A Multi-Agent Learning Approach to Resource Sharing across Computing Clusters

Chongjie Zhang Victor Lesser
Prashant Shenoy
Computer Science Department
University of Massachusetts Amherst
UMass Computer Science Technical Report UM-CS-2008-035

Abstract

Resource management in clusters traditionally uses centralized approaches, which restricts the cluster scale. To expand this limit, we develop a multi-agent approach to sharing resources across clusters in a decentralized manner. We organize shared clustered into an overlay network and formulate resource sharing in such a network as a distributed sequential resource allocation problem (DSRAP). We then propose a multi-agent reinforcement learning algorithm for each cluster to learn both local allocation decision policy and task routing policy so that clusters cooperatively allocate tasks and maximize the global utility of the system. Heuristic strategies are developed to speed up the learning in such a complex problem. We compare our approach with a centralized allocation approach that can generates optimal solutions in some cases. Experimental results show that our approach is very effective and even outperforms the centralized allocation approach in some cases where it does not generate optimal solutions.

1 Introduction

As "Software as a service" becomes a popular business model, it is becoming increasingly demanding to build computing infrastructures to host application services. *Shared clusters* built using commodity PCs or workstations offer a cost-effective solution for constructing such infrastructures. Unlike a dedicated cluster, where each computing node is dedicated to a single application, a shared cluster can run the number of applications significantly larger than the number of nodes, necessitating resource sharing among applications. Thus the central challenge in shared clusters is resource management that addresses such issues as resource reservation for individual applications, performance isolation between applications, and performance guarantee to applications.

Several approaches have been developed for allocating resources in a shared commodity cluster. A resource-sharing technique for a network of workstations was proposed in [5]. This approach is based on fair relative allocation of cluster resources using proportional-sharing scheduling. Cluster Reserves [4] provides a cluster-wide abstraction that maps the resource assigned to a cluster reserve to individual nodes in the cluster. Cluster Reserves built upon resource containers [7] and employed a linear programming formulation for allocation resources. Sharc [20] focuses on absolute al-

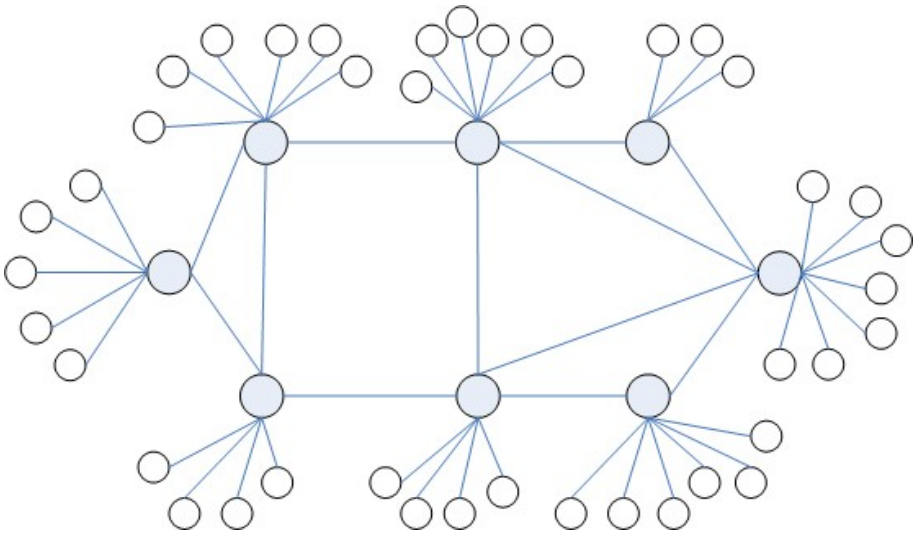

Figure 1: A partially centralized system

location of resources by reservations. Techniques employed by Sharc have complexity that is linear in the number of applications in the cluster.

However, these approaches use centralized resource management, which limits the cluster scale and its computing capacity. To build larger shared computing infrastructures, one common model is to organize a set of shared clusters into a network and enables resource sharing across shared clusters. Figure 1 shows an example of a shared cluster network, where shaded circles represents managers and blank circles represents computing nodes. The resource allocation decision is now distributed to each shared cluster. Each cluster still uses a cluster-wide technique for managing its local resources. However, as task (also referred to applications services) allocation requests vary across clusters, an cluster may need to dynamically decide what tasks to allocated locally and where to forward unallocated tasks to cooperatively optimize the global utility of the whole system. To achieve scalability, each cluster has limited number of neighboring clusters that it interacts with.

We describe this decision problem as a distributed sequential resource allocation problem (DSRAP). We consider DSRAP is a novel and practical application for multiagent learning. In DSRAP, each agent (referred to a cluster) has only a partial view of the whole system and does not have access to the system-level utility (because it is not directly measurable in real-time). All agents make decisions concurrently and autonomously. Each agent's decision depends not only on its local state but also on other agents' states and policies.

This paper is intended to demonstrate applicability and effectiveness of multiagent learning for DSRAP or similar distributed problems. We propose a multi-agent learning algorithm, called Fair Action Learning (FAL) which is a variant of the Generalized Infinitesimal Gradient Ascent (GIGA) algorithm [23], for each agent to learn local decision policies. To simplify the learning, we decomposes each agent's decisions into two connected learning problems: *local allocation problem* (deciding what tasks to be allocated locally) and *task routing problem* (deciding where to forwarded a task). To avoid poor initial policies during learning, heuristic strategies are developed to speed

up the learning. The learning approach is tested in a network of simulated clusters and compared with a centralized greedy allocation approach, which is optimal in some cases. Experimental results show that our multi-agent learning works effectively and even outperforms the centralized approach in some cases. Although we discuss our approach in this particular problem, it can be more generally useful in other online resource allocation problems, for example, when shared resources are storage devices in distributed file systems, documents in peer-to-peer information retrieval, or energy in sensor networks.

The rest of this paper is structured as follows. Section 2 formulates the problem above as a distributed sequential resource allocation problem. Section 3 introduces background knowledge about reinforcement learning and multi-agent reinforcement learning. Section 4 describes decision-making processes of our approach and learning models for each type of decisions. Section 5 describe experiment design and analyzes experimental results. Related work is presented in Section 6. Finally, Section 7 concludes our work and discusses future work.

2 Problem Formulation

The runtime model of DSRAP is described as follows. Each cluster manager receives tasks either from the external environment or a neighboring manager. At each time step, a manager makes decisions on what tasks are allocated locally and how remaining tasks are forwarded to neighboring managers. Due to the task transfer time cost, there is communication delay between two agents. To reduce the communication overhead, the number of tasks a manager transfers at each time step is limited. When a task is allocated locally, the manager gains an amount of utility at each time step, which is specified by the task utility rate. If a task can not be allocated within its maximum waiting time, it will be removed from the system. If an allocated task is finished, all resources it occupies will be freed and available for future tasks. The main goal of DSRAP is to derive decision policies for each manager that maximize the average utility rate (AUR) of the whole system .

We denote a DSRAP with a tuple $\langle \mathscr{C}, \mathcal{A}, \mathcal{T}, \mathcal{B}, \mathcal{R} \rangle$, where

- $\mathscr{C} = \{C_1, \ldots, C_m\}$ is a set of shared clusters.
- $\mathcal{A} = \{a_{ij}\} \in \Re^{m \times m}$ is the adjacent matrix of clusters and each element a_{ij} is the task transfer time between cluster C_i and cluster C_j.
- $\mathcal{T} = \{t_1, \ldots, t_l\}$ is a set of task types.
- $\mathcal{B} = \{D_{ij}\}$ is the task arrival pattern and each element D_{ij} is the arrival distribution of tasks of type t_j at cluster C_i.
- $\mathcal{R} = \{R_1, \ldots, R_q\}$is a set of resource types (e.g., CPU and network) that each cluster provides.

Each cluster $C_i = \{n_{i1}, n_{i2}, \ldots, n_{ik}\}$ contains a set of computing nodes. Each computing node n_{ij} has a set of resources, represented as $\{\langle R_1, v_{ij1} \rangle, \ldots, \langle R_q, v_{ijq} \rangle\}$, where R_h ($h = 1, \ldots q$) is the resource type and $v_{ijh} \in \Re$ is the capacity of resource R_h on node n_{ij}. We assume there exist standards that quantify each type of resource. For example, we can quantify a fast CPU as 150 and a slow one with a half speed as 75.

In DSRAP, each task consists of at least one *capsule* or more if the task is distributed. A capsule is a component of a task that can run on an individual computing node. A task with multiple capsules can run on one node or multiple nodes. Note that resource management of shared clusters guarantees resources for each capsule, so multiple capsules running in parallel on one node will perform the same as running on multiple nodes. A capsule is not decomposable and can only run on one node. One task can only run in one cluster.

A task type characterizes a set of tasks. Tasks of one type have the fixed number of capsules. A task type t_i is also denoted as a tuple $\langle D_i^s, D_i^u, D_i^w, D_i^{rs_1}, \ldots, D_i^{rs_n} \rangle$, where

- D_i^s is the task service time distribution
- D_i^u is the task utility rate (utility per time step) distribution
- D_i^w is the distribution of the task maximum waiting time before being allocated
- $D_i^{rs_j}$ is the resource specification distribution of capsule j ($j = 1, \ldots, n$) of a task. $D_i^{rs_j} = \{\langle R_1, D_i^{rs_{j1}} \rangle, \ldots, \langle R_q, D_i^{rs_{jq}} \rangle \}$ and $D_i^{rs_{jh}}$ ($h = 1, \ldots q$) is a resource demand distribution of capsule j for resource R_h.

A task is denoted as a tuple $\langle t, u, w, RS_1, \ldots, RS_n \rangle$, where

- t is the task type.
- u is the utility rate of the task.
- w is the maximum waiting time before being allocated.
- RS_i is the resource specification of capsule $i = 1, \ldots, n$. $RS_i = \{\langle R_1, d_{i1}, \ldots, R_q, d_{iq} \rangle$, where d_{ih} ($h = 1, \ldots q$) is capsule i's demand for resource R_h.

Based on the model of DSRAP developed above, the average utility rate of the whole system to be maximized can be defined as following:

$$AUR = \lim_{n \to \infty} \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{x \in T_i(C_j)} u(x)}{n} \tag{1}$$

where $T_i(C_j)$ is the set of tasks that allocated to cluster C_j at time i and $u(x)$ is the utility of task x. Note that, due to its partial view of the system, each individual cluster can not observe the system's AUR.

3 Background

3.1 Reinforcement Learning

Reinforcement learning (RL) [19] addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals. The agent's learning environment is typically formulated as a Markov Decision Process (MDP). An MDP for a single agent (decision maker) can be described by a tuple $\langle S, A, T, r \rangle$, where

- S is the state space,

- A is the action space,
- T is the transition function and $T(s, a, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$,
- r is the reward function and $r(s, a, s')$ is the immediate reward (or expected immediate reward) received after transition to state s' from state s by executing action a.

The agent's goal is to maximize, at each time step k, the discounted total reward:

$$R_k = \sum_{j=0}^{\infty} \gamma^j r(s_{k+j}, a_{k+j}, s_{k+j+1}), \tag{2}$$

where $\gamma \in (0, 1)$ is the discount factor.

A solution to a MDP is called a *policy*. A *deterministic* policy directly maps each state to one action, while a *stochastic* policy specifies a probability distribution over the available actions for each state. Both can be represented as a function $\pi(s, a)$, which specifies the probability that an agent will execute action a at state s. An optimal policy is a policy that allows an agent to achieve its goal and maximize the discounted total reward at each time step.

If the underlying MDP model of a learning environment is completely known, the optimal policy can be directly derived through dynamic programming. However, in most real problems, this condition does not hold. The key feature of RL is that it can learns an optimal policy through a trial-and-error exploration in the absence of explicit MDP models.

Algorithm 1: Q-learning Algorithm

begin
  Initialize $Q(s, a)$ arbitrarily
  repeat
    Initialize s
    repeat
      Choose action a from state s using policy derived from Q (e.g., ϵ-greedy)
      Take action a, observe r, s'
      $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
      $s \leftarrow s'$
    until *s is terminal*
  until *the learning stops*
end

The Q-learning [21] algorithm is an example RL algorithm that learns a action-value function $Q(s, a)$ estimating an agent's long-range expected value, starting in state s and taking initial action a. The action-value function update rule of Q-learning is defined by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)], \tag{3}$$

where α is the learning rate, γ is the discount factor, and r_{t+1} is the immediate reward at time $t + 1$ after taking action a in state s. Algorithm 1 shows the procedural form

of the Q-learning algorithm. With the update rule 3, the learned action-value function, Q, directly approximates Q^*, the optimal action-value function, independent of the policy being followed. A deterministic optimal policy π^* can be derived from Q^* as following:

$$\pi^*(s,a) = \begin{cases} 1 & \text{if } a = \text{argmax}_{a'} Q^*(s,a') \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

3.2 Multi-Agent Reinforcement Learning

A multi-agent system (MAS) is a system composed of multiple interacting intelligent agents. In a typical MAS, each agent has only a partial view of the system, but it is autonomous or at least partially autonomous in decision-makings. A MAS is a *cooperative* MAS if all agents' ultimate goals are to maximize the *social or global* utility (the utility gained by the whole system). For example, our model for DSRAP is a cooperative MAS.

In the single-agent setting, several interesting and powerful theorems guarantee that RL algorithms, such as Q-learning, learn optimal value functions and optimal policies for MDP environments when lookup tables are used to represent the state-action value function. However, in the multi-agent setting, due to the non-stationary environment (all agents are simultaneously learning their own policies), the usual conditions for single-agent RL algorithms' convergence to an optimal policy do not necessarily hold [10, 13, 8]. As a result, the learning of agents may diverge due to lack of synchronization. Several multi-agent reinforcement learning (MARL) algorithms have been developed to address this issue [23, 8, 1], with convergence guarantee in specific classes of games and with two agents.

Algorithm 2: Fair Action Learner (FAL) Algorithm

begin
  $r \leftarrow$ the cost for action a at state s
  update Q-value table using $< s, a, r >$
  $\bar{r} \leftarrow$ average reward $= \sum_{a \in A} \pi(s,a) Q(s,a)$
  foreach *action* $a \in A$ **do**
    $\Delta(s,a) \leftarrow \zeta(Q(s,a) + \bar{r})$
  end
  $\pi(s) \leftarrow limit(\pi(s) + \Delta(s))$
end

In this paper, we used a multi-agent reinforcement learning algorithm, called Policy Gradient Ascent (PGA), to learn decision-making policies. The PGA algorithm is a direct policy search technique and a variant of the GIGA algorithm [23] that approximates the policy gradient of each state-action pair with the difference of the expected Q-value on that state and its Q-value. Algorithm 2 describes its policy update rule, where ζ is the policy learning rate. To normalize $\pi(s)$ such that it sums to 1, the *limit* function from GIGA [23] is applied with minor modifications so that every action is explored with minimum probability ϵ:

$$\pi(s) = limit(\pi(s)) = argmin_{x:valid(x)} |\pi(s) - x|$$

i.e., $limit(\pi(s))$ returns a valid policy that is closest to $\pi(s)$.

One advantage of the PGA algorithm is that it learns stochastic policies and does not require a global reward signal. As argued in [18], stochastic policies can work better than deterministic policies in partially observable environments, if both are limited to act based on the current percept. The PGA algorithm needs an algorithm to update its state-action value function. We use the Q-learning algorithm in this paper.

4 Learning the Distributed Sequential Resource Allocation

In DSRAP, at each time step, a manager may receive tasks from neighboring managers or the external environment. It needs to decide what tasks are to be allocated in the local cluster (if its resources are available) and how to forward the remaining tasks. Since each manager makes decisions to maximize the global utility of the system, whether to allocate a subset of tasks locally depends on both resource availability of its local cluster and other clusters in the network. One straightforward approach is to unify the decision-makings above as one. However, this idea faces a difficult problem, that is, how to assign the reward for forwarding a task to neighboring managers. This reward should be comparable to the utility the manager gains when allocating a task locally. For example, if the manager gains all or partial utility of the task for forwarding it, then other managers have no or less incentive to allocate this task than tasks directly from the external environment. If receiving no utility, it will greedily allocate all tasks locally, if local resources permit, and can not distinguish which neighbor is a better candidate for this task. Another option is to duplicate the utility of the task, that is, all managers that forward or allocate this task receive its utility. This option causes the sum of local utilities not to be equal to the real global utility the whole system gains. Obviously, looping tasks in the network tends to increasing the local utility of each agent. This tendency results in "bad" decision policies of managers that do no good for the global utility.

Algorithm 3: General DSRAP Decision-Making Algorithm

```
begin
    TASKS ← set of tasks received in current time cycle;
    ALLOCATED_TASKS ← selectAndAllocateTasks(TASKS);
    TASKS ← TASKS \ ALLOCATED_TASKS ;
    foreach task t ∈ TASKS do
        chooseANeighborAndForward(t) ;
    end
end
```

We approach this problem by decoupling decisions above into two decision-makings: *local allocation decision* (whether to allocate a task locally) and *task routing decision* (to which neighboring cluster a task is forwarded). Algorithm 3 shows the general decision-making process for DSRAP. This algorithm uses two functions: *selectAndAllocateTasks* and *chooseANeighborAndForward*. Function *selectAndAllocateTasks* is responsible for the first decision-making that selects and allocates a subset of received tasks to its local cluster in order to maximize its local utility. The manager gains utilities only from locally allocated tasks. The loop *foreach* statement accounts for the second decision-making. For each task that is not allocated locally, function

chooseANeighborAndForward compares potential processing capabilities of neighbors and chooses a better candidate and forward the task in order to maximize its allocation probability. As the second decision policy of a manager depends on the first decision policies of other managers, our approach also explicitly establishes connections between two decision-makings, which will be discussed later.

The way that we model the decision process for DSRAP has following advantages:

- As the reward function for the second decision-making is only used to compare neighbors, it does not have to be comparable with that of the first decision-making.
- The global utility of the system is exactly equal to the sum of local utilities of all managers. Therefore, the global utility can potentially increase as each manager improves its local utility.
- Our approach also deals well with the issue of information asymmetry. Each manager has full knowledge about its own state, but limited knowledge about each neighboring manager. By separating decisions, each manager not only takes advantages of its detailed local information, but also allows neighbors to more fairly compete for tasks.
- As two decisions are inherently connected, our approach does not cut down this connection.
- Our approach allows managers to demonstrate both self-interest (in the first decision-making) and cooperativeness (in the second decision-making). As the global utility is the sum of local utilities, both characteristics direct and indirectly tends to improve the performance of the whole system.

The environment where managers situate is open, dynamic, and non-stationary. No predefined decision policies can work effectively in all situations. Each manager needs to learn and adapt its policies to the dynamics of both the environment and other agents. The following subsections will describe in detail both decision-making processes and how to learn decision policies.

4.1 Modeling the First Decision-Making

Algorithm 4 shows the first decision-making process for DSRAP. This algorithm incrementally selects and allocate tasks locally. It uses three functions: *getAllocable-Tasks, allocate*, and *learn*. Function *getAllocableTasks* filters *tasks* based on current local resource availability and returns allocable tasks. Function *allocate* is responsible for allocating resources to the task and update local resource availability information. Function *learn* updates its allocation decision policy for selecting a task. Here we use π_1 to denote a policy for the first decision-making. *VOID_TASK* is a unique, fake task without resource requirements and utility. If this task is selected, then the remaining tasks will be not allocated and the allocation process stops.

4.1.1 The Learning Model for the First Decision-Making

To model this decision-making as a learning problem, we first define the state space, the action space, and the reward function. A manager's allocation decision may depend on various information, including availability of each resource on each computing node

Algorithm 4: *selectAndAllocateTasks*(TASKS)

```
Input: Set of tasks
Output: Set of allocated tasks
begin
    ALLOCABLE_TASKS ← getAllocableTasks(TASKS);
    ALLOCATED_TASKS ← ∅ ;
    while ALLOCABLE_TASKS ≠ ∅ do
        ALLOCABLE_TASKS ← ALLOCABLE_TASKS ∪ {VOID_TASK} ;
        update current state s with local resource availability and
        ALLOCABLE_TASKS;
        t ← task selected from ALLOCABLE_TASKS according to decision
        policy π1(s, ·);
        if t = VOID_TASK then
            ALLOCABLE_TASKS ← ∅;
        else
            allocate(t);
            ALLOCATED_TASKS ← ALLOCATED_TASKS ∪{t} ;
            TASKS ← TASKS \{t} ;
            ALLOCABLE_TASKS ← getAllocableTasks(TASKS);
            learn(s, t);
        end
    end
    return ALLOCATED_TASKS;
end
```

in the local cluster and information about tasks to be allocated. As most of these information values are continuous, the decision state space is immensely huge. In order to make the learning problem tractable, we approximate the original state space by using an abstract state space.

Each abstract state $s = \langle s_t, s_c \rangle$ consists of two feature vectors s_t and s_c, respectively describing the task set to be allocated and availability of various resources in a cluster. We assume the task type of a task can approximately represent information about the task. So s_t can be represented as a vector $\langle y_1, y_2, \ldots, y_m \rangle$, where each feature y_i corresponds to task type i and m is the number of task types. If the task set to be allocated contains a task with type i, then $y_i = 1$.

To describe resources in a cluster, we first categorize availability of each resource into multiple levels and then use combinations of levels of different resources as features to represent s_c. The value of a feature is the number of computing nodes in the cluster that have such resource levels. For example, if availability of CPU or network on a computing node is described with two levels: *LOW* and *HIGH*, then there are 4 combinations or features: $x_1 = \langle LOW, LOW \rangle, x_2 = \langle LOW, HIGH \rangle, x_3 = \langle HIGH, LOW \rangle$, and $x_4 = \langle HIGH, HIGH \rangle$, where CPU is the first component of vectors. Feature vector $s_c = \langle x_1, x_2, x_3, x_4 \rangle$ describes resource availability of a cluster. For example, a feature vector of a cluster with 5 nodes can be $\langle 0, 4, 1, 0 \rangle$, which represents that the cluster has 4 nodes with LOW CPU availability and HIGH network availability and one node with HIGH CPU availability and LOW network availability.

An action of this decision is to select a task to allocate. So each task t corresponds to an action. In a real environment, it is not frequent to see two exact same tasks. To

reduce the action space, the type of the task is used to approximately represent the task itself. Therefore, action set is mapped to the set of task types. Then binary feature vector s_t of an abstract state s determines available actions for state s. It is possible that one task set to be allocated may have several tasks with the same type. When such task type is selected, the task of this type with the greatest utility rate will selected and allocated.

The reward function $r(s,t)$ returns the utility rate for allocating task $t = \langle S, t, w, u \rangle$ at state s, which is defined as following:

$$r(s,t) = \begin{cases} 0 & \text{if } t = \text{VOID\_TASK} \\ u & \text{otherwise} \end{cases} \tag{5}$$

In addition to the external environment, a manager may also receive tasks from other managers. Decision policies of other managers have an impact on task arrivals at this manager and consequently affect its learning. As all managers are simultaneously learning their own policies, the learning environment of each manager is non-stationary. In addition, each manager only interacts their immediate neighbors and has partial observation of the system. As argued in [18], stochastic policies can work better than deterministic policies in partially observable environments, if both are limited to act based on the current percept. For each manager, we choose Q-learning to update its value function $Q(s,a)$ and use PGA algorithm to learn its allocation decision policy $\pi_1(s,a)$.

As the allocation decision policy $\pi_1(s,a)$ is stochastic, the update rule of Q-learning needs to be changed as follows:

$$\begin{aligned} Q(s_n,a_n) &\leftarrow Q(s_n,a_n) + \alpha[r_{n+1} + \gamma E\{Q(s_{n+1},a_{n+1})|s_n\} - Q(s_n,a_n)] \\ &\leftarrow Q(s_n,a_n) + \alpha[r_{n+1} + \gamma \textstyle\sum_s p(s|s_n,a_n) \sum_a \pi(s,a) Q(s,a) - Q(s_n,a_n)] \end{aligned}$$

where $p(s|s_n,a_n)$ is the probability of transiting into state s by executing action a_n at state s_n, and $\pi(s,a)$ is the probability of executing action a at state s. The new update rule is just like that of Q-learning except that instead of the maximum over next state-action pairs it uses the expected value, taking into account both the state-transition probability and the probability each action will be executed under the current policy. The new update rule needs the state-transition probability, because the next state may not be determined until the task set to be allocated is known. The state-transition probability can be estimated during the learning.

4.1.2 Accelerating the Learning Process

Even when using approximated state space and action space developed above, the state-action space of each agent is still extremely large. Assume that a cluster has n computing nodes, m types of resources, and receives k types of tasks and availability of each resource is discretized into d levels, the size of the state-action space is $k2^k n^{d^m}$. In addition, any pure knowledge-free RL exploration strategies could entail running arbitrarily poor initial policies, which should be avoided in the live system. To address those issues, we proposed several techniques to guide the exploration. Policies are initialized with a greedy allocation algorithm, which allocates all tasks in an decreasing order of their utilities if resources permit. The ϵ-greedy strategy is used to allow each action to be explored with a minimum rate. To avoid unwanted system performance, we set a utilization threshold for each cluster. If the utilization of every resource is below this threshold, then the manager stops ϵ-greedy exploration and uses the greedy

algorithm for exploration. In addition, note that rejecting too many tasks will degrade the system performance. We also limit the exploration rate to select VOID_TASK.

4.2 Modeling the Second Decision-Making

The second decision-making for DSRAP addresses the question: to which neighboring manager should a manager forward a given task to get it to a wanting cluster before it expires? As the supply of resources are usually less than the demand of tasks, not all tasks will be allocated to some cluster. The performance of this decision-making can be measured by the probability that a task will be allocated in some cluster. With the approach developed above for the first decision-making, each cluster manager learns to choose and allocate tasks to maximize its utility, so improving the task forwarding policy will increase the global utility of the system.

Algorithm 5: *chooseANeighborAndForward*(task t)

Input: A task to be forwarded
begin
  update current state s with task t;
  $n \leftarrow$ a neighbor selected according to decision policy $\pi_2(s, \cdot)$;
  forward task t to neighbor n ;
  $r \leftarrow$ the reward returned from neighbor n for task t ;
  learn(s, n, r) ;
end

However, there is no "training signal" for directly evaluating or improving the forwarding policy until a task is either allocated or abandoned. To address this issue, we employ reinforcement learning to update the policy using only local information. As previous work [22] showed that PGA outperformed Q-learning in distributed decision-making problems, we choose PGA to learn the forwarding policy. Algorithm 5 shows the second decision-making process on forwarding a task. The learning model is described below.

4.2.1 The Learning Model for the Second Decision-Making

The state s_x is defined by the characteristics of the task x that an agent is forwarding. More specifically, s_x can be represented by a feature vector $\langle t_x, w_x \rangle$, where t_x is the type of the task x and w_x is the remaining waiting time of the task x. We assume the type of a task can characterize the task itself. An action j corresponds to choosing neighboring manager j for forwarding a task. We define $Q_i(s_x, j)$ as the expected probability that the task x will be allocated if an agent i forwards it to its neighbor j, and $\pi_{2i}(s_x, j)$ as the probability that agent i will forward task x to agent j.

Upon sending a task to a, agent i immediately gets the reward single $r(s_x, j)$ from the agent j's. The reward $r(\langle t_x, w_x \rangle, j)$ is the estimated probability that the task x will be allocated based on agent j current policies, namely

$$r(s_x, j) = p_j(x) + (1 - p_j(x)) \sum_{k \in \text{neighbors of} j} \pi_{2j}(s'_x, k) * Q_j(s'_x, k) \qquad (6)$$

where $p_j(x)$ is the probability that agent j will allocate task x locally, π_{2j} is the policy of agent j for the second decision-making, and s'_x is the state where agent j makes a

	Greedy	FDL	SDL	BDL
First Decision-Making Algorithm	Best-first	Learning_1	Best-first	Learning_1
Second Decision-Making Algorithm	Random	Random	Learning_2	Learning_2

Table 1: Distributed resource allocation approaches

decision for forwarding task x. If the state $s_x = \langle t_x, w_x \rangle$, then $s'_x = \langle t_x, w_x - a_{ij} \rangle$, where a_{ij} is the time cost for transferring a task between agent i and agent j.

The probability $p_j(x)$ depends on agent j's allocation policy π_{1j} for the first decision-making. It is calculated as following:

$$p_j(x) = \begin{cases} 0 & \text{if agent } j \text{ has no resources for task } x \\ & \text{or task } x \text{ will be expired upon arriving} \\ \sum_{s_t} q(\langle s_c, s_t \rangle | t) \pi_{1j}(\langle s_c, s_t \rangle, t) & \text{otherwise} \end{cases} \tag{7}$$

where t is the type of task t, s_c is the current feature vector of resource availability, $q(\langle s_c, s_t \rangle | t)$ is the probability that agent j is on state $\langle s_c, s_t \rangle$ when it allocates tasks with type t, and π_{1j} is the policy of agent j for the first decision-making. The probability $q(\langle s_c, s_t \rangle | t)$ can be estimated during the learning.

The simple version of Q-learning algorithm is used to update agent i's estimate:

$$Q_i(s_x, j) = (1 - \alpha) * Q_i(s_x, j) + \alpha * r(x_s, j)$$

where α is a learning rate (usually 0.5 in our experiments). With updated Q-values, the PGA algorithm revises its policy π_{2i}.

4.2.2 Dual Exploration

Exploration of Q-values is done locally between neighboring agents during a task transfer to avoid excessive exploration overhead. In the learning model developed for the second decision-making, only one Q-value ($Q_i(s_x, j)$) is updated when agent i forwards task x to agent j. However, one more Q-value ($Q_j(s_x, i)$) can also be updated in the same transfer. This idea of using information about the traversed path for exploration in the reverse direction is called *backward exploration* [14]. When agent i transfer task x to its neighbor j, the message that contains task x can take along reward information $r(s_x, i)$ of agent i about allocating x. This reward information can be used by agent j to update its own estimated pertaining to i. Later when agent j has to make a decision, it has the updated Q-value for i. As a result, backward exploration speeds up the learning.

5 Experiments

5.1 Experiment Design

To evaluate the performance of learning models developed in Section 4, we compared five resource allocation approaches: *greedy allocation*, *first-decision learning* (FDL), *second-decision learning* (SDL), *both-decision learning* (BDL), and *centralized allocation*. The first four approaches are distributed techniques and derived from our two-decision model developed in Section 4. As shown in Table 1, they use different

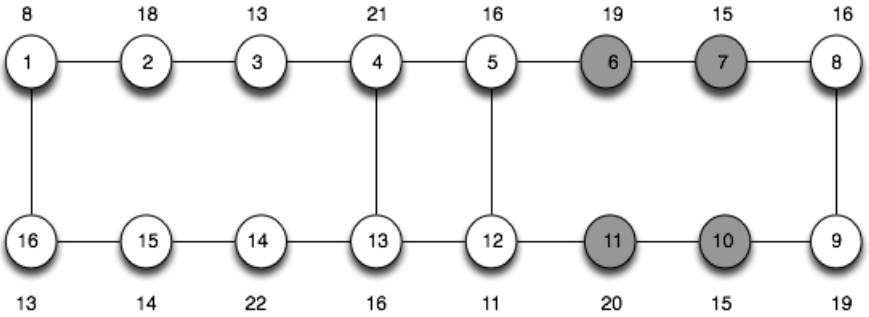

Figure 2: The network topology with 16 clusters

algorithms for each decision-making. With *best-first algorithm* for the first decision-making, at each time step, a manager first sorts all received tasks in a descending order of utility rate and then processes tasks one by one. If there are resources available for a task, the manager then allocates it to its local cluster. *Learning*$_1$ and *Learning*$_2$ respectively refer to the learning algorithms we developed for the first and second decision-making. Note that, for SDL, the task allocation probability $p_j(x)$ in Equation 6 will be directly estimated from the past allocation experience. With *random* algorithm for the second decision-making, a manager randomly picks a neighboring manager to forward an unallocated task. The *centralized allocation* approach has only one manager that fully controls all computing nodes and uses *best-first* algorithm to allocate tasks.

In *Learning*$_1$, availability of each resources needs to be discretized. Our approach is to dynamically build a Gaussian distribution for each resource and discretize its availability into multiple levels, all of which have the same probability. Gaussian distribution's parameters, *mean* and variance, are estimated from the resource specification of tasks. In our experiments, each resource are categorized into three levels.

We have tested approaches on several network topologies with 2, 4, 8, and 16 clusters, all of which show similar results. Here we present detailed results for a network topology with 16 clusters pictured in Figure 2. The number outside a circle represents the number of computing nodes of that cluster. For example, cluster 1 has 8 computing nodes and cluster 4 has 21 nodes. The CPU capacity and network capacity vary on different computing nodes, whose range is in $[50, 150]$.

We assume all tasks have one capsule and the same maximum waiting time w (Our experiments use $w = 10$). Each type can be represented by a feature vector $\langle d_{cpu}, d_{network}, u, st\rangle$, where d_{cpu} is the average demand for CPU, $d_{network}$ is the average demand for network bandwidth, u is the average utility rate and st is the average service time. We assume CPU demand, network demand, and utility rate of tasks of each type are under Poisson distribution, and their service time are under exponential distribution. Our experiments use four types of tasks: *ordinary, IO-intense, compute-intense*, and *demanding*. Their feature vectors are respectively $\langle 9, 8, 1, 20\rangle$, $\langle 15, 48, 6, 35\rangle$, $\langle 45, 8, 5, 30\rangle$ and $\langle 47, 43, 25, 50\rangle$. Note that the more demanding tasks usually have much higher utility rate.

There are four clusters that receives tasks from external environment, which are shaded in Figure 2, and all other clusters receives no tasks from external environment. Task arrivals of each type on each cluster are under some Poisson distribution. We tested two different task loads: *heavy* and *light*. The average number of tasks of each

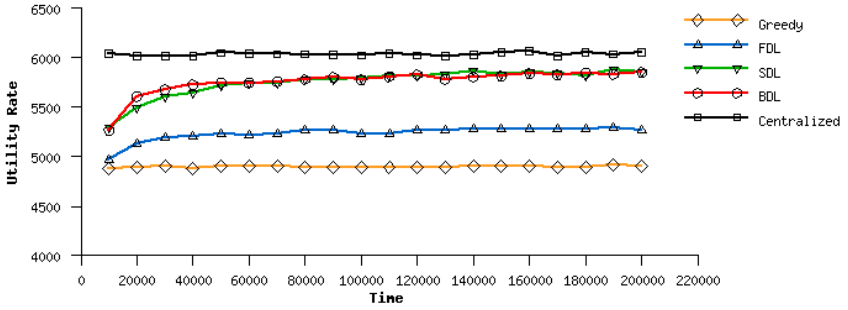

Figure 3: Utility rate under light task load

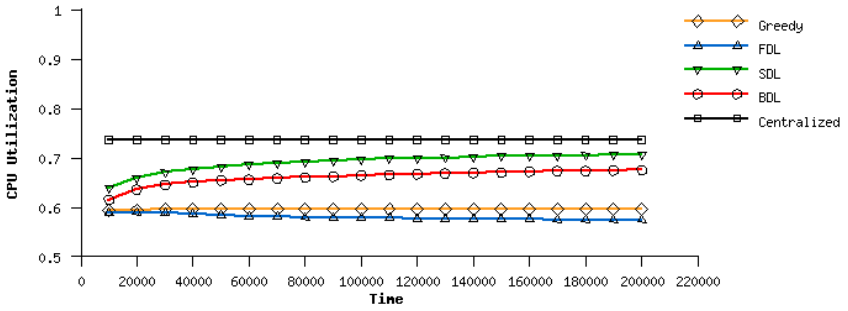

Figure 4: CPU utilization under light task load

type on each node under the heavy load is shown as below:

Node 5 ordinary: 9, IO-intense: 1, compute-intense: 5, demanding: 2

Node 6 ordinary: 7, IO-intense: 1, compute-intense: 4, demanding: 1

Node 9 ordinary: 8, IO-intense: 4, compute-intense: 1, demanding: 2

Node 10 ordinary: 3, IO-intense: 5, compute-intense: 1, demanding: 1

Note that, in overall, the more demanding tasks arrive much less frequently. Under light task loads, these average numbers will be half of those of heavy task loads.

Each computing node can transfer 40 tasks per time step. PGA's learning rate $\zeta = min(0.1, max(2000/(2000+t))$ in Learning$_1$, and $\zeta = min(0.5, max(10000/(10000+t))$ in Learning$_2$, where t is the current simulation time. Each simulation runs with 200000 time steps. All performance measures are computed every 5000 time steps. Results are then averaged over 10 simulation runs and the deviation is computed across the runs.

5.2 Results & Discussions

Figure 3 shows trends of utility rate of the whole cluster network as it runs with different approaches under the light load. The curved lines of FDL, SDL, and BDL demon-

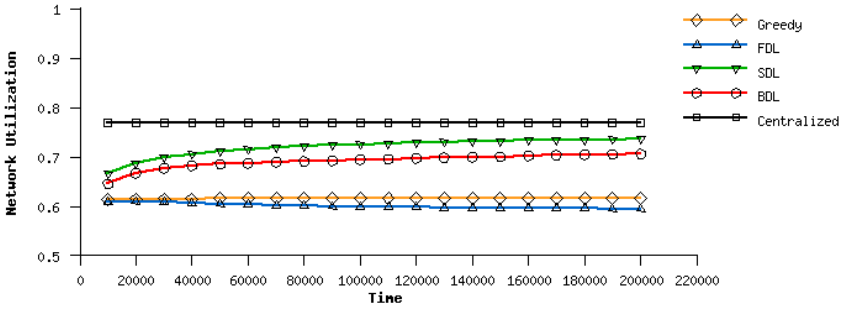

Figure 5: Network utilization under light task load

strate that Learning$_1$, Learning$_2$ and their combination monotonically improve the system performance. Under the light load where the demand for resources is less than the supply, the best solution is to allocate all received tasks within the system. In such a setting, the centralized allocation approach generates the optimal solution. For distributed allocation approaches, how to route tasks and balance the loads across clusters becomes very important. From Figure 3, it can be seen that the performance of SDL and BDL is close to the optimal approach and much better than FDL and greedy approach. So Learning$_2$ for the second decision-making effectively learns distributed task routing policies.

When task loads are well-balanced across clusters, resources of each cluster usually can meet the demand and the best-first algorithm is almost optimal for the first decision-making by greedily allocating tasks locally. However, when task loads are not well distributed, some clusters received more tasks than their capacity. In such a situation, the best-first algorithm will not be optimal, because it does not take account of future tasks in this current decisions. In contrast, Learning$_1$ can predict future task arrivals and learn to give up some tasks with low utilities and reserve resources for future tasks with high utilities. As a result, Learning$_1$ will outperforms the best-first algorithm. This claim is verified by both different performance between FDL and the greedy approach and close performance between SDL and BDL.

Although BDL and SDL perform very well, the gap between them and the optimal approach (the centralized allocation approach under light load) is still noticeable. Two reasons may account for this gap. First, our learning models uses both approximate state space and action space, so both Learning$_1$ and Learning$_2$ may not learn perfect policies. Second, both Learning$_1$ and Learning$_2$ never stop the exploration, although the exploration rate is small. The Learning$_1$'s exploration can reject all received tasks at one time, which wastes resources. The Learning$_2$'s exploration causes some tasks to loop in the network until they expires.

Figure 4 and 5 respectively show CPU and network utilization as the system runs with different approaches under light load. The utilization trends of both CPU and network resources are similar. It can be observed that, although FDL outperforms the greedy approach in terms of utility rate, it has lower resource utilization than the greedy approach. This is because each task type's average ratio of utility rate to resource requirements are not equal. Task type *demanding* has much higher ratio than other types. Table 3 shows the number of abandoned tasks of each type during the last 5000 time period of simulations. We can see that, in order to allocate tasks of type *demanding*

Approaches	Utility Rate	CPU Utilization	Network Utilization	Hops
Greedy	4900 ± 28	0.62 ± 0.00	0.60 ± 0.00	1.80 ± 0.01
FDL	5281 ± 41	0.60 ± 0.00	0.58 ± 0.00	3.88 ± 0.04
SDL	5851 ± 37	0.74 ± 0.00	0.71 ± 0.00	1.50 ± 0.02
BDL	5837 ± 39	0.70 ± 0.00	0.67 ± 0.00	4.30 ± 0.06
Centralized	6038 ± 47	0.77 ± 0.00	0.74 ± 0.00	0.00 ± 0.00

Table 2: Performance with light load

Approaches	Ordinary	IO-Intense	Compute-Intense	Demanding	Total
Greedy	1 ± 0	4889 ± 129	5319 ± 115	3618 ± 73	13828 ± 209
FDL	18797 ± 1351	8542 ± 306	9399 ± 256	515 ± 94	37255 ± 1462
SDL	0 ± 0	367 ± 33	284 ± 27	625 ± 29	1278 ± 59
BDL	11046 ± 854	1867 ± 264	1605 ± 179	165 ± 22	14685 ± 745
Centralized	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0

Table 3: The number of tasks of each type abandoned under light load

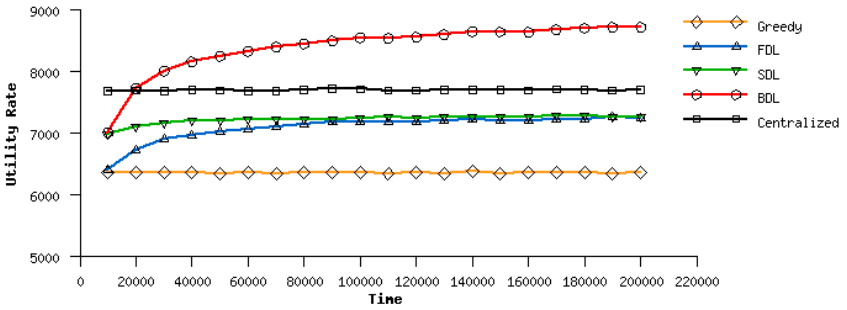

Figure 6: Utility rate under heavy task load

as many as possible, FDL gives up many opportunities to allocate tasks of other types. In contrast, the greedy approach almost allocates all ordinary tasks, which arrives more frequently and have more chance to be allocated than tasks of other types. The same reason also explains that, although SDL and BDL have similar performance, SDL has higher resource utilization than BDL. Additionally, that no tasks are abandoned by the centralized allocation verifies its optimality.

Table 2 summarizes the performance measures of different approaches during the last 5000 time period of simulations. We can see that utility rates of SDL and BDL are 3% less than the optimal one. The column *hops* shows the average number times that a task has been transfered before being allocated. Obviously, tasks in the centralized allocation approach has no hops. The greedy approach and SDL has less hops per task then FDL and BDL. This is because, with the greedy approach and SDL, a cluster manager greedily allocates tasks whenever it has resource available, while, with SDL and BDL, a manager is willing to give up tasks with low utility for future high-utility tasks and forwards tasks more frequently (under heavy load, low-utility tasks arrive much more frequently than high-utility tasks).

Figure 6, 7, and 8 show trends of utility rate, CPU utilization, and network utiliza-

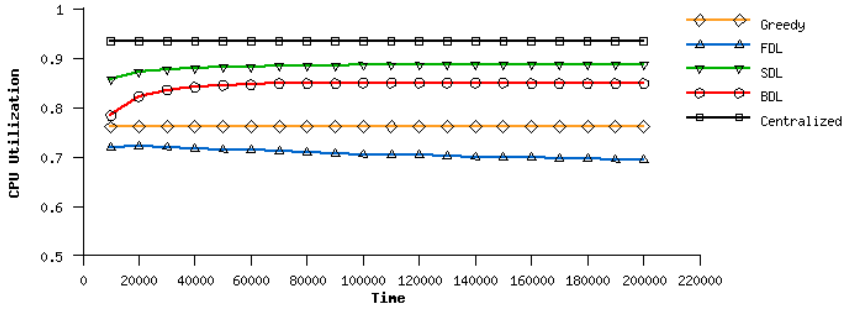

Figure 7: CPU utilization under heavy task load

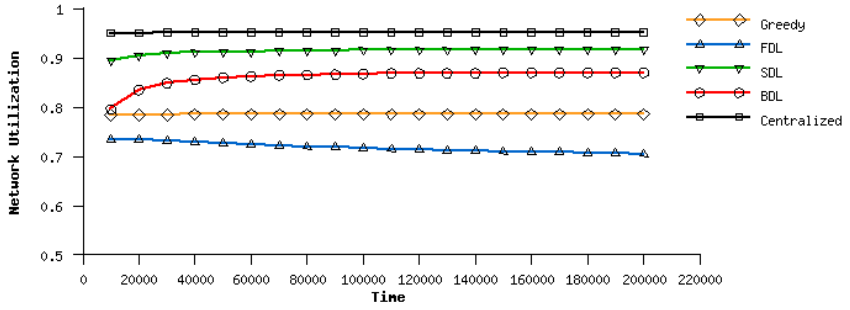

Figure 8: Network utilization under heavy task load

Approaches	Utility Rate	CPU Utilization	Network Utilization	Hops
Greedy	6364 ± 30	0.79 ± 0.00	0.76 ± 0.00	2.31 ± 0.01
FDL	7249 ± 29	0.71 ± 0.00	0.69 ± 0.00	4.33 ± 0.06
SDL	7273 ± 27	0.92 ± 0.00	0.89 ± 0.00	2.13 ± 0.01
BDL	8719 ± 49	0.87 ± 0.00	0.85 ± 0.00	5.33 ± 0.05
Centralized	7700 ± 33	0.95 ± 0.00	0.93 ± 0.00	0.00 ± 0.00

Table 4: Performance with heavy load

tion of the cluster network under the heavy load. Table 4 and 5 show the performance measures and the number of abandoned tasks of each type during the last 5000 time period of simulations. As seen in those figures and tables, most results obtained from the light load case also holds in the heavy load case. However, in this more complicated case, a few additional observations are noted.

The most significant one is that BDL outperforms the centralized allocation approach. Under the heavy load, the overall demand for resources exceeds their supply by the whole cluster network. To address this sequential problem, the optimal centralized allocation approach should take into consideration future task arrivals to make current allocation decisions. Because of short-sight decision-making, our centralized allocation approach is not optimal in this situation. For distributed allocation approach,

Approaches	Ordinary	IO-Intense	Compute-Intense	Demanding	Total
Greedy	650 ± 52	28058 ± 230	29444 ± 293	16437 ± 147	74590 ± 453
FDL	69151 ± 2480	37947 ± 434	41159 ± 374	7854 ± 261	156112 ± 2823
SDL	28 ± 6	20379 ± 291	20876 ± 246	15406 ± 145	56691 ± 495
BDL	60319 ± 3237	31095 ± 524	33421 ± 617	4381 ± 341	129218 ± 3580
Centralized	276 ± 39	18662 ± 273	20839 ± 258	13943 ± 159	53722 ± 407

Table 5: The number of tasks of each type abandoned under heavy load

its local allocation policies are getting important for the system performance, at least as important as its task routing policies. This point is verified by the similar performance of FDL and SDL (both better than the greedy approach) under the heavy load, as shown in Figure 6. By comparing the performance of FDL and the greedy approach, we see that Learning$_1$ obtains better policies than the best-first algorithm when the resource demand exceeds the supply. This advantage of Learning$_1$ can offset disadvantages of partial information and control of distributed allocation approaches. Therefore, although neither FDL nor SDL alone can perform as well as the centralized allocation approach, with advantages of both FDL and SDL, BDL can perform better than the centralized allocation approach.

Another observation is that the performance improvement from BDL can be greater than the sum of those from FDL and SDL. From Table 4, we can calculate the difference between FDL and the greedy approach is $7249 - 6364 = 885$ and the difference between SDL and the greedy approach is $7273 - 6364 = 909$. So their total is 1794. But the difference between BDL and the greedy approach is $8719 - 6364 = 2355 > 1794$. This is because BDL establishes a connection between Learning$_1$ and Learning$_2$ with Equation 6. Learning$_1$ improves local allocation policies, which generates better demands for certain tasks. Those better demands boost the performance of Learning$_2$, which in turn learns better task routing policies to deliver tasks to the right clusters.

6 Related Work

Several techniques for managing resources in shared clusters have been developed over the past decade [5, 4, 20]. A key contribution of our approach is to enable resource sharing across shared clusters in a distributed way.

Several distributed scheduling algorithms based on heuristics are developed for allocating tasks with deadlines and resource requirements in [15]. These algorithms are dynamic and function in a decentralized manner. Although they are originally targeted for sharing resources among computing nodes, they can easily extended for sharing resources among clusters. However, unlike our approach where each agent only interacts with a limited number of neighboring agents, both their basis algorithms, *focused address algorithm* and *bidding algorithm*, assume each agent can interact with all other agents and request resource information from them in a real-time manner. As a result, these algorithms have potential scalability issues.

A semi-distributed load balancing approach for massively parallel multicomputer systems has been proposed in [3]. The proposed strategy uses a two-level hierarchical control by partitioning a distributed multiprocessor system into multiple spheres (a similar concept to *cluster*). The runtime model is similar to our approach. For each task, a sphere may make two decisions on: whether to allocate it locally and which

sphere for transfer the task to. However, each sphere uses simple load thresholds to make such decisions and assume each sphere can request load information from all other sphere. Multi-agent reinforce learning algorithms have been proposed for addressing a distributed task allocation problem (DTAP) [2, 22]. Unlike our allocation model, all computing nodes in this DTAP are homogeneous and tasks have no deadline and resource requirements.

A different resource allocation model is formulated in [17], which assumes a strict separation between agents and resources. Jobs arrive to agents who make decisions about where to execute them and the resources are passive (i.e., do not make decisions). Therefore, there is no direct interaction between agents. Agents use reinforcement learning based on local observations to adapt to changing resource availability (because of other agents' actions). Work [12] extends this model to allow heterogeneity of tasks and resources and attach a queue to each resource.

Much work on resource allocation and load balancing have been done within the framework of game dynamics which is closely related to multi-agent reinforcement learning [11]. Congestion games [16] and minority games [6, 9] are two examples of applying game dynamics to the resource allocation problem. The assumption of their games usually eliminates or reduces complex issue (e.g., learning efficiency due to state-action space, scalability with the increasing number of agents) of real problems, which may be key challenges to solving those problems.

7 Conclusion and Future Work

In this paper, we formulate resource sharing among clusters as a distributed sequential resource allocation problem (DSRAP). We also present a multi-agent approach to addressing this problem, where all agents cooperatively allocate task to maximize the global utility of the system. In our approach, each agent's decision is decomposed into two subproblems: *local task allocation* and *task routing*, and then use multi-agent reinforcement learning to learn a decision policy for each subproblem, only based on its local observations. Experimental results show that, comparing to the performance of a centralized resource allocation approach, our distributed approach works very effectively for addressing DSRAP. Although the simulation described here are not fully realistic from the standpoint of actual networks of shared clusters, we believe this paper has shown multi-agent reinforcement learning is a promising approach to resource management across shared clusters.

One of interesting directions for future work is to generalize the table-based value functions and policies with function approximation. This generalization allows us to refine both action and state spaces to better approximate their original spaces or directly use their original spaces. Potentially, less exploration is needed for the learning because the generalization can transfer knowledge of well-learned states to their neighboring states. Another direction is to incorporate MASPA [22] into our approach to speed up the learning with a large policy search space. We also plan to extends our approach to handle situations, such as dynamically changing resource capacity of clusters and temporary failure of clusters.

References

[1] Sherief Abdallah and Victor Lesser. Learning the task allocation game. In *AAMAS'06*, 2006.

[2] Sherief Abdallah and Victor Lesser. Multiagent reinforcement learning and self-organization in a network of agents. In *AAMAS'07*, 2007.

[3] Ishfaq Ahmad and Arif Ghafoor. Semi-distributed load balancing for massively parallel multicomputer systems. *IEEE Trans. Softw. Eng.*, 17(10):987–1004, 1991.

[4] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Measurement and Modeling of Computer Systems*, pages 90–101, 2000.

[5] Andrea C. Arpaci-dusseau and David E. Culler. Extending proportional-share scheduling to a network of workstations. In *Proceedings of Parallel and Distributed Processing Techniques and Applications*, 1997.

[6] W Brian Arthur. Inductive reasoning and bounded rationality. *American Economic Review*, 84(2):406–11, May 1994.

[7] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *In Proceedings of the third Symposium on Operating System Design and Implementation (OSDI'99)*, pages 45–58, 1999.

[8] Michael Bowling. Convergence and no-regret in multiagent learning. In *NIPS'05*, pages 209–216, 2005.

[9] Damien Challet and Matteo Marsili. Phase transition and symmetry breaking in the minority game. *Physical Review Letters*, 1999.

[10] Caroline Claus and Craig Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. In *AAAI'98*, pages 746–752. AAAI Press, 1998.

[11] James P. Crutchfield and Yuzuru Sato. Coupled replicator equations for the dynamics of learning in multiagent systems. *Physical Review Letters*, 2002.

[12] Aram Galstyan, Karl Czajkowski, and Kristina Lerman. Resource allocation in the grid using reinforcement learning. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1314–1315, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Spiros Kapetanakis and Daniel Kudenko. Reinforcement learning of coordination in cooperative multi-agent systems. In *AAAI'02*, pages 326–331, 2002.

[14] Shailesh Kumar and Risto Miikkulainen. Confidence based dual reinforcement q-routing: An adaptive online network routing algorithm. In *IJCAI '99*, pages 758–763, 1999.

[15] K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Trans. Comput.*, 38(8):1110–1123, 1989.

[16] Robert W. Rosenthal. A class of games possessing pure-strategy nash equilibria. *International Journal of Game Theory*, pages 65–67, 2005.

[17] Andrea Schaerf, Yoav Shoham, and Moshe Tennenholtz. Adaptive load balancing: A study in multi-agent learning. *Journal of Artificial Intelligence Research*, 2:475–500, 1995.

[18] Satinder P. Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvari. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308, 2000.

[19] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[20] Bhuvan Urgaonkar and Prashant Shenoy. Sharc: Managing cpu and network bandwidth in shared clusters. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 14(11), 2003.

[21] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3/4):279–292, 1992.

[22] Chongjie Zhang, Sherief Abdallah, and Victor Lesser. MASPA: Multi-agent automated supervisory policy adaptation. In *University of Massachusetts Amherst Computer Science Technical Report #08-03*, 2008.

[23] Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *ICML'03*, pages 928–936, 2003.