ID#727: Learning Task Allocation via Multi-level Policy Gradient Algorithm with Dynamic Learning Rate

Sherief Abdallah

Victor Lesser
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003
{shario,lesser}@aaai.org

April 12, 2005

Abstract

Task allocation is the process of assigning tasks to appropriate resources. To achieve scalability, it is common to use a network of agents (also called mediators) that handles task allocation. This work proposes a novel multi-level policy gradient algorithm to solve the local decision problem at each mediator agent. The higher level policy stochastically chooses a task decomposition. The lower level policy assigns subtasks to neighboring agents also stochastically. Agents learn autonomously, cooperatively, and concurrently to increase system performance. No state information is used except for the task being allocated. Furthermore, the algorithm dynamically adjusts the learning rate, to speed up convergence, using the ratio of action values. Experimental results show how our proposed solution outperforms other deterministic approaches by balancing the load over resources and converging faster to better policies.

1 Introduction

Task allocation is the process of assigning tasks to appropriate resources. The problem appears in many real applications like the Grid, web services, sensor nets and other domains[**?**]. Consider the web services as an example. In that domain there are servers distributed over the web. Each server provides a set of services for applications. Users may appear anywhere in the web asking for a composition of services (also called a task) that requires more than one server. Any server can work on more than one task at a time. However, the cost of executing a task increases in proportion to the total number of tasks being serviced by the server. Since users usually do not know where servers are, a network of agents (also called mediators) that know about different servers is

used. Such agents take requests from users and reply to users with the appropriate set of servers.

This work illustrates how agents in such a network can learn to work cooperatively in order to optimize the task allocation problem. In particular, this work proposes a novel multi-level policy gradient algorithm to optimize the local decision of each agent in the network. The higher level policy stochastically chooses a task decomposition. The lower level policy stochastically assigns subtasks to neighboring agents. Agents learn autonomously, cooperatively, and concurrently to minimize the cost of executing tasks. The algorithm does not use any state information except the type of the task being allocated and estimates of the cost for assigning task types to neighbors. Furthermore, to speed up convergence, our proposed algorithm dynamically changes the learning rate in proportion to the cost of choosing an action (whether this action is choosing a decomposition or assigning a task to a neighbor).

Two factors make the problem both interesting and challenging: the limited local view of each agent and the need for load balancing. In large distributed systems, having a global view of the system's state is impossible from practical point of view. Agents usually rely on their limited local view and use communication to augment this view. This is a trade-off between optimality and scalability. In our system, the only a *priori* knowledge known by an agent (as will be described shortly) is the addresses of its direct neighbors. Furthermore, agents do not communicate their states, but rely solely on the statistical outcomes of interacting with neighboring agents. In other words, agent A's knowledge about its neighbor agent B is summarized via a statistical average of previous outcomes when A tried assigning a task to B.

What makes load balancing needed in many real systems is the nonlinear increase of task execution cost with respect to the increase in load. Cost here is a signal of the system's performance. For example, cost may increase due to an increase in task waiting time to indicate a reduction in users' satisfaction. Therefore, in real systems, it is almost always better to divide the load as fairly as possible among servers and resources. The algorithm presented in this paper aims at balancing the load over servers/resources. Experimental results show how our algorithm significantly outperforms deterministic approaches that ignore load balancing.

The paper is organized as follows. The rest of this section presents a motivating example. Then a formal problem definition is presented, followed by a description of local agent decision problem. Next a description of our algorithm that optimizes the local agent decision is given. Experimental results are then presented and discussed, showing how our algorithm outperforms other deterministic approaches. Then a discussion of related work is given. We finally conclude and lay out our future directions.

1.1 Motivating Example

To get a better insight into the complexity of the decision making of each agent, consider the example in Figure 1. This system consists of three agents, MA, MD, and MF. Each agent is connected to two resources. There are two main types of resources, A and B. Resource A_i is of type A and can undertake only task type TA. Similarly, resource B_i is of type B and can only undertake task type TB. Resources A_f and B_f are of types A and B respectively but they are fast resources that need half the time of other

resources to finish their tasks. Task TAB is a more complex task that has three alternative decompositions: $\{TA, TA\}$, $\{TB, TB\}$ and $\{TA, TB\}$. However, only agent MD knows how to decompose task TAB.

Suppose agent MA receives many tasks of type TA. If MA always chooses $A1$ to assign TA to (i.e. deterministic policy), then $A1$ will be overloaded and its cost rapidly increases. After several trials, MA will not see $A1$ as appealing and will switch its policy to another neighbor. As the other neighbor gets overloaded MA will switch again and so on. This means that MA will not converge on a deterministic policy and will keep oscillating after spikes of high costs due to overloading. One would expect a stochastic policy, where MA chooses each neighbor with a certain probability, would perform better. Similarly, suppose agent MD receives a task of type TAB. MD then needs to choose among the three possible decompositions of TAB. Each decomposition imposes certain load patterns on the system. For example, always choosing the decomposition $\{TA, TB\}$ means there will be equal load on both resource types A and B.

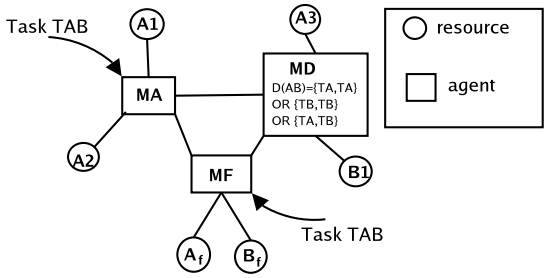


Figure 1: A network of agents that are responsible for assigning resources to incoming tasks.

2 Problem Definition

Let $T = \{T_1, ..., T_{|T|}\}$ be the set of task types. Different instances of tasks types appear randomly at different agents. Each task instance I_j is defined by an arrival time t_{I_j}, a task type $T_{I_j} \in T$, and a payoff O_{I_j}. A decomposition function $D(T_i) = \{d_1, ..., d_{|D(T_i)|}\}$ associates with each task a set of decompositions, where $d_i \subseteq T$ (hence $D(T_i)$ provides alternative ways to do task T_i). The system has a set of resources $R = \{R_1, ..., R_{|R|}\}$. A resource R_i can undertake a set of task types $H_{R_i} \subseteq T$. $\forall T_j \in H_{R_i} : F_{R_i}(T_j) > 0$ is the time resource R_i needs to finish a task of type T_j. The cost of executing a task T_j at resource R_i at time t is $C^t(T_j, R_i)$. The cost is time-dependent because it depends on the total load on resource R_i at that time. The goal is to optimize the allocation of tasks to appropriate resources such that net profit (which is payoff reduced by total cost) over period of time Δ is maximized. More formally, the global system goal is to maximize the objective function Γ defined as follows.

$$\Gamma = \sum_{I:t_I \in \Delta} O_I - \sum_{\langle T_i, R_j \rangle \in A} C^{t_I}(T_i, R_j)$$

where $A = \{a_1, ..., a_{|A|}\}$ is a set of task-resource assignments, where $a_i = \langle T_i, R_i \rangle$. However, because there is not any centralized entity that has a global view of the whole system, evaluating and optimizing Γ is practically impossible. Instead, one needs a local objective function Γ_{M_x} that each agent M_x attempts to optimize. Let $M = \{M_1, ..., M_{|M|}\}$ be the set of agents interconnecting the set of resources R. Each agent M_x has a set of neighbors $N(M_x) \subseteq M \cup R$. Each agent knows of a set of decompositions D_{M_x}, where $D_{M_x}(T_j) \subseteq D(T_j)$. The goal of each agent M_x is to allocate incoming task instance $I = \langle t_I, T_I, O_I \rangle$ to neighboring agents such that $\Gamma_{M_x}(I)$ is maximized, where

$$\Gamma_{M_x}(I) = O_I - \sum_{\langle T_i, n_j \rangle \in A_{M_x}(I,d)} C^{t_I}(T_i, n_j)$$

where $A_{M_x}(I, d) = \{a_k : a_k = \langle T_i, n_j \rangle\}$ is a set of task-neighbor assignments, where $n_j \in N(M_x)$ and $T_i \in d \in D_{M_x}(T_I)$. $C^{t_I}(T_i, n_j)$ is the cost of assigning task T_i to neighbor n_j. In other words, agent M_i needs to find both a decomposition d and an assignment of neighbors to each of the subtask types in d such that the total *estimated* cost of executing I is minimized. The cost $C^{t_I}(T_i, n_j)$ is only an estimation because it depends on how agent n_j will conduct the allocation of T_i. For example, if n_j is still learning then it is likely that the cost will be higher than the real cost (e.g. because n_j is allocating T_i poorly). As agents interact with each other, one would hope that the local agent policies converge to good (if not optimal) collective policy. Therefore, the local objective function at each agent Γ_{M_i} only approximates the global objective function Γ. However, as the results in this paper show, using our algorithm agents successfully converge and learn cooperate in allocating tasks. The following section presents our algorithm

3 Multi-level policy gradient algorithm

An agent, in a task allocation framework, makes its decision in a two-step process. First, it chooses a decomposition from the set of alternative decompositions. Then for each subtask in the chosen decomposition the agent chooses one of its neighbors to assign. Formally, each agent needs to learn two policies: $\pi_{high}(T_i, d_j)$ and $\pi_{low}(T_i, n_j)$. $\pi_{high}(T_i, d_j)$ is the probability of choosing decomposition $d_j \in D(T_i)$, while $\pi_{low}(T_i, n_j)$ is the probability of choosing neighbor n_j to assign to task T_i. Any of the two policies (or both) can be deterministic (e.g. $\forall T_i \exists n_j : \pi(T_i, n_j) = 1$). However, one would expect deterministic policies to be suboptimal as they can not balance the load as well as stochastic policies.

While π_{low} could have been conditioned on the chosen composition (i.e. $\pi_{low}^{d_k}(T_i, n_j)$, where d_k is the chosen decomposition by π_{high}) we opted to make both π_{high} and π_{low} independent. This speeds up learning, because a single π_{low} is shared across decompositions and across tasks, but is not always optimal. For example, consider again the scenario in Figure 1. Let agent MA receives task TAB and assume MA can decompose TAB to $\{TA, TA\}$. Now the cost of assigning one task TA to $A1$ is not independent of the decomposition. The cost depends on how the other task is assigned (if both are

assigned to the same agent then the cost should be higher). Nevertheless, in most cases this is a valid approximation as verified by our results.

Agents communicate with each other using messages. There are only two types of messages: REQUEST and RESPONSE. A agent M_{sender} sends a REQUEST message to agent $M_{receiver}$ asking it to accomplish certain task. $M_{receiver}$ estimates the cost for accomplishing the requested task (as will be described shortly) and sends a RESPONSE message, with the estimated cost, back to M_{sender}. Therefore, the operation of each agent is driven by received messages (i.e. event driven) and is divided into two algorithms for processing each message type. Algorithm 1 is where decision making occurs (deciding how to decompose a task and assign subtasks) while Algorithm 2 is where learning takes place.

Algorithm 1: Process REQUEST message

Input: REQUEST from M_{sender} to $M_{receiver}$ to do task T_i

1.1 begin

1.2 Choose a decomposition d^* uniformly at random proportional to $\pi_{high}(T_i, d_j)$, $\forall d_j \in D(T_i)$.

1.3 for each task $T_k \in d^*$ choose a neighbor n_l uniformly at random proportional to $\pi_{low}(T_k, n_l)$, $\forall n_l \in N(M_{receiver})$. Let $A = \{a_1, ..., a_{|d^*|}\}$ be the chosen set of assignments for each subtask of d^*, where $a_k = \langle T_{a_k}, n_{a_k} \rangle$.

1.4 Send a RESPONSE message to M_{sender} with the estimated cost of A, $C_{T_i} = \sum_{\langle T_i, n_j \rangle \in A} C(T_i, n_j)$.

1.5 Send REQUEST messages to neighbors according to A.

1.6 end

Algorithm 2: Process RESPONSE message

Input: RESPONSE from neighbor n_j regarding task T_i with estimated cost C

2.1 begin

2.2 Let $n^* = argmin_{n_j} C(T_i, n_j)$.

2.3 update the cost $C(T_i, n_j) \leftarrow (1 - \alpha)C(T_i, n_j) + \alpha C$.

2.4 update policy (either deterministically or stochastically as described shortly)

2.5 end

3.1 Learning

Learning a stochastic policy is usually slower and more difficult than learning a deterministic policy (Q-learning [**?**] is a well known and understood learning algorithm for deterministic policies). Learning a stochastic policy usually involves some sort of policy gradient algorithms as described in Algorithm 2. The main unknown variable for each agent is the cost of assigning a certain task type to a neighbor. This negative value can be learned using a simple update equation derived from Q-routing [**?**]: $C(T_i, n_j) \leftarrow (1 - \alpha)C(T_i, n_j) + \alpha C^{new}(T_i, n_j)$. The equation merges previous cost

estimate, $C(T_i, n_j)$, with a newly received cost estimate, $C^{new}(T_i, n_j)$, using a weight parameter α.

Updating policies π_{low} and π_{high} can be done either deterministically using Q-routing-based [**?**] approach or stochastically using policy gradient approach. Algorithm 3 shows the deterministic approach while Algorithm 4 shows the policy gradient approach. Experimental results compares both extremes and hybrids of them.

Algorithm 3: Deterministic Policy Update

Input: task T_i

3.1 **begin**

3.2 $\forall n_j : \pi_{low}(T_i, n_j) \leftarrow 1$ iff $n_j = argmax_k C_{n_k}(T_i)$

3.3 otherwise $\pi_{low}(T_i, n_j) \leftarrow 0$

3.4 $\forall T_l, d_j$ s.t. $T_i \in d_j$ and $d_j \in D(T_l) : \pi_{high}(T_l, d_j) \leftarrow 1$ iff $d_j = argmax_k \sum_{T_u \in d_k} max_m C_{nm}(T_u)$

3.5 otherwise $\pi_{high}(T_l, d_j) \leftarrow 0$

3.6 **end**

Algorithm 4: Policy Gradient Update

Input: task T_i and neighbor n_j

4.1 **begin**

4.2 $\pi_{low}(T_i, n_j) \leftarrow \pi_{low}(T_i, n_j) + \delta$ iff $n_j = argmax_k C_{n_k}(T_i)$

4.3 otherwise $\pi_{low}(T_i, n_j) \leftarrow \pi(T_i, n_j) - \delta$

4.4 normalize π_{low} s.t. $\sum_{n_j} \pi(T_i, n_j) = 1$

4.5 $\forall T_k, d_l$ s.t. $d_l \in D(T_k)$ and $T_i \in d_l : \pi_{high}(T_k, d_l) \leftarrow \pi_{high}(T_k, d_l) + \delta$ if d_l is the best decomposition for T_k

4.7 otherwise $\pi_{high}(T_k, d_l) \leftarrow \pi(T_k, d_l) - \delta$

4.8 normalize π_{high} s.t. $\sum_{d_j} \pi(T_i, d_j) = 1$

4.9 **end**

The policy gradient algorithm above uses a fixed learning rate δ. The smaller δ is the more careful our algorithm explores the policy space, and hence the more likely it will converge to an optimal policy. However, the smaller the δ is the slower the convergence. In this work we propose using dynamic learning rates that are derived from learned costs. The use of different learning rate of an agent depending on the agent's performance has been proposed before [**?**]. However, previous work only used two fixed values of learning rates. We propose taking advantage of the consistency of the cost estimates (all are non positives) and scales δ accordingly. In particular, line 4.7 is modified to "otherwise $\pi_{high}(T_k, d_l) \leftarrow \pi(T_k, d_l) - \delta \frac{C(T_k, d_l)}{max_{d_u} C(T_k, d_u)}$", where $C(T_k, d_u)$ is the minimum cost of allocating T_k if decomposition $d_u \in T_k$ is chosen; i.e. $C(T_k, d_u) = \sum_{T_i \in d_u} min_{n_j} C(T_i, n_j)$. To prevent spikes in learning rate, especially in the beginning of learning, the learning rate is not allowed to surpass a threshold δ_{max}.

3.2 Cycles

Like any routing algorithm, it is possible to have cyclic policies. For example, two neighboring agents may send the same task back and forth between each other. Such a policy is undesirable as it wastes system resources without getting any real work done. This problem has two aspects. The first is detecting such a cycle. The second is choosing an appropriate reinforcement signal to penalize such behavior.

There are two known methods to detect cycles. The first method assigns a unique identifier for each task. Each agent then keeps track of task identifiers it had seen. A cycle is detected once a task identifier is seen twice. Two problems make this approach unappealing: ensuring the uniqueness of task identifiers across the distributed network and deciding for how long to keep task identifiers. A simpler yet approximate approach is to use the *age* of a task to detect a cycle. If a task has been floating in the system for too long, then it is likely that there is a cycle. What makes this approach approximate is defining the maximum age. Optimally, maximum age should be the diameter of the network. However, in an open and dynamic system, it is unlikely that any agent would know the diameter of the network. We use the second approach in our experiments.

Once a cycle is detected at an agent, the system faces the credit assignment problem: determining who is/are responsible and penalizing them. Several factors make this problem difficult: use of stochastic policies, partial observability, and using task age for detecting cycles. All these factors add uncertainty to determining who is/are responsible for the cycle. For example, the agent that received an old task may not even be part of the cycle. The work in [**?**] used a global penalty signal (i.e. all agents are penalized once a cycle is detected). Their approach does not scale to a large open system. Our work on the other hand uses a local penalty: the agent who received a too old task sends a high negative penalty to the sender of that task. Experimental results show the effectiveness of this approach in conjunction with our learning algorithm.

4 Experimental Results

The first part of our results evaluate the performance using the example scenario in Figure 1. This helps in getting better understanding of how our approach works. The second part evaluates the scalability of the approach using the scenario in Figure 2. Both parts aim at evaluating the benefit of both multi-leveled policies and the dynamic learning rate.

For the small scenario in Figure 1, in each time step a task of type TAB appear at agent MF with probability 0.5 and at agent MA with probability 0.5. Agent MD, the only agent who knows how to decompose TAB, does not receive any task directly. The cost of any task at a resource R is $-10 \times load(R)^2$, where $load(R)$ is the number of tasks currently being serviced at resource R. When a resource fails to accomplish a task (e.g. when a resource of type A is assigned a task of type TB), a penalty of -10000 is imposed as a cost. A task also fails if it reaches age 10 time units. The cost of communicating a task to a neighbor is -1. Tasks takes 5 time units to execute on resources of type A or B and only 3 time units to execute on either A_f or B_f.

Figure 3 compares the performance of our algorithm for three settings of the learn-

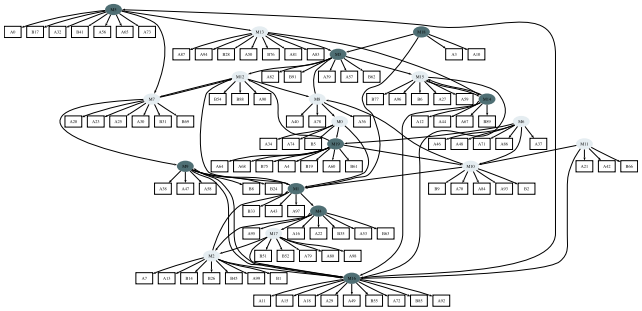


Figure 2: A large scale network of 100 resources and 20 agents.

ing rate δ: dynamic between 0.0001 and 0.01, static at 0.01, and static at 0.0001. The horizontal axis is the time steps while the vertical axis is the absolute sum of incurred costs per 100 time steps, averaged over 10 simulation runs (lower is better). The static-at-0.0001 is too slow and it did not converge even after 10000 times steps. As expected, a larger static learning rate (0.01) leads to faster convergence. Using a dynamic learning rate strikes a balance by converging to a much better policy than static-at-0.01 (less than 25% of its cost) in much less time than the static-at-0.0001. Although there might be a static learning rate that achieves performance similar to that of the dynamic rate, it is much harder to fine tune the learning rate to a fixed value than to specify the range of the dynamic rate (we used $\delta = 0.0001$ and $\delta_{max} = 0.01$).

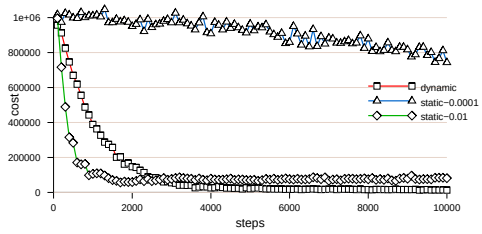


Figure 3: The effect of the dynamic learning rate.

Figure 4 compares the performance of our algorithm using four settings of the policies π_{low} and π_{high}: both are deterministic (*deterministic*), only π_{low} is stochastic (*low*), only π_{high} is stochastic (*high*), and both are stochastic (*two-level*). As expected, *two-level* is the slowest to converge but achieves the lowest steady cost (about 80% of the second lowest steady cost, *low*). On the other hand, and to our surprise, *high* converges faster than *deterministic* (and achieves lower steady cost than *deterministic*, which is expected). The reason is that even without any learning, π_{high} selects a decomposition uniformly at random. This slightly balances the load without paying the price of slow convergence due to learning a stochastic π_{low}.

Figure 5 illustrates the evolution of stochastic policies in agents MD and MA during a simulation run. The horizontal axis represents time steps. The vertical axis

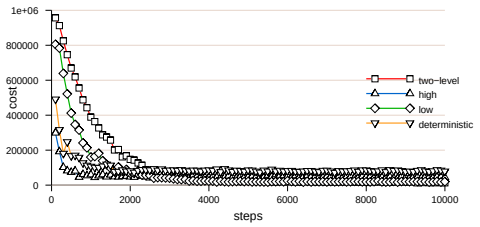


Figure 4: The effect of two level stochastic policies on performance.

represents policies, i.e. the total 1.0 probability divided over actions (an action is a neighbor in case of π_{low} and a decomposition in case of π_{high}). Figure 5(a) shows π_{low} of task TB at agent MD. There are four possible assignments of TB, to each neighbor of MD. The probability of assigning $A3$, which is a resource of type A, quickly drops to zero as expected. Also since MA is not directly controlling any resources of type B, the probability of MD choosing MA also drops to zero but after a while (about 6000 steps). The reasons are cycles and indirect links. Initially MD may send a request for a task of type TB to MA who in turn either sends it to MF or back to MD. However, using the simple maximum task age mechanism, eventually MD learns to stop sending tasks of type TB to MA. In the end, MD only chooses among two assignments for TB: $B1$ and MF, with more probability of choosing MF. This what one would expect to balance the load: faster resources get more tasks.

Figure 5(b) shows $\pi_{low}(TB, .)$ for agent MA. After step 6000 we see the policy almost fixed. This is because MA is not receiving any tasks of type TB from agent MD, therefore it stopped learning about it. Figure 5(c) shows how MD learns π_{high} for different decompositions of task TB. Agent MD quickly learns to drop decomposition $\{TA, TB\}$. The reason is that this decomposition requires equal numbers of resource types A and B, while the system contains 4 A resources and only 2 B resources. MD converges to an intuitive policy that produces more TA tasks than TB tasks.

The second part of the results show the scalability of our approach using the system in Figure 2. This system consists of 100 resources (rectangles) and 20 agents (ellipses). With probability 0.67 the resource is of type A, otherwise it of type B. Also with probability 0.67 the resource is normal, otherwise it is fast. Each agent has two neighboring agents picked randomly from the set of agents. Each resource is connected randomly to one of the agents. At each time step, tasks of type TAB appear at 11 agents (light gray) with probability 0.5. The other 9 agents (dark gray) know how to decompose tasks of type TAB. Other parameters are the same as the small scenario. Therefore, the average number of TAB tasks per 100 time steps is $0.5 \times 11 \times 100 = 550$, which requires (after decomposition) 1100 resources. A lower bound on the average cost, assuming perfect knowledge and perfect distribution of load, is 11000. The highest average cost (if all tasks allocated to the same resource) is Figures 6 and 7 show the performance of the different approaches in the larger system. We can see significant savings of our approach compared to the other approaches.

5 Related Work

In [**?**], a mediator serially allocates tasks to agents. That work used a Markov Decision Process (MDP) model where actions are agent-task assignments and learned a deterministic policy. This differs from our work where all subtasks are allocated concurrently and using two-level stochastic policy. That work also assumed the set of tasks were *fixed* and arrived in fixed order, while we assume tasks arrive stochastically in time and location. They also assumed agents with homogeneous capabilities, while our model supports heterogeneous agents.

The work in [**?**] modeled the resource allocation problem as a constrained MDP, or CMDP. A CMDP is an MDP augmented with a set of (resource) constraints. The set of actions were assumed fixed and the policy was serial and deterministic. They also used an offline algorithm which solved the problem assuming the transition probabilities are known. We use an on-line algorithm without sharing state information among agents.

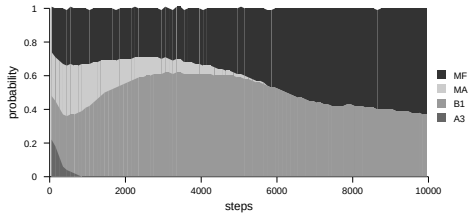
Task allocation can be viewed as a more complex and more general form of packet routing. As in routing, each agent acts as a router, trying to route the packet through the least costly path. Packets impose little load on the nodes (resources) as opposed to tasks, which raises the issue of load balancing. Also task allocation involves alternative decompositions while packets are routed as non-decomposable units. Most of the previous work in packet routing [**?**, **?**] maps the routing problem to a set of local decision problems for each agent. The work used reinforcement learning techniques to learn a deterministic policy for each router. The goal was to minimize average packet delay. Experimental results showed the effectiveness of the approach. More recently, a policy gradient approach was used to solve the packet routing problem [**?**]. However, the work ignored the load on the nodes and only focused on the capacity of links. Their policy gradient also used a fixed learning rate, unlike the algorithm presented here.

6 Conclusion and Future Work

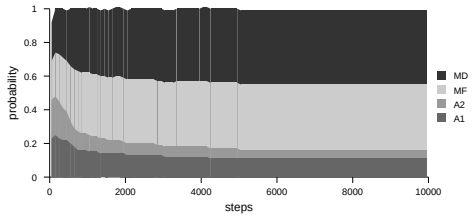
This paper presents a novel algorithm that allows agents in a network to learn cooperatively how to allocate a task. The algorithm learns two-level stochastic policies using policy gradient. The high level policy selects a decomposition for an incoming task while the low level policy assigns a neighboring agent to each task in the selected decomposition. Experimental results show the benefit of introducing each of these levels with more than four times saving in cost as compared to deterministic approaches. Our algorithm also dynamically adjusts the learning rate. Experimental results show how using a dynamic learning rate significantly speeds up convergence while outperforming learners with fixed learning rate.

An interesting issue that was not covered in the paper is how to set up the network connections, i.e. the neighborhood of each agent N. Optimally, the network should reduce communication overhead by adapting to task arrival patterns. For example, an agent that receives many tasks asking for resource R_x should be connected as closely as possible to resources of that type. A related issue is how the system would perform in the face of changes in the network (e.g. an agent or a resource leaving the system or another agent or a resource entering.) Furthermore, this work models resource failures

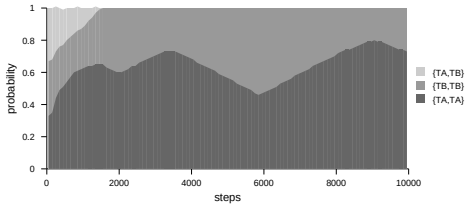
implicitly using penalties. An explicit model of failure probability may allow agents to learn better policies (e.g. preferring an agent with high probability of failure if its cost is cheap and the task payoff is low, or vice versa).



(a) $\pi_{low}(TB, .)$ for agent MD.



(b) $\pi_{low}(TB, .)$ for agent MA.



(c) $\pi_{high}(TAB, .)$ for agent MD.

Figure 5: Policies of different agents.

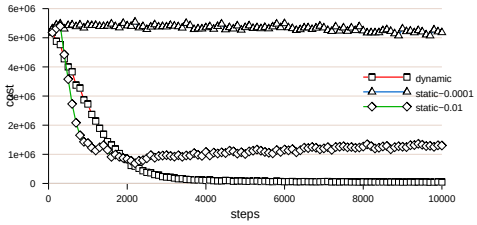


Figure 6: The effect of dynamic learning rate in the large system scenario.

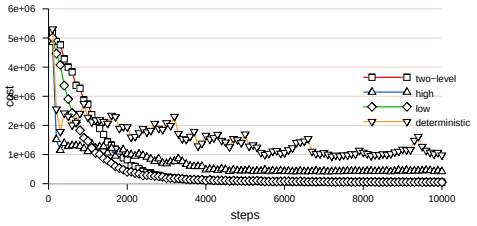


Figure 7: The effect of two level policies in the large system scenario.