A Multi-Agent Learning Approach to Online Distributed Resource Allocation

Chongjie Zhang
Computer Science Department
University of Massachusetts
Amherst, MA 01003 USA
chongjie@cs.umass.edu

Victor Lesser
Computer Science Department
University of Massachusetts
Amherst, MA 01003 USA
lesser@cs.umass.edu

Prashant Shenoy
Computer Science Department
University of Massachusetts
Amherst, MA 01003 USA
shenoy@cs.umass.edu

Abstract

Resource allocation in computing clusters is traditionally centralized, which limits the cluster scale. Effective resource allocation in a network of computing clusters may enable building larger computing infrastructures. We consider this problem as a novel application for multiagent learning (MAL). We propose a MAL algorithm and apply it for optimizing online resource allocation in cluster networks. The learning is distributed to each cluster, using local information only and without access to the global system reward. Experimental results are encouraging: our multiagent learning approach performs reasonably well, compared to an optimal solution, and better than a centralized myopic allocation approach in some cases.

1 Introduction

As "Software as a service" becomes a popular business model, it is becoming increasingly difficult to build large computing infrastructures that can host effectively the wide spread use of such 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. Resource management approaches developed for shared clusters [Arpaci-dusseau and Culler, 1997; Aron *et al.*, 2000; Urgaonkar and Shenoy, 2003] are centralized, which limits the cluster scale.

To build larger shared computing infrastructures, one common model is to organize a set of shared clusters into a network and enable resource sharing across shared clusters. 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, a 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 as 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 [Zinkevich, 2003], 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 for this particular problem, it is also 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 defines DSRAP. Section 3 introduces the Fair Action Learner algorithm. Section 4 presents decision-making processes of each agent and learning models for both decisions. Section 5 describe experiment design and analyzes experimental results. Related work is presented in Section 6. Finally, Section 7 concludes our work.

2 Problem Description

The runtime model of DSRAP is described as follows. Each agent receives tasks from either the external environment or a neighbor. At each time step, an agent makes decisions on what tasks are allocated locally and to which neighbors the

tasks not allocated locally should be forwarded. Due to the task transfer time cost, there is communication delay between two agents. To reduce the communication overhead, the number of tasks an agent can transfer at each time step is limited. To allocated a task, an agent should have available resources to satisfy its resource requirements. When a task is allocated locally, the agent gains 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 completes, all resources it occupies will be freed and available for future tasks. The main goal of DSRAP is to derive decision policies for each agent 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 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.

A task type characterizes a set of tasks. A task type t_i is also denoted as a tuple $\langle D_i^s, D_i^u, D_i^w, D_i^{d_1}, \ldots, D_i^{d_q} \rangle$, 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
- $D_i^{d_j}$ is the demand distribution of resource j of a task.

A task is denoted as a tuple $\langle t, u, w, d_1, \ldots, d_q \rangle$, where

- t is the task type.
- u is the utility rate of the task.
- w is the maximum waiting time before being allocated.
- d_i is the demand of resource $i = 1, \ldots, q$.

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

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

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 Fair Action Learning Algorithm

In the single-agent setting, reinforcement learning algorithms, such as Q-learning, learn optimal value functions and optimal policies in MDP environments when lookup tables are used to represent the state-action value function. However, in the multiagent 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 [Claus and Boutilier, 1998]. As a result, the learning of agents may diverge due to lack of synchronization. Several multiagent reinforcement learning (MARL) algorithms have been developed to address this issue [Zinkevich, 2003; Bowling, 2005], with convergence guarantee in specific classes of games with two agents.

Algorithm 1: Fair Action Learning (FAL) Algorithm

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

To address DSRAP, we propose a multiagent reinforcement learning algorithm, called Fair Action Learning (FAL). The FAL algorithm is a direct policy search technique and a variant of the GIGA algorithm [Zinkevich, 2003]. For many practical problems, the value function on policies is usually not known, so the policy gradient for GIGA can not be directly calculated. To deal with this issue, the FAL algorithm 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 1 describes its policy update rule, where $\pi(s,a)$ is the probability of taking action a in state s under policy π, $\pi(s)$ is the distribution over all actions in state s, and ζ is the policy learning rate. The *limit* function from GIGA is applied to normalize $\pi(s)$ such that it sums to 1.

FAL learns stochastic policies. As argued in [Singh *et al.*, 2000], stochastic policies can work better than deterministic policies in partially observable environments (e.g., DSRAP), if both are limited to act based on the current percept. To improve the expected value for each state, FAL will increase the probability of actions that receive an expected reward above the current average. Therefore FAL will converge to a policy where, for each state, all actions receive the same expected reward and are fairly treated. (It is possible that FAL converges to a deterministic policy when an action is always more favorable than other actions). In a multiagent setting, this property will help agents to converge to an equilibrium.

4 Learning Distributed Resource Allocation

Algorithm 2 shows the general decision-making process of each agent, which repeats at each time step. This algorithm

Algorithm 2: General Decision-Making Algorithm

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

uses two functions: *selectAndAllocate* and *chooseANeighborAndForward*. The first function selects and allocates a subset of received tasks to its local cluster to maximize its local utility. As the global utility is the sum of all local utilities, optimizing this function can potentially improve the system performance. The second function chooses a neighbor and forwards an unallocated task to maximize the allocation probability of the task. This function aims to route tasks to unsaturated agents and balance the task load in the system.

4.1 Local Allocation Decision

Algorithm 3: *selectAndAllocate*(TASKS)

```
begin
    ALLOCABLE ← getAllocable(TASKS);
    ALLOCATED ← ∅ ;
    while ALLOCABLE ≠ ∅ do
        ALLOCABLE ← ALLOCABLE ∪ {VOID} ;
        update current state s;
        t ← task selected based on policy π1(s, ·);
        if t = VOID then
            ALLOCABLE ← ∅;
        else
            allocate(t);
            ALLOCATED ← ALLOCATED ∪{t} ;
            TASKS ← TASKS \{t} ;
            ALLOCABLE ← getAllocable(TASKS);
            learn(s, t);
        end
    end
    return ALLOCATED;
end
```

Algorithm 3 shows the local allocation decision-making process. This algorithm incrementally selects and allocate tasks locally. It uses three functions: *getAllocable*, *allocate*, and *learn*. Function *getAllocable* 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 the local allocation policy. *VOID* is a unique, fake task with no resource requirements and zero utility rate. Selecting this task indicates that the process of selecting a subset of tasks to be allocated locally is finished.

Now we define the state space, the action space, and the reward function for learning this decision policy. A decision state $s = \langle s_t, s_c \rangle$ consists of two feature vectors s_t and s_c, describing the task set to be allocated and availability of various resources in a cluster respectively. As the type of a task approximately represents information about the task, we use task types to characterize the task set to be allocated. The feature vector $s_t = \langle y_1, y_2, \ldots, y_m \rangle$, where each feature y_i corresponds to task type i and m is the number of task types. If the task set contains a task with type i, then $y_i = 1$. To represent s_c, we first categorize availability of each resource into multiple levels and then use combinations of levels of different resources as features. The value of a feature is the number of computing nodes in the cluster that have corresponding availability level for each resource.

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 tasks that are exactly the same. To reduce the action space, the type of the task is used to approximately represent the task itself. Therefore, the action set is mapped to the set of task types. Then the 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 a task type is selected, the task of this type with the greatest utility rate will selected and allocated. The reward for allocating task t is the utility rate associated with t.

An agent receives tasks from both the external environment and its neighbors. Thus, other agents' decision policies affect task arrivals at the agent. As all agents concurrently learn their policies, the learning environment of each agent becomes non-stationary. We use FAL algorithm to learn local allocation decision policies $\pi_1(s, a)$. As $\pi_1(s, a)$ is stochastic, the following rule is used to update Q-value function:

$$Q(s_n, a_n) \leftarrow (1-\alpha)Q(s_n, a_n) + \alpha[r_n + \gamma \textstyle\sum_a \pi(s_{n+1}, a)Q(s_{n+1}, a)]$$

This new update rule is just like that of Q-learning except that instead of the maximum over next state-action pair it uses the expected value under the current policy.

Accelerating the Learning Process

Even when using the 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 reinforcement learning exploration strategies could entail running arbitrarily poor initial policies, which should be avoided in the practical system. To address those issues, we proposed several heuristics to speed up learning. Policies are initialized with a greedy allocation algorithm, which allocates all tasks in an decreasing order of their utilities if resources permit. The learning is online and the ϵ-greedy strategy is used to ensure that each action will 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, which will not reject tasks if resources permit. In addition, rejecting too many tasks will degrade the system performance and thus we also limit the exploration rate of selecting *VOID* task.

4.2 Task Routing Decision

Task routing addresses the question: to which neighbor should an agent forward an unallocated task to get it to a unsaturated cluster before it expires? As each agent interacts with a limited number of neighbors, it may not know where are unsaturated clusters that can be multiple hops away from it. An agent can learn to route tasks via interacting with its neighbors. The learning objective for task routing is to maximize the probability of each task to be allocated in the system.

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. An action j corresponds to choosing neighbor j for forwarding a task. The value function $Q_i(s_x, j)$ returns the expected probability that the task x will be allocated if an agent i forwards it to its neighbor j.

Upon sending a task to agent j, agent i immediately gets the reward signal $r(s_x, j)$ from agent j. 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's policies for both local allocation and task routing:

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

where $p_j(x)$ is the probability that agent j will allocate task x locally, π_{2j} is the task routing policy of agent j, and s'_x is the state where agent j makes a 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 for transferring a task between agent i and j.

The probability $p_j(x)$ depends on agent j's policy π_{1j}:

$$p_j(x) = \sum_{s_t} q(\langle s_c, s_t \rangle | t) \pi_{1j}(\langle s_c, s_t \rangle, t)$$

where t is the type of task x, 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 local allocation policy of agent j. The probability $q(\langle s_c, s_t \rangle | t)$ can be directly estimated during the learning.

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

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

where α is a learning rate (0.5 in our experiments). With modified Q-value function, the FAL algorithm updates the task routing policy π_{2i}.

To speed up the learning, we use an idea, called *backward exploration* [Kumar and Miikkulainen, 1999], of using information about the traversed path for exploration in the reverse direction. When agent i transfer task x to its neighbor j, the message that contains pass x can take along reward information $r(s_x, i)$ of agent i for allocating x. This reward information can be used by agent j to update its own estimate 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.

	Greedy	FDL	SDL	BDL
Local	Best-first	Learning_1	Best-first	Learning_1
Routing	Random	Random	Learning_2	Learning_2

Table 1: Distributed resource allocation approaches

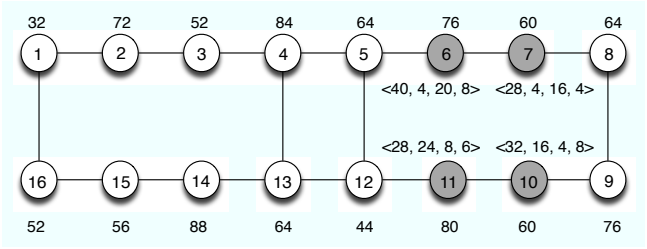

Figure 1: The network with 16 clusters and 1024 nodes

5 Experiments

5.1 Experiment Design

To evaluate the performance of learning models developed above, we compared five resource allocation approaches: *greedy allocation*, *first-decision (local allocation) learning* (FDL), *second-decision (task routing) learning* (SDL), *both-decision learning* (BDL), and *centralized allocation*. The first four approaches are distributed techniques. As shown in Table 1, they use different algorithms for each decision-making. The *best-first* algorithm, at each time step, first sorts all received tasks in a descending order of utility rate and then uses the best-fit algorithm in Sharc [Urgaonkar and Shenoy, 2003] to allocate tasks one by one. *Learning*$_1$ and *Learning*$_2$ respectively refer to the learning algorithms we developed for local allocation and task routing. The *random* algorithm for task routing picks a random neighbor 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 directly allocate tasks to resources without any routing.

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 and total 1024 nodes, as pictured in Figure 1. Each cluster uses Sharc to manage its local resources. The number outside a circle represents the number of computing nodes of that cluster. The CPU capacity and network capacity vary on different computing nodes, whose range is in $[50, 150]$.

We use four task types: *ordinary*, *compute-intensive*, *IO-intensive*, and *demanding*. Their feature vectors are respectively $\langle 20, 1, 9, 8 \rangle$, $\langle 30, 5, 45, 8 \rangle$, $\langle 35, 6, 15, 48 \rangle$ and $\langle 50, 25, 47, 43 \rangle$, each of which shows the mean of service time, utility rate, CPU demand, and network demand. All tasks have waiting time $w = 10$. The service time is under exponential distribution and the rest is under Poisson distribution. Note that the more demanding tasks usually have much higher utility rates.

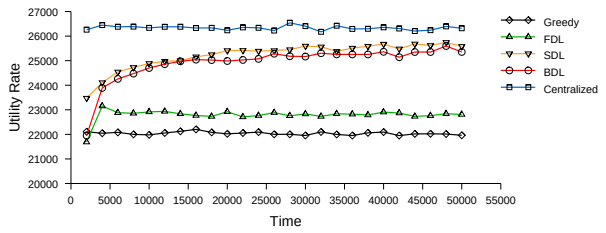

Figure 2: Utility rate under light task load

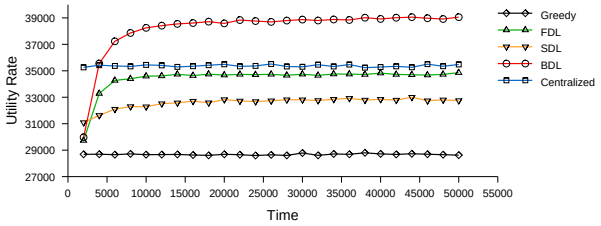

Figure 3: Utility rate under heavy task load

Only four clusters, shaded in Figure 1, receive tasks from external environment. We tested two different task loads: *heavy* and *light*. The vector besides each shaded node shows, under heavy load, the average number of tasks of four types arriving on that node. Under light task loads, these average numbers are half of those of heavy task loads. Task arrivals of each type on each cluster are under a Poisson distribution. Cluster communication limit is 300 tasks per time step.

In our experiments, availability of each resource is categorized into three levels. All performance measures shown below are computed every 2000 time steps. Results are then averaged over 10 simulation runs and the deviation is computed across the runs.

5.2 Results & Discussions

Figure 2 shows utility rate trends of the whole cluster network as it runs with different approaches in a lightly loaded environment. The curved lines of FDL, SDL, and BDL demonstrate that local allocation learning, task routing learning and their combination gradually improve system performance. Under 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 2, it can be seen that the performance of SDL and BDL is close to the optimal approach and much better than FDL and the greedy approach. So the learning for task routing policy works effectively.

When task loads are well-balanced across clusters, resources of each cluster usually can meet tasks' demand and the best-first algorithm is almost optimal for local allocation decisions. In some sense, the similar performance between SDL and BDL verifies the effectiveness of learning local allocation policies. However, when task loads are not well distributed across the clusters, 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 into account future task arrival patterns in its current decisions. In contrast, the learning approach implicitly estimates future task arrival patterns and give up some tasks with low utilities and reserve resources for future tasks with high utilities. Therefore, FDL will outperforms the greedy algorithm.

Figure 3 show utility rate trends of the cluster network under the heavy load. Most analysis results for the lightly loaded case also holds in the heavily loaded case. In this more complicated case, one significant observation 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. Without considering future task arrivals, the best-first centralized allocation approach is not optimal in such a situation. On the other hand, the learning approach implicitly takes account of future tasks for making current decisions and can work better than the best-first algorithm, which is verified by the performance of FDL and the greedy approach. Combined with effective learned routing policies, the advantage of learning local allocation offsets disadvantages due to partial information and distributed resource control in distributed approaches, which allows BDL to performs better than the centralized allocation approach.

Approaches	Utility	CPU	Network	Hops
Greedy	21996 ± 173	0.66 ± 0.00	0.64 ± 0.00	1.57 ± 0.01
FDL	22840 ± 152	0.61 ± 0.00	0.59 ± 0.00	4.24 ± 0.02
SDL	25661 ± 171	0.80 ± 0.00	0.77 ± 0.00	1.66 ± 0.02
BDL	25415 ± 139	0.74 ± 0.00	0.71 ± 0.00	5.87 ± 0.04
Centralized	26302 ± 216	0.84 ± 0.00	0.81 ± 0.00	0.00 ± 0.00

Table 2: Performance with light load

Approaches	Utility	CPU	Network	Hops
Greedy	28656 ± 125	0.83 ± 0.00	0.81 ± 0.00	2.41 ± 0.03
FDL	34734 ± 107	0.77 ± 0.00	0.74 ± 0.00	5.07 ± 0.02
SDL	32785 ± 143	0.95 ± 0.00	0.92 ± 0.00	2.05 ± 0.01
BDL	39000 ± 136	0.88 ± 0.00	0.84 ± 0.00	5.19 ± 0.03
Centralized	35460 ± 171	0.97 ± 0.00	0.96 ± 0.00	0.00 ± 0.00

Table 3: Performance with heavy load

Table 2 and 3 respectively summarize the performance measures (including utility rates, CPU utilization, network utilization, and task routing hops) of different approaches under light and heavy load during the last 2000 time period of simulations. Under light load, although BDL performs very well in a distributed way, the difference between its utility rate and the optimal one (generated by the centralized approach) is still noticeable, which is around 3%. Several factors contribute to this gap. First, due to partial observation, distributed learned routing policies can not be perfect. In addition, the communication of each agent is limited. As a result, some tasks are not allocated before their deadline. Second, to reduce the policy search space, both learning models use both approximate state space and action space, which introduces further uncertainty that has the effect of decreased performance. We tested more accurate models, such as discretizing

availability of each resource into more levels and using more task features in addition to the type task to represent actions. Although experiment results are slightly better, the learning converges much slower and has poor policies for a long period. Third, the learning never stops its exploration.

Note that BDL has both lower CPU and network utilization than SDL, although it performs better. This is because, with a learned local allocation policy, an agent is willing to give up tasks with low utility and reserve resources for future high-utility tasks, which causes resources to be idle for a higher percentage of the time. This giving-up behavior causes more tasks to be routed, which explains that the greedy approach and SDL have less hops per task than both FDL and BDL. The *hops* describes the average number times that a task has been transfered before being allocated.

Parameters of our heuristics for speeding up learning are set in the same way: utilization threshold $\beta = 0.7$ and minimum random exploration rate $\epsilon = 0.005$. We observe that when task arrival rate becomes higher, properly improving β and decreasing ϵ can improve the system performance.

6 Related Work

Several distributed scheduling algorithms based on heuristics are developed for allocating tasks with deadlines and resource requirements in [Ramamritham *et al.*, 1989]. Unlike our approach, 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 different resource allocation model is formulated in [Schaerf *et al.*, 1995], which assumes a strict separation between agents and resources. Jobs arrive at agents who use reinforcement learning to make decisions about where to execute them and the resources are passive (i.e., do not make decisions) and dedicated. Therefore, there is no direct interaction between agents. The work in [Tesauro, 2005] has a similar model, but there is a resource arbiter who dynamically decides resource allocation based on value functions of agents, which are learned independently.

Reinforcement learning has been applied to network routing [Boyan and Littman, 1994; Kumar and Miikkulainen, 1999]. In their problems, each package has a pre-specified destination, so the routing is targeted. In contrast, in our problem, agents do not know the destination for an task, which is supposed to be learned. In addition, our task routing learning is also affected by the local allocation learning.

7 Conclusion

The empirical results presented in this paper provide evidence that multiagent learning is a promising and practical method for online resource allocation in real computing infrastructures with a network of shared clusters. Compared with a single global learning, multiagent learning scales up to many applications by partitioning state and action spaces over agents and through concurrent learning over more computational hardware. This work also plausibly suggests that multiagent learning may be an approach to address online optimization problems in distributed systems, such as large-scale grid computing, sensor networks, and peer-to-peer information retrieval.

References

[Aron *et al.*, 2000] 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.

[Arpaci-dusseau and Culler, 1997] 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.

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

[Boyan and Littman, 1994] Justin A. Boyan and Michael L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In *NIPS'94*, volume 6, pages 671–678, 1994.

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

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

[Ramamritham *et al.*, 1989] 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.

[Schaerf *et al.*, 1995] 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.

[Singh *et al.*, 2000] 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.

[Tesauro, 2005] Gerald Tesauro. Online resource allocation using decompositional reinforcement learning. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 886–891. AAAI Press / The MIT Press, 2005.

[Urgaonkar and Shenoy, 2003] 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.

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