Applying Organizational Self-Design to a Real-world Volunteer Computing System

Sachin Kamboj, Trilce Estrada, Michela Taufer and Keith S. Decker
Dept. of Computer and Information Sciences
University of Delaware
Newark, DE 19716, USA
{kamboj,estrada,taufer,decker}@cis.udel.edu

Abstract

Organizations are a basis for task allocation, resource allocation and coordination in multiagent systems. Organizational Self-Design (OSD) is an approach to designing suitable organizations at run-time in which the agents are responsible for constructing their own organizational structures. OSD has also been shown to be especially suited for environments that are dynamic or semi-dynamic.

One application of OSD is to allocate and manage resources in grid, cloud and volunteer computing systems. Whereas there has been a body of theoretical research on the underpinnings and algorithms that can be used for OSD, we aren't aware of any implemented system that uses OSD for these kind of applications. This paper bridges this gap by applying OSD to a real-world volunteer computing system. We show that applying OSD to such a system results in better throughput than the current task allocation methods.

1 Introduction

Organizations are a basis for task allocation, resource allocation and coordination in multiagent systems. Generating an organization involves constructing an organizational structure and instantiating this structure with agents.

Contingency theory posits that there is no best way of organizing and all ways of organizing are not equally effective. Instead the optimal organizational structure depends on the problem being solved and the environmental conditions (problem arrival rate, deadlines, etc) under which the problems are being solved.

If the environment is dynamic or semi-dynamic, it precludes the use of a static, design-time generated organizational-structure. Organizational self-design (OSD), in which the agents are responsible for designing their own organizational structures at run-time, has been proposed as a mechanism for designing organizations for such environments [1, 2].

Our primary hypothesis is that multiagent organizations, in general, and OSD, in particular, are especially suited to the problem of generating virtual organizations for

grid-, volunteer-, and cloud- computing systems. This is because (a) such systems are often used to solve complex problems in worth oriented domains and would benefit from having a more flexible workflow representation language that allows quality, cost and duration tradeoffs to be made [3]; and (b) such systems are typically deployed in dynamic and semi-dynamic environments [4].

In this paper, we apply OSD to a real world volunteer computing system. Volunteer computing [5, 6] is a form of distributed computing in which a group of volunteers donate their computing resources to a cause, such as folding proteins, predicting climate change, etc. Currently volunteer computing has been implemented using a master-slave (client-server) architecture. Volunteers download a client that connects to one or more centralized servers and requests jobs that make use of the volunteer's computing resources. The centralized servers, in turn, need to figure out a scheduling policy that tries to perform an optimal allocation of jobs to the clients (for some definition of optimality).

The clients running on the volunteer machines can be thought of as agents. This leads to a direct mapping from the problem of determining a suitable scheduling policy for the clients to the problem of determining a suitable organization for the agents. Hence, the solution to the organizational issues, such as the allocation of agents to the subtasks of the problem being solved and the coordination of inter-agent activities, will generate a scheduling policy that can be used to allocate jobs to the agents.

We focus on applying OSD to the problem of studying protein-ligand docking [7]. Ligands are small molecules that bind to proteins and can be used to regulate their function. Inhibiting the activity of key enzymes (proteins) may result in entire biochemical pathways being turned on or off. Indeed, many small molecule drugs marketed today function by inhibiting enzymes. Hence, protein-ligand docking can be the first step towards discovering new drugs.

To allow various quality[1]/throughput tradeoffs to be made, we use TÆMS (Task Analysis, Environment Modeling and Simulation) to represent the protein-ligand docking workflow. TÆMS [8] is a quantitative extended hierarchical task representation language that can be used to represent complex problems in worth oriented domains. TÆMS has been used to model many different problem-solving environments including distributed sensor networks, information gathering, hospital scheduling, EMS, and military planning[9].

Furthermore, we advocate moving volunteer computing from a strictly master-slave paradigm to a more distributed peer-to-peer model. We show that such a move allows for increased throughput of such systems, while at the same time minimizing the load on the centralized servers.

The main contributions of this paper are as follows:

1. We apply OSD to a real-world volunteer computing system and try to bridge the gap between theoretical OSD research and a practical application of such research to grid-, volunteer-, and cloud- computing systems.

2. We show how the cross-fertilization of ideas from multiagent systems research can benefit volunteer computing systems.

[1]Quality in such systems is usually a function of the accuracy of the docking.

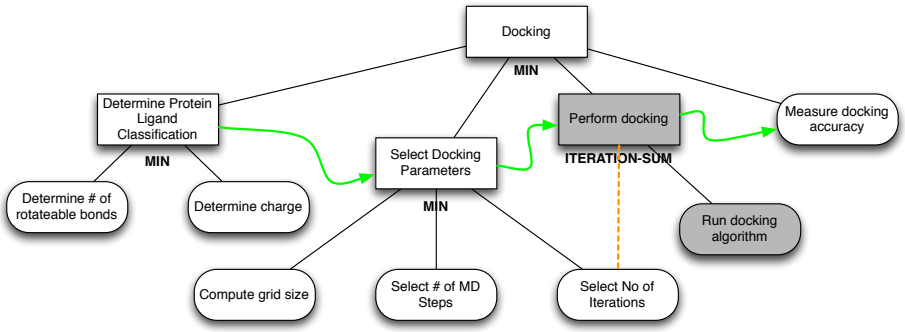


Figure 1: A simplified TÆMS task structure representing protein-ligand docking. The white rectangles represent tasks, the shaded rectangles represent templates the rounded rectangles represent executable methods. The arrows represent non-local effects. *Method outcomes and characteristics are not shown*.

3. We extend TÆMS to allow the representation of templates, a mechanism of generating new TÆMS nodes in predictable way. (See *Extensions to TÆMS* below).

2 Task and Resource Model

Our task and resource model has been previously described in [2]. To summarize, a TÆMS task structure is represented by the tuple $< T, \tau, M, Q, E, R, \rho, C >$, where T is the set of tasks (high-level goals). Each task, $t_i = (q_i, s_i)$, consists of a set of subtasks, $s_i \in (T \cup M)$, which have to be completed in order to complete the task; and a Characteristic/Quality Accumulation Function (CAF), $q_i \in Q$, which describes how the quality of a high level task is computed from the quality of the subtasks. For example, a CAF of MIN indicates that the quality of the task is going to be the minimum of the quality of the subtasks.

M is the set of executable methods. TÆMS is quantitative in that each method may have multiple outcomes with varying probabilities. Each method is represented using a probability distribution, i.e. $M = \{(o_1, p_1), ..., (o_n, m_n)\}$, where o_i is an outcome; and p_i is the probability of that outcome occurring on execution of this method, $\sum_{i=0}^{n} p_i = 1$. Each outcome, $o_i = (q_j, c_j, d_j)$, is, in turn, represented using a quality, cost and duration probability distribution.

E is the set of non-local effect (NLEs); τ represents the highest level goal that the agent is trying to achieve; R is the set of resources; ρ is a function that returns the amount of a resource needed to execute a method; and C is a function mapping a resource to the cost of that resource.

2.1 Extensions to TÆMS

.

Generally, TÆMS task structures can be thought of as being the output of a planning process [10]. This is because TÆMS task structures are basically high-level plans for achieving some goal, in which the steps required for achieving the goal — as well as the possible contingency situations — have been pre-computed offline and represented in the task structure.

Most scheduling [11] and organizational research [12], including our previous OSD research, assumes that the task structure is provided as an input to the system. Since TÆMS task structures are used to represent many contingencies, alternatives, uncertain characteristics and run-time flexible choices, the process of organizing can be thought of as the process of performing long-term task and resource allocation.

For a number of applications, including protein-ligand docking, generating a complete task structure offline (at design time) and providing it as input to the system is often not feasible. This is because the number and type of nodes in the task structure might depend on the result of some execution that has to be done at run-time.[2]

For example, in protein-ligand docking, the number of docking configurations or attempts, and hence the number of docking tasks in the TÆMS task structure, depends on specific protein-ligand pair provided as input to the system. Furthermore factors such as the protein/ligand flexibility and the presence of metals can influence the docking mechanism used.

Hence, for this application, we needed some way of generating TÆMS nodes at run-time while constraining and guiding the kind of nodes that can be generated, without having to resort to an expensive general purpose planner or plan-repair algorithm.

To this end, we extended the TÆMS task representation language by adding *Template Nodes*, $\int = (tp, c, e, q)$, where *tp* $\in$ { ITERATION, SELECTION }, is the type of template node; $c \in T \cup M$ is a *control node*, which constrains the expansion of this template node; $e \subset (T \cup M \cup \int)$ represents the *expansion* of this node. In the case of an ITERATION template node, the expansion node is "generated" a fixed number of times depending on the output of c and $|e| = 1$. In the case of a SELECTION node, one of the subsets of e is selected as the expansion of this node. Finally, $q \in Q$ is a CAF with similar semantics to the CAF of a task.

Hence, in Figure 1 *Perform docking* is an ITERATION template node; *Select Docking Parameters* is the control node; *Run Docking Algorithm* is the expansion node and SUM is the CAF.

3 Organizational Self-Design

Organizations are primarily defined by an organization structure consisting of the roles enacted by the agents and the coordination relationships between the roles.

[2]Note that it might still be possible to represent the complete task structure, *a priori*, by having the task structure contain the maximum number of nodes possible. However, this would often result in an exponential increase in the size of the task structure.

In our approach [2], the input to the system is an extended TÆMS task structure, called the *global task structure*. This task structure represents the problem being solved. The organizational structure is generated by rewriting the *global task structure* and is guided by the environmental conditions, i.e. the task arrival-rate, deadline and available resources.

To form or adapt their organizational structure, the agents use two organizational primitives: agent spawning and composition. These two primitives result in a change in the assignment of roles to the agents.

In order to participate in the organization, and to apply these primitives, the agents need to explicitly represent and reason about the role assignments and must maintain some organizational knowledge. This knowledge is represented in each agent using a TÆMS task structures, called the *local task structure*. Hence, we define a role as a local task structure. These local task structures are obtained by rewriting the global task structure.

In our previous work [2], we have described (a) the organizational nodes needed to represent *local task-strcuctures*; (b) the three operators, *breakup*, *cloning* and *composition* used to rewrite the task structures; and (c) the triggers for organizational change. In the following subsection, we will define a new operator to deal with templates.

3.1 Extensions to OSD

To allow the application of OSD to a real world volunteer computing system, we have made three major changes to our OSD approach presented in [2]:

- We introduce the *Iterate* operator. The *Iterate* operator allows us to deal with ITERATION template node ($\oint \in \int$) by (a) checking the results of executing the control node to determine the number of iterations, n, of the template node required and (b) checking the number of existing copies, m of the *expansion node*, e — If $m < n$, $n - m$ extra copies of the expansion subtree are generated according to the algorithm presented in Algo 1.

- We add *norms* to our OSD approach. Norms are used to constrain the kind of organizations that can be generated. For example, most volunteer computing systems have a requirement that a job must be executed by three different volunteers, and two of the volunteers must produce identical results before the results are "accepted" by the system.

  Since roles are represented in our system using TÆMS nodes ($T \cup M$), norms are implemented as constraints on the nodes of a TÆMS task structure. Currently, we have implemented two kind of norms: (a) unary norms of the form $U(tp, n)$, where $tp \in \{$*NEVER-BREAKUP*, *ALWAYS-BREAKUP*$\}$, is the *type* of the norm — The *NEVER-BREAKUP* type ensures that a TÆMS node will never be selected for breakup by an agent and the *ALWAYS-BREAKUP* norm forces the agent to breakup at that node; and (b) binary norms of the form $B(tp, n, m)$, where $\{tp \in$ *SAME*, *DIFFERENT*$\}$ is the type of the norm and $n, m \in (T \cup M)$ are TÆMS nodes. The *SAME* norm indicates that the nodes n and m should be

Algorithm 1 ITERATE ($\tau \in \Sigma, \upsilon \in \int$)

```
1: Let υ = (tp, c, e, q)
2: n ⇐ RESULTS(c)
3: m ⇐ | SUBTASKS(υ) |
4: τ̄ ⇐ DESCENDENTS(τ) − DESCENDENTS(e)
5: ῡ ⇐ DESCENDENTS(e)
6: for i ← 1, (n − m) do
7:    for all { x | x ∈ ῡ } do
8:       y ⇐ COPYNODE(x)
9:       ADDNODE(υ, y)
10:      for all { N | N ∈ NLES(x) } do
11:         if SOURCE(N) ∈ τ̄ then
12:            z ⇐ FINDINHERITINGNODE(N)
13:            M ⇐ COPYNLE(N)
14:            REPLACENODE(M, SOURCE(M), y)
15:            REPLACENODE(M, SINK(M), z)
16:         else if SINK(N) ∈ τ̄ then
17:            {Similar to the source}
18:         end if
19:      end for
20:   end for
21: end for
```

done by the same agent, whereas the *DIFFERENT* norm implies that the nodes n and m should be performed by different agents.

- Finally, we added the ability to have *soft* deadlines in addition to *hard* ones. Hard deadlines are ones that must be met by the agents — completing a task after a hard deadline has passed results in a quality of 0 on that task (i.e. the task has failed). Soft deadlines, on the other hand, are guidelines to the agents. The agents should make best-effort attempts to try and meet soft deadlines and can make organizational decisions based on the soft deadlines. However, completing a task after a soft deadline has expired still results in positive quality on the task[3].

4 Applying OSD to Docking

4.1 Current Approaches

The use of molecular dynamics for protein-ligand docking[4] was first explored in [7]. In this work, the researchers used a traditional cluster to run their docking attempts. Subsequent work lead to the creation of the docking@home project, a volunteer computing system based on the BOINC framework [5].

[3]We have not currently implemented a penalty for breaking soft deadlines — so agents can essentially ignore soft deadlines. However these deadlines are useful for detecting a need for reorganization.

[4]Henceforth, referred to as simply docking

BOINC (Berkeley Open Infrastructure for Network Computing) is an open-source framework that allows researchers to harness the computing resources of a large and heterogeneous group of volunteers. Task and resource allocation amongst the volunteers is performed by the BOINC middleware, which allocates jobs (tasks) to the volunteers based on a *scheduling policy*.

Current approaches to task allocation in BOINC [13] are based on a naive and greedy algorithm. In the current system, volunteers (agents) periodically issue a *scheduler request* to a centralized server for a certain number of jobs. The centralized server then assigns a group of jobs to the requesting volunteer. To select the assigned jobs, the server randomly scans its cache of outstanding jobs for a set of jobs that can feasibly be run on the volunteer's machine. This set of jobs is assigned to the user.

4.2 Our Approach

In the OSD approach to docking, the task structure presented in Fig. 1 is provided as an input to the system. As with regular OSD, the organization starts of with a single agent responsible for all the activities of the organization. This single agent is equivalent to the BOINC server in regular docking and performs the same functions as the BOINC server — it accepts protein-ligand pairs for docking, from the user, and "generates" a task instance for the input pair. It then applies the standard OSD approach to this task instance, with the exception that (a) spawning a new agent involves removing a volunteer from the volunteer pool and creating an agent (a wrapper or plugin for the standard BOINC docking client) for that volunteer; and (b) composition involves destroying the agent and re-adding the volunteer to the volunteer pool[5].

Two key departures from regular BOINC docking are in the way in which we handle the clients:

1. Instead of a regular master-slave paradigm (i.e. a hierarchical organization that has a depth of 1), the volunteer clients in our approach form a peer-to-peer sub-organization with a much deeper hierarchical structure[6].

   A volunteer that detects an overload while trying to complete the jobs assigned to it will spawn off a new agent. The spawning volunteer will, in turn, be responsible for assigning a subset of its jobs to the newly spawned volunteer. The spawned volunteer may itself spawn-off new agents, hence forming a multi-level hierarchy.

2. As in regular docking, the BOINC server is responsible for assigning jobs to the volunteer agents. However, (a) the volunteers may reassign jobs to other volunteers at lower levels of the hierarchy; and (b) instead of the naive BOINC scheduling algorithm, the assignment of jobs to the lower level sub-organizations is based on the past performance of the sub-organization. Specifically we maintain two pieces of information about each sub-organization, i:

[5]For the purposes of this paper, we assume that the volunteer pool is infinite. In our future work, we would like to investigate OSD approaches that work with a finite albeit changing set of volunteers.

[6]The volunteers only form hierarchical structure is composition is disabled. If composition is enabled, the hierarchy would break down to form an interconnected mesh structure because we allow any agent to compose with any other agent

(a) $t_d[i]$ or the estimated amount of time needed by sub-organization i to complete a *single* job. For every result, j, returned, this time is updated according to the formula: $t_d^{n+1}[i] = \alpha d_j + (1-\alpha)t_d^n[i]$, where $t_d^{n+1}[i]$ is the new value of $t_d[i]$, d_j is the amount of time it took for job j to complete, and α is a constant between 0 and 1.

(b) $t_c[i]$ or the estimated completion time for *all* the jobs assigned to the sub-organization.

Based on these values, the jobs are assigned to the sub-organization based on the equation: $\arg\min_{i \in \text{sub-organizations(k)}}\{t_c[i] + t_d[i]\}$

5 Evaluation

As a first step towards evaluating our approach, we ran a series of experiments comparing our OSD approach to the BOINC approach with 25, 50 and 100 volunteers[7].

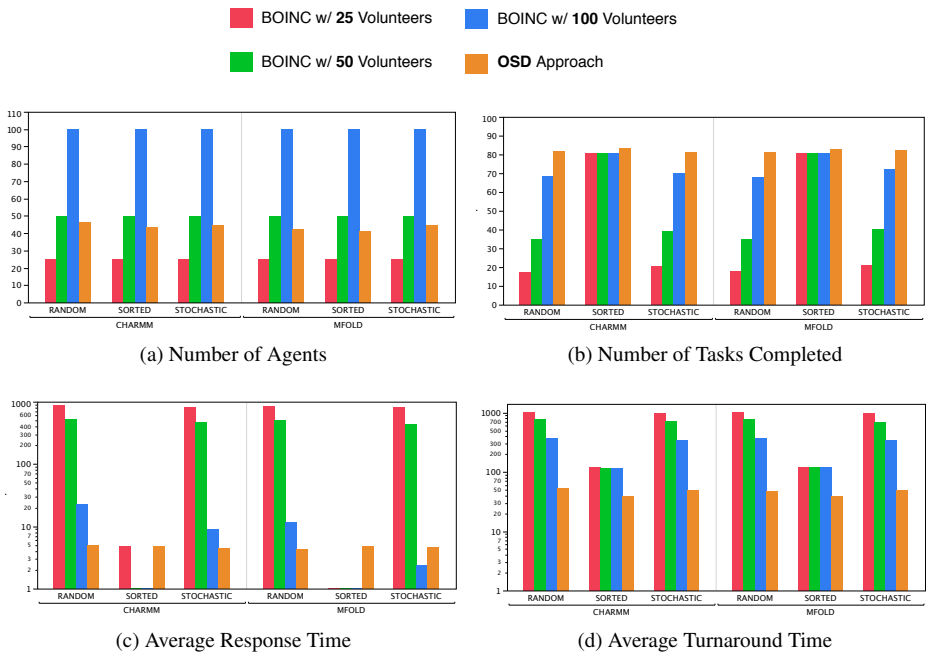
To simulate our approach we needed a volunteer population. To keep the runs realistic, we used the statistical models reported in [14] to compute the *work-in-progress* time or the time needed by a volunteer to generate a result for a job submitted to it (henceforth referred to as the *runtime*). These models were generated from actual BOINC traces. In this paper, we report the results obtained by assuming two different volunteer populations: Charmm and MFold from the Prediction@Home BOINC project.

To generate a volunteer population consisting of n volunteers, we used the statistical models to generate an array of $n.m$ runtimes, where m is the maximum number of jobs generated during a simulation run. We then partitioned this array to generate the runtime behavior of a specific volunteer. This partitioning was done in three different ways:

1. **Random**, the generated array of runtimes was partitioned into n equal parts.
2. **Sorted**, the generated array of runtimes was first sorted and then partitioned into n equal parts. The sorted volunteer population would simulate volunteers with a fixed and predictable runtime behavior.
3. **Stochastic**, the generated array of runtimes was first sorted. However, instead of performing a simple partition, the sorted array was sampled stochastically to generate the volunteer population. That is, for each volunteer, in turn, the next runtime was selected with probability p and randomly with probability $(1-p)$.

Each experiment was repeated 15 times with a new runtime array and a new volunteer population for each run. The results are show in Fig. 2. As can be seen, the initial results are extremely promising. Not only does our approach complete a larger number of tasks than the BOINC approaches, the result is also statistically significant. Furthermore, our approach has a lower response-time and turnaround-time while using fewer than 50 agents on average.

[7] Our OSD approach automatically selects an appropriate number of agents through spawning and composition



(a) Number of Agents

(b) Number of Tasks Completed

(c) Average Response Time

(d) Average Turnaround Time

Figure 2: Graphs showing the average performance of our OSD approach against the BOINC approach with 25, 50 and 100 volunteers.

6 Conclusion and Future Work

This paper presented a first attempt at applying OSD to a real world volunteer-computing application, specifically protein-ligand docking. We show that applying OSD to the job-allocation problem in volunteer computing gave some promising results — our approach completed a larger number of tasks and had a better turnaround time compared to the existing BOINC scheduling approach.

In our future work, we would like to:

1. Scale and test our approach with several hundreds of thousands of volunteers. Specifically we would like to derive bounds for the load on the server and measure how the load varies depending on (a) the number of volunteers; and (b) the depth and branching factor of the organization.

2. Maintain a more accurate history model for the volunteers in our organization. We can anticipate a situation where the time required to complete a job would depend on the time of the day when the job is being run. For example, jobs run during the day when a volunteer's machine is being heavily used would take significantly longer than jobs run during the night when the volunteer is idle. We should be able to vary the number of jobs allocated to an agent depending on how its execution profile varies according to the time of the day.

3. Implement our approach in the standard BOINC server and clients.

References

[1] Ishida, T., Gasser, L., Yokoo, M.: Organization self-design of distributed production systems. IEEE Trans. on Knowledge and Data Engineering **4**(2) 123–134

[2] Kamboj, S., Decker, K.S.: Organizational self-design in worth-oriented domains. In Dignum, V., ed.: Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models. Information Science Reference

[3] Atlas, J., et. al: Flexible grid workflows using taems. Workshop on Exploring Planning and Scheduling for Web Services, Grid and Autonomic Comp. (2005)

[4] Taufer, M., Teller, P.J., Anderson, D.P., III, C.L.B.: Metrics for effective resource management in global computing environments. International Conference on e-Science and Grid Technologies (e-Science 2005). (December 2005) 204–211

[5] Anderson, D.P.: Boinc: A system for public-resource computing and storage. Fifth IEEE/ACM International Workshop on Grid Computing **00** (2004) 4–10

[6] Shirts, M., Pande, V.S.: COMPUTING: Screen Savers of the World Unite! Science **290**(5498) (2000) 1903–1904

[7] Taufer, M., et. al.: Study of a highly accurate and fast protein-ligand docking method based on molecular dynamics. Concurrency and Computation: Practice and Experience **17**(14) (2005) 1627–1641

[8] Lesser, V.R., et. al.: Evolution of the GPGP/TÆMS Domain-Independent Coordination Framework. Autonomous Agents and Multi-Agent Systems **9**(1-2) (2004) 87–143

[9] Maheswaran, R., et. al.: Predictability and criticality metrics for coordination in complex environments. AAMAS 2008 647–654

[10] Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers (May 2004)

[11] Wagner, T., Lesser, V.: Design-to-criteria scheduling: Real-time agent control. AAAI 2000 Spring Symposium on Real-Time Autonomous Systems 89–96

[12] Carley, K.M.: Computational organizational science and organizational engineering. Simulation Modelling Practice and Theory **10** (July 2002) 253–269

[13] Anderson, D.P., Reed, K.: Celebrating Diversity in Volunteer Computing. Hawaii International Conference on System Sciences (HICSS) (2009)

[14] Estrada, T., Taufer, M., Reed, K.: Prediction of upper time bounds from volunteer computing traces. IEEE (2009)