Comp.*, vol. 61, no. 6, pp. 810–837, 2001.
[19] The SimGrid project. [Online]. Available: http://simgrid.gforge.inria.fr
[20] "IBM ILOG CPLEX Optimization Studio for Academics Initiative." [Online]. Available: http://www01.ibm.com/software/websphere/products/optimization/academic-initiative/