2016 IEEE International Conference on Data Science and Advanced Analytics

Parallel Least-Squares Policy Iteration

Jun-Kun Wang
National Taiwan University
wangjim123@gmail.com

Shou-De Lin
National Taiwan University
sdlin@csie.ntu.edu.tw

***Abstract*—Inspired by recent progress in parallel and distributed optimization, we propose parallel least-squares policy iteration (parallel LSPI) in this paper. LSPI is a policy iteration method to find an optimal policy for MDPs. As solving MDPs with large state space is challenging and time demanding, we propose a parallel variant of LSPI which is capable of leveraging multiple computational resources. Preliminary analysis of our proposed method shows that the sample complexity improved from $\mathcal{O}(1/\sqrt{n})$ towards $\mathcal{O}(1/\sqrt{Mn})$ for each worker, where n is the number of samples and M is the number of workers. Experiments show the advantages of parallel LSPI comparing to the standard non-parallel one.**

I. Introduction

Learning an optimal policy for MDPs with large state space has gained many interests recently [3], [24], [4], [25]. Different from previous works, our proposed method is inspired by the recent success in distributed optimization [13], [17], [20]. The goal is to parallelize an existing policy iteration method called least-squares policy iteration. The algorithm takes advantage of the multi-core or multi-machine architecture, where each worker (one per core or machine) individually executes a fraction of episodes and estimates a parameter while a consensus is maintained by parameter averaging. With the feedback of global consensus, each worker can access the information learned by other workers at the previous iterations. As the result, the learning process of each individual worker can be accelerated, as compared to learning alone.

Recently, there has been several works focusing on distributing the computational burden to many available computational resources. For example, open-source toolkits like GraphLab [17] and Petuum [20] implement parallel variants of existing machine learning methods such as Lasso regression, Latent Dirichlet Allocation, Neural Net, and Matrix Factorization. Yet, to our knowledge, these toolkits currently lack libraries specially aiming at parallelization of sequential prediction and learning. We bridge this gap by proposing a parallel version of LSPI, which is a common policy iteration method in reinforcement learning.

Our work is different from the works of multi-agent MDP [1], [5]. In multi-agent MDP, a large, structured MDP is used to model the corporation of agents. The actions, rewards, and even states of the MDP can be factored into several subsystems with each corresponding to subgroups of agents, while some variables in a subsystem are shared with others. Each subsystem then can be solved locally with message passing for coordination with others, and can be solved in a centralized or distributed way. In contrast, our work aims at answering the following question: *Given multiple computational resources, how to efficiently solve an MDP?* In our problem, each worker faces the same MDP, and each worker communicates with others about the estimated parameter during learning. Thus, our work can be regarded as a complement to multi-agent MDP.

There exists some related works that share the same goal with ours. Kretchmar [9] proposed to average the state-action value function in parallel for sharing the experience of workers, but the method was only designed for a simple state problem with tabular representation. Li and Schuurmans [21] studied how to parallelize several existing reinforcement learning algorithms using Map-Reduce. They achieved the goal by decomposing some matrix-vector multiplications in the original algorithms so these decomposed computations can be performed in parallel, which is different from a parallel variant as we consider here. Perhaps the most closely related work to ours is [18]. They parallelized the temporal difference (TD) learning algorithm by allowing each worker to independently run K episodes for T durations, then gathers the results to initialize the weights for next iteration. Although their aim is very similar to ours, one major concern in their work is that in order to guarantee the result does not significantly deviate away from that of its non-parallel counterpart, it is required that the number of episodes K and number of workers M to be small. In contrast, in our parallel LSPI, each worker can execute large K number of episodes before communicating with others, which is very important in practice because the consumption of network bandwidth is then significantly reduced. In other words, parallel LSPI allows each worker to individually collect a large amount of samples before parameter mixing.

Our analysis on parallel LSPI shows that the correlation between the learning processes of each individually learned model can influence the effectiveness of the method. The computation gains achieved with parallel LSPI is less significant when there exists high correlation between workers. To deal with this issue, a heuristic is proposed to encourage each worker to explore (i.e. taking random action) more when it collects samples, which increases the randomness and in turn reduces correlation.

To summarize, we propose parallel LSPI to efficiently solve an MDP through parallel programming. Our method can also balance the communication overhead and required number of iterations to find the optimal solution, which is suitable for situation when only limited bandwidth is available. We give some analysis for the proposed method and conduct experiments to show its effectiveness on queueing networks [3], [25] and persistent search and track [7] domains. The codes to reproduce the experiments will be available online.

978-1-5090-5206-6/16 $31.00 © 2016 IEEE
DOI 10.1109/DSAA.2016.24

II. Backgrounds

A. Preliminaries

A Markov decision process (MDP) [23], [22] is a 5-tuple (S, A, P, R, γ), where $S = \{s_1, s_2, \ldots, s_{|S|}\}$ is a set of states, $A = \{a_1, a_2, \ldots, a_{|A|}\}$ is a set of actions, P is the transition matrix, where $P(s, a, s')$ is the transition probability to the next state s' when taking action a in state s, R is the reward matrix, where $r(s, a)$ is the reward of taking action a in state s, and γ is the discount factor. A deterministic policy π maps each state to a single action. The state value function $V^\pi(s)$ and the state-action value function $Q^\pi(s, a)$ underlying a policy π are defined respectively as

$$V^\pi(s) = E_{a_t \sim \pi, s_t \sim P}[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s], \tag{1}$$

$$Q^\pi(s, a) = E_{a_t \sim \pi, s_t \sim P}[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a]. \tag{2}$$

(2) can also be expressed as

$$Q^\pi(s, a) = \sum_{s' \in S} P(s, a, s')[r(s, a) + \gamma Q^\pi(s', \pi(s'))], \tag{3}$$

which is known as the Bellman equation, and the state-action value function is the fixed point of the equation. Let us write it in a matrix form,

$$Q = R + \gamma P B_\pi Q \doteq T^\pi Q, \tag{4}$$

where Q and R are $|S||A|$ dimensional vectors. An entry $P((s, a), s')$ in $P \in \mathbb{R}^{|S||A| \times |S|}$ is defined as $P((s, a), s') = P(s, a, s')$, and an entry in $B_\pi \in \mathbb{R}^{|S| \times |S||A|}$ is defined as $B_\pi(s, (s, a)) = 1$, if $a = \pi(s)$.

For a large state-action space, we generally use compact representation to approximate the action value,

$$Q^\pi(s, a) \approx \widehat{Q}^\pi(s, a) = \phi(s, a)^T \theta, \tag{5}$$

where $\phi(s, a) \in \mathbb{R}^d$ is the feature vector of the state-action pair (s, a), and $\theta \in \mathbb{R}^d$ is the vector of weights for each dimension and needs to be learned. Generally, $d \ll |S||A|$. We can also write it in a matrix form: $\widehat{Q}^\pi = \Phi\theta$, where Φ is the feature matrix with its row being $\phi(s, a)^T$.

The *optimal action value* $Q^*(x, a)$ is the maximum of the expected discounted rewards given the process starts from state s and takes action a at the beginning. The *optimal value function* $V^*(x)$ is defined similarly, and is connected with $Q^*(x, a)$ by

$$V^*(x) = max_{a \in A} Q^*(x, a). \tag{6}$$

The goal for solving MDP is to find an optimal policy. An optimal policy π^* maximizes the expected discounted rewards in all states, as defined below,

$$\pi^* = \text{argmax}_\pi V^\pi(s), \forall s \in S. \tag{7}$$

From (6), we see that a greedy policy with respect to Q^* (i.e. choosing an action that maximizes $Q^*(s, \cdot)$) is optimal.

B. Least-Squares Policy Iteration (LSPI)

Policy iteration is an algorithm to learn an optimal policy by iteratively performing policy evaluation and policy improvement [23], [22]. In policy evaluation, the approximated state-action value $\widehat{Q}$ under current policy π is computed. In policy improvement, the (greedy) policy is updated based on the new approximated state-action value. As function approximation used in (5), $\widehat{Q}^\pi$ lies in the space spanned by the column space of Φ. However, $T^\pi Q^\pi$ may not lie in the feature space. Let $\Phi(\Phi^T\Phi)^{-1}\Phi^T$ be the orthogonal projection to the space, the following fix point equation is solved instead of (4)

$$\widehat{Q}^\pi = \Phi(\Phi^T\Phi)^{-1}\Phi^T(T^\pi \widehat{Q}^\pi), \tag{8}$$

where $\widehat{Q}^\pi$ is parameterized by θ. After some manipulation [12], the solution θ of the above equation can be shown as

$$\theta = (\Phi^T(\Phi - \gamma P B_\pi \Phi))^{-1} \Phi^T R. \tag{9}$$

Let us denote $A = \Phi^T(\Phi - \gamma P B_\pi \Phi)$ and $b = \Phi^T R$, then, (9) can be written as $\theta = A^{-1}b$.

LSPI [12] does not assume the knowledge of P and R. Still, given the policy π, the samples can be collected along with π. That is, sample are collected in the form $(s_i, a_i, r_i, s'_i, a'_i)$, where $a_i = \pi(s_i)$, $s'_i \sim P(s_i, a_i, \cdot)$, $a'_i = \pi(s'_i)$, $s_{i+1} = s'_i$, and $a_{i+1} = a'_i$. Using the collected samples, A and b can be approximated as

$$\begin{aligned} \widehat{A} =& \frac{1}{L} \sum_{i=0}^{L} [\phi(s_i, a_i)(\phi(s_i, a_i) - \gamma \phi(s'_i, a'_i))^T], \\ \widehat{b} =& \frac{1}{L} \sum_{i=0}^{L} [\phi(s_i, a_i) r_i]. \end{aligned} \tag{10}$$

When collecting samples, one typically uses ϵ-greedy policy. With probability $1 - \epsilon$, an action is selected greedily with respect to current $\widehat{Q}^\pi$; and with probability ϵ, an action is chosen randomly. The strategy tries to deal with the so-called exploration and exploitation tradeoff [23], [22].

In LSPI, the original form [12] is an off-line algorithm. It takes transitions as the input, which can come from the random policy, and outputs the learned policy. Here we consider the online version of LSPI [15], [7]. First, θ is initialized arbitrarily. Then, samples are gathered by following the ϵ-greedy policy π based on the most recent updated θ. For each transition, $\widehat{A}$ and $\widehat{b}$ are incrementally updated using the contribution of the new sample $(s_i, a_i, r_i, s'_i, a'_i)$,

$$\begin{aligned} \widehat{A'} =& \widehat{A} + \phi(s_i, a_i)(\phi(s_i, a_i) - \gamma \phi(s'_i, a'_i))^T, \\ \widehat{b'} =& \widehat{b} + \phi(s_i, a_i) r_i. \end{aligned} \tag{11}$$

After collecting a few samples, θ is updated through $\widehat{A}^{-1}\widehat{b}$. In practice, one may add a small constant λ to the diagonals of $\widehat{A}$ before calculating its inverse to prevent numerical instability. $\widehat{A}$ and $\widehat{b}$ are then reset to zeros and new samples are collected based on the updated ϵ-greedy policy π. The algorithm also reuses the samples collected at previous iterations by switching a'_i to $\pi(s'_i)$, where π is the current policy. The procedure is iteratively repeated for pre-specified number of iterations.

Algorithm 1 Parallel Least-Squared Policy Iteration (Parallel LSPI)

```
Input: MDP $\backslash\{P, R\}$, $\epsilon$, number of iterations $T$, number of episodes per iteration $K$, and number of workers $M$ .
Each worker $m$ initialize $\theta_m$ randomly and set $\widehat{A}_m = \mathbf{0}$, $\widehat{b}_m = \mathbf{0}$.
for $t = 1$ to $T$ iterations do
    for $m = 1$ to $M$ workers do
        (each work executes the following in parallel)
        for $k = 1$ to $K$ episodes do
            Each worker collects samples by following $\epsilon$-greedy policy and incrementally updates $\widehat{A}_m$, $\widehat{b}_m$ by (11).
        end for
        For samples collected at $t' = 1, \ldots, t-1$, $a'_i \leftarrow \text{argmax}_{a \in A} \widehat{Q}^{\pi}(s'_i, a), \forall i$
        Updating $\theta_m = \widehat{A}_m^{-1} \widehat{b}_m$
        Each worker $m$ sends its $\theta_m$ to the master.
    end for
    The master reaches a consensus $z = \Sigma_m \theta_m / M$ and broadcasts to all the workers.
    Each worker sets $\theta_m = z$ and resets $\widehat{A}_m$, $\widehat{b}_m$.
end for
Output: The most recent $z$.
```

III. Parallel Least-Squares Policy Iteration (Parallel LSPI)

We propose the parallel least-squares policy iteration to handle the large-scale learning problem. The setting is that there are M workers (cores) available (either multiple machines or multiple cores on a single machine) for computations. To fully exploit the available computational resources, each worker m collects samples and runs by itself, and then updates its estimated $\widehat{A}_m^{-1}$, $\widehat{b}_m$, and θ_m. At some point during learning, it communicates the learned θ_m with other workers.

The algorithm is shown in Algorithm 1. For every outer iteration t, each worker m individually collects samples by following ϵ-greedy over K episodes. When collecting samples, each worker also incrementally updates $\widehat{A}_m$ and $\widehat{b}_m$. After conducting K episodes of learning, each worker also reuses the samples collected at previous iteration to updates $\widehat{A}_m$ and $\widehat{b}_m$. Then, each worker updates θ_m and sends it to the master. The master then averages the models to obtain the consensus z and broadcasts it to all the workers. The workers then update the policy with the new consensus and proceed to the next iteration. After T iterations, parallel-LSPI outputs the most recent consensus z_T.

In parallel LSPI, each worker m communicates to the master only after updating its estimator θ_m, which occurs when it has executed sufficient number episodes. This is the strategy that balances between communication overheads and required iterations to the optimal solution. If the algorithm dictates each worker to communicate right after every episode, communications overheads becomes heavy. If each worker independently runs the episodes during training and the parameters are only averaged at the end, the communication overhead would be minimized but it may not lead to satisfactory results as the information from other workers is completely ignored during training. Thus, a better strategy is to strike a balance between the two extreme. Compared to the related work of parallel TD [18] that only allows small K (roughly $K < 5$), the proposed parallel LSPI can reduce communication cost by allowing the workers to run sufficient amount of episodes before mixing. Note that the underlying core of LSPI is least-squares temporal difference learning (LSTD_Q), which is naturally a batch method in contrast to TD learning. Therefore, it does not need to update θ for every transition as TD learning does. Thus, parallel LSPI naturally enjoys the benefit of parallelization without the burden of frequent communication.

IV. Analysis

Here we analyze the sample complexity of the proposed method. As [11] for non-parallel LSPI did, the analysis is first performed on a version of LSTD called *pathwise-LSTD* for policy evaluation. It analyzes LSTD at the states along a sampled trajectory following a given policy. As there are M workers (and M trajectories) with parameter averaging conducted in our case, we have to analyze the averaged estimated parameters from the trajectories. Then, by the strategy stated in [11], one may generalize the analysis over entire state space under certain condition and derive the finite-sample bound of parallel LSPI in turn. Thus, here we focus on *parallel pathwise-LSTD* as the insight on the sample complexity can already be seen in this step.

Before explaining pathwise-LSTD (see also [11]), let us first describe the notations. As an MDP is reduced to a Markov chain given a policy π, let $(X_1, X_2, \ldots, X_n)$ be a trajectory of size n generated by the Markov chain. With abuse of notation [1], we denote $\Phi = [\phi(X_1)^T; \ldots; \phi(X_n)^T]$ as the feature matrix defined along the trajectory. The estimated value function is thus constrained on the feature space $\mathcal{F} = \{\Phi\theta, \theta \in \mathbb{R}^d\}$. Pathwise-LSTD takes the feature matrix Φ generated by a trajectory following π as input. It builds the empirical transition matrix $\widehat{P} : \widehat{P}_{ij} = \mathbb{I}\{j = i+1, j \neq n\}$, and sets the quantities $A = \Phi^T(I - \gamma\widehat{P})\Phi$, and $b = \Phi^T r$. It then outputs the solution $\widehat{\theta} = A^+ b$ with minimum norm, where A^+ represents the Moore-Penrose pseudo-inverse of A.

Let us denote v, $\widehat{v}$ as the value function and its estimated one along the trajectory $\{X_t\}_{t=1}^n$, and $\|f\|_n^2 = \frac{1}{n}\sum_{t=1}^{n} f(X_t)^2$ as the empirical norm. Moreover, let V_{max} represent the maximum of the value function, $\widehat{\Pi}$ be the projection to the feature space, and ν be the smallest positive eigenvalue of the Gram matrix $\Phi^T\Phi/n$. From Theorem 1 in [11], $\|v - \widehat{v}\|_n$ is bounded as

$$\|v - \widehat{v}\|_n \le \frac{1}{\sqrt{1-\gamma^2}}\|v - \widehat{\Pi}v\|_n + \frac{1}{1-\gamma}[\gamma V_{max} L \sqrt{\frac{d}{\nu}}(\sqrt{\frac{8log(2d/\delta)}{n}} + \frac{1}{n})], \tag{12}$$

with high probability $1 - \delta$.

For parallel pathwise-LSTD, denote $(X_{1,m}, X_{2,m}\ , \ldots, X_{n,m})$ as the m_{th} trajectory of the Markov chain induced by a policy π, $\Phi_m = [\phi(X_{1,m})^T\ ; \ldots; \phi(X_{n,m})^T]$ as the corresponding feature matrix, and ν_m as the smallest positive

[1] The analysis can be extended to action value function as well, for brevity, we just use value function here.

eigenvalue of the corresponding sample based Gram matrix of features. Moreover, let $\widehat{\theta}_m$ represent the pathwise solution of the trajectory m, and v_m, $\widehat{v}_m = \Phi_m \widehat{\theta}_m$ represent the value function and the estimated one at the states along trajectory m respectively. Since parallel LSPI conducts parameter averaging, we are interested in the sample complexity associated with the averaged estimator $\widehat{\theta} = \Sigma_m \widehat{\theta}_m / M$. Let π' represent the new policy at the next iteration of parallel LSPI and $X'_1, X'_2, \ldots, X'_n$ be the trajectory of the Markov chain following π' with Φ' being the corresponding feature matrix. Then, we want to analyze the quantity $\|v' - \Phi' \widehat{\theta}\|_n$ as it would give us insight on the sample complexity. To estimate the upper bound $\|v' - \Phi' \widehat{\theta}\|_n$, we make the following assumption that connects the performance of θ_m evaluated at the new trajectory $\{X'_t\}_{t=1}^n$ to the one evaluated at the original $\{X_{t,m}\}_{t=1}^n$ where θ_m is estimated from.

Assumption *There exists a constant c such that the empirical norm of the difference between value function v' and the one implied by θ_m evaluated at the trajectory $\{X'_t\}_{t=1}^n$ is upper bounded as $\|v' - \Phi' \widehat{\theta}_m\|_n^2 \leq c^2 \|v_m - \Phi_m \widehat{\theta}_m\|_n^2$.*

Notice that the empirical norms on both sides of the inequality are evaluated at different trajectories; the left hand side is on $\{X'_t\}_{t=1}^n$, while the right hand side is on $\{X_{t,m}\}_{t=1}^n$ for an m. Using the assumption, we can bound $\|v' - \Phi' \widehat{\theta}\|_n$, which measures the performance of the averaged estimator.

Proposition *Following the above assumptions, if $(v' - \Phi' \widehat{\theta}_m)$ of each m is near orthogonal, we have*

$$\begin{aligned} \|v' - \Phi' \widehat{\theta}\|_n \leq \frac{c}{\sqrt{M(1-\gamma^2)}} Max_m \{\|v_m - \Pi_m v_m\|_n\} + \\ + \frac{c}{1-\gamma}[\gamma V_{max} L \sqrt{\frac{d}{\nu_{min}}} (\sqrt{\frac{8 log(2d/\delta)}{Mn}} + \frac{1}{n\sqrt{M}})], \end{aligned} \tag{13}$$

where ν_{min} represents the smallest of the ν_m, and Max_m $\{\|v_m - \Pi_m v_m\|_n\}$ represents the largest of the $\|v_m - \Pi_m v_m\|_n$ terms. If $(v' - \Phi' \widehat{\theta}_m)$ are highly correlated, we have

$$\begin{aligned} \|v' - \Phi' \widehat{\theta}\|_n \leq \frac{1}{\sqrt{1-\gamma^2}} Max_m \{\|v_m - \Pi_m v_m\|_n\} \\ + \frac{1}{1-\gamma}[\gamma V_{max} L \sqrt{\frac{d}{\nu_{min}}} (\sqrt{\frac{8 log(2d/\delta)}{n}} + \frac{1}{n})]. \end{aligned} \tag{14}$$

The proposition suggests that, for the ideal case, parallel LSTD can estimate the value function at an improved rate $\mathcal{O}(1/\sqrt{Mn})$ of each worker comparing to the original rate of $\mathcal{O}(1/\sqrt{n})$. This implies that the effectiveness of process in which M workers collect some n/M samples in parallel is comparable to a single worker collecting n number of samples. Yet, the sampling is conducted in a distributed fashion in parallel LSTD as compared to standard LSTD. At the other extreme, for the worst case scenario, parallel LSTD has the same sample rate, $\mathcal{O}(1/\sqrt{n})$, for each worker as that of standard LSTD. Yet, each worker individually collects n samples meaning that the total samples in parallel LSTD are M times larger than that of the non-paralleled one.

Proof: First, let us rewrite $\|v' - \Phi' \widehat{\theta}\|_n^2$ as $\frac{1}{M^2} \| \sum\limits_{m=1}^{M} (v' - \Phi' \widehat{\theta}_m)\|_n^2$. Suppose, for each m, $(v' - \Phi' \widehat{\theta}_m)$ is nearly mutual orthogonal. Then, we have

$$\begin{aligned} \frac{1}{M^2} \| \sum\limits_{m=1}^{M} (v' - \Phi' \widehat{\theta}_m)\|_n^2 \approx \frac{1}{M^2} \sum\limits_{m=1}^{M} \|(v' - \Phi' \widehat{\theta}_m)\|_n^2 \\ \leq \frac{c^2}{M^2} \sum\limits_{m=1}^{M} \|(v_m - \Phi_m \widehat{\theta}_m)\|_n^2 \leq \frac{c^2}{M} G^2, \end{aligned} \tag{15}$$

where

$$\begin{aligned} G = \frac{1}{\sqrt{1-\gamma^2}} Max_m \{\|v_m - \Pi_m v_m\|_n\} + \\ + \frac{1}{1-\gamma}[\gamma V_{max} L \sqrt{\frac{d}{\nu_{min}}} (\sqrt{\frac{8 log(2d/\delta)}{n}} + \frac{1}{n})]. \end{aligned} \tag{16}$$

Taking square root on both sides gives us the result.

In contrast, if all $(v' - \Phi' \widehat{\theta}_m)$ are highly correlated, we have $\|v' - \Phi' \widehat{\theta}\|_n^2 \approx \frac{1}{M^2} \| \sum\limits_{k=1}^{M} (v_m - \Phi' \widehat{\theta}_m)\|_n^2 = \|(v_m - \Phi' \widehat{\theta}_m)\|_n^2$, which reduces to the case of a single worker. ■

To achieve better performance for parallel LSPI, according to the proposition, we should make $(v' - \Phi' \widehat{\theta}_m)$ as less correlated to each other as possible. However, $(v' - \Phi' \widehat{\theta}_m)$ is unknown in advance. To deal with the issue, a heuristic is used to enforce each worker to take random actions more when collecting samples. Since ϵ-greedy is adopted here, a larger ϵ that encourages more exploration (i.e. randomly choosing an action) will bring us closer to the goal. If ϵ gets close to zero, the randomness would mostly comes from the transitions *P* of the process. In this case, performance of parallel LSPI may not achieve significant speedup, depending on the underlying MDP and the feature space. Note that the degree of correlation between $\widehat{\theta}_m$ is not equivalent to the degree of the correlation between $(v' - \Phi' \widehat{\theta}_m)$. Still, our experiments reveal that such heuristic does have positive effect on the performance.

V. Experiments

A. Domains

We conduct the experiments on two domains. One is the discrete-time four-dimensional queuing network, which also appears in [3], [25]. Figure 1 illustrates the network, which includes four queues, each with buffer size B. Here server 1 can only serve queue 1 or 4, and server 2 can only serves queue 2 or 3 one at a time but not simultaneously. Each server can only handle one customer at a time at most. Moreover, neither server can be idle. Let the tuple $\{a_1, a_2, a_3, a_4\}$ represents an action combination which the servers take by considering conditions in the queues $\{q_1, q_2, q_3, q_4\}$, where $a_i = \{1, 0\}$ indicates whether q_i is currently being served or not. Then, there are total four actions $\{1,1,0,0\}$ $\{0,1,0,1\}$ $\{1,0,1,0\}$ $\{0,0,1,1\}$ the servers can take. As the result, the number of states is $(1+B)^4 \times 4$, which means that a modest B will result in a huge state space.

The dynamics of the network are defined by the rate parameters $\mu_1, \mu_3, d_1, d_2, d_3, d_4 \in (0,1)$, all follow Bernoulli distribution. μ_1 and μ_3 are coming rates of new customers. At each time step, with probability μ_i, a new customer comes to queue i. d_i is defined as follows: if $a_i = 1$, which indicates queue i is being served, the server would succeed in handling a customer with probability d_i before the next time step, and fail with probability $1 - d_i$. Starting with empty queues, each

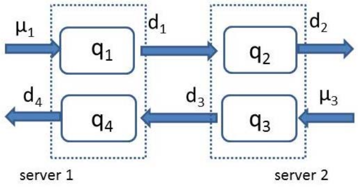


Fig. 1. The discrete-time four dimensional queuing network [3], [25]. Customers can arrive at q_1 or q_3. The customer that is served and finished by q_1/q_3 is then referred to q_2/q_4.

episode spans a fixed number of time steps, T. The goal is to minimize the average of total waiting (unserved) customers in the network during an episode. The loss for a state-action pair is defined as $l(s,a) = l(s) = |X|$, which is the total number of unserved customers in all the queues. After an episode, the network is reset to empty and a new episode begins.

We consider four types of networks:
1) $\mu_1 = 0.5, \mu_3 = 0.5, d_1 = 0.5, d_2 = 0.8, d_3 = 0.8, d_4 = 0.5$, episode duration $T = 50$, and buffer size $B = 5$, which results in $5,184$ state-action pairs.
2) $\mu_1 = 0.5, \mu_3 = 0.8, d_1 = 0.5, d_2 = 0.1, d_3 = 0.8, d_4 = 0.8$, episode duration $T = 100$, and buffer size $B = 10$, which results in $58,564$ state-action pairs.
3) $\mu_1 = 0.4, \mu_3 = 0.4, d_1 = 0.5, d_2 = 0.8, d_3 = 0.3, d_4 = 0.3$, episode duration $T = 200$, and buffer size $B = 15$, which results in $262,144$ state-action pairs.
4) $\mu_1 = 0.4, \mu_3 = 0.4, d_1 = 0.4, d_2 = 0.8, d_3 = 0.8, d_4 = 0.4$, episode duration $T = 200$, and buffer size $B = 15$, which results in $262,144$ state-action pairs.
We design 340-dimensional sparse binary feature for type 1 network and 1048-dimensional features for the others. In our design, only two entries in the feature are non-zeros for each state-action pair.

Another domain is the persistent search and track [7]. The scenario is that there are three Unmanned Aerial Vehicles (UAV) to corporate for a mission. There are three available actions for each UAV: $\{advance, retreat, loiter\}$, resulting in 27 total possible action combinations. The current state of a UAV is described by : location, fuel, actuator status, and camera status. The goal is to fly to the target site and perform surveillance, while ensuring that there is a UAV with a working actuator loitering at the intermediary site to transfer the information of the targets to the base.

We modify the scenario in [7] because the reported performance of LSPI is not good in the original setting. Each UAV starts from the base with 6 units of fuels. The camera and actuator of each UAV can may with a 3% probability at each time step. The camera cannot function under failed actuator, so a UAV with a failed actuator cannot perform surveillance. Yet, a UAV can perform communication even its camera malfunctions. A successful surveillance mission must have at least one UAV with working actuator at the intermediary site, and at least one UAV with working actuator and camera at the surveillance site. At each time step, each UAV loses 1 unit of fuel except when it "loiters" at the base or at the intermediary site. When a UAV "loiters" at the base, the failed camera and actuator are fixed, and the fuels are recharged fully. When a UAV with working actuator "loiters" at the intermediary site, the messages is transmitted to the base and UAV's fuel tank is recharged by 2 units (the fuel cannot exceed the capacity, which is 6 units). If a UAV "retreats" at the base, then it may "advance" to the intermediary site or "loiter" at the base with equal probability. Executing "advance" action at the surveillance site has similar effect. The reward for each state-action pair is defined as $r(s,a) = 15 \times I_{comm} \times I_{surv} - 10 \times I_{crash} - (18 - \text{total remained fuels})$, where I_{comm} indicates whether there is a UAV with working actuator at the intermediary site, I_{surv} indicates whether there is a UAV with working actuator and camera at the surveillance site, and I_{crash} represents whether a UAV crashes. If a UAV runs out of fuel, which means it crashes, the episode is terminated. In total, the state-action pair has the size of about 1.6×10^6, and about 2000-dimensional sparse binary feature including fixed sparse representation [7] is used.

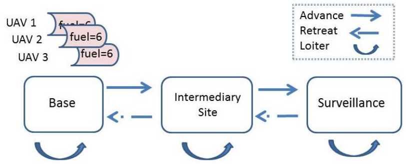


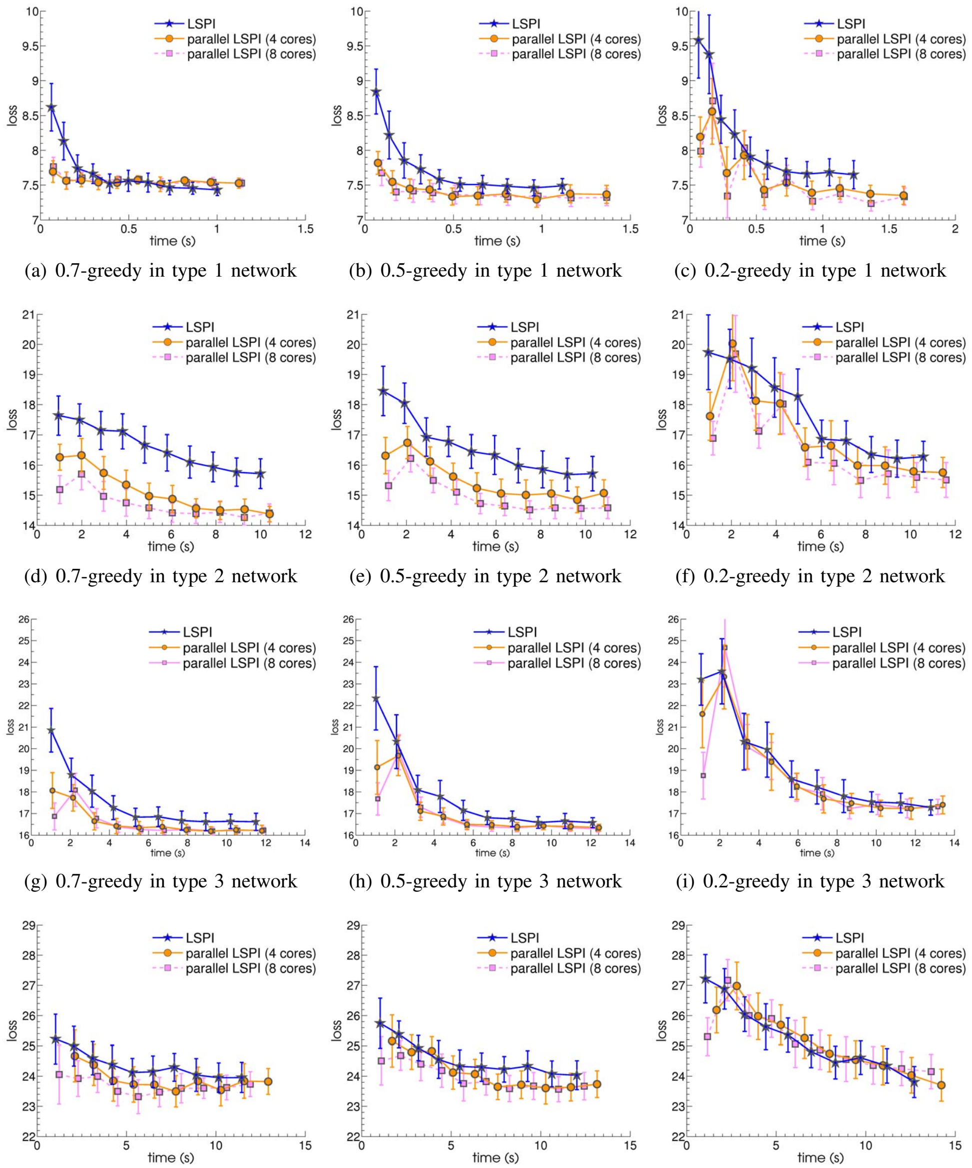
Fig. 2. Persistent search and track.

B. Setup and Results

We compare parallel LSPI with standard LSPI on the two domains. Both methods are implemented in C and the communication in parallel LSPI is implemented with MPI. For parallel LSPI, we report the performance of $M = 4$ and $M = 8$ workers. In our implementation, parallel LSPI uses single core machines, yet the shared-memory architecture (i.e. multi-core on a single machine) is also applicable. For the queuing network domain, LSPI and parallel-LSPI update their policies every 100 episodes, and both of them terminate after learning 1000 episodes, which corresponds to $K = 100$ and $T = 10$ in the algorithm. For persistent search and track, both LSPI and parallel-LSPI update their policies every 1000 episodes, and terminate after 10,000 episodes. We set $\gamma = 0.95$ for both domains. All the experiments are repeated 50 runs with the averaged results and standard deviations reported.

We also try different ϵ for ϵ-greedy policy. Higher ϵ means each worker takes random action with higher probability, which could reduce the correlation between workers. The results for the queuing network are shown in Figure 3, and results for persistent search and track are shown in Figure 4. The learned parameter θ is recorded when it is updated, so each point on the line represents the performance of learned parameter at the end of an iteration of the corresponding algorithm. For parallel LSPI, we record the consensus as the the learned parameter. The performance of a learned parameter in the queuing network domain is measured by the average of losses (where loss of an episode is defined as the average of all waiting customers in the network during an episode) over additional 500 episodes, which are conducted by following the deterministic policy implied by the learned parameter. In persistent search and track domain, the performance is measured by cumulated discounted rewards, with the same evaluation procedure as the queueing network.

Figure 3 suggests for queueing network, higher ϵ yields



(a) 0.7-greedy in type 1 network

(b) 0.5-greedy in type 1 network

(c) 0.2-greedy in type 1 network

(d) 0.7-greedy in type 2 network

(e) 0.5-greedy in type 2 network

(f) 0.2-greedy in type 2 network

(g) 0.7-greedy in type 3 network

(h) 0.5-greedy in type 3 network

(i) 0.2-greedy in type 3 network

(j) 0.7-greedy in type 4 network

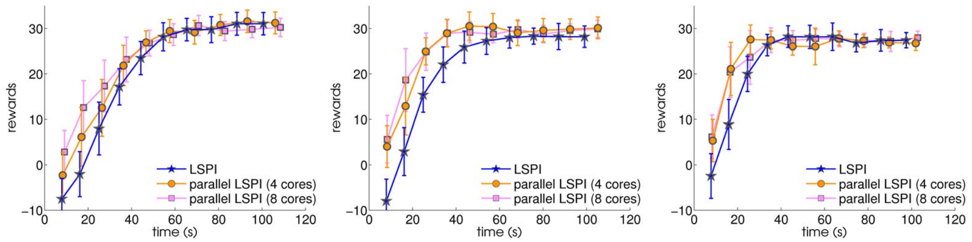
(k) 0.5-greedy in type 4 network

(l) 0.2-greedy in type 4 network

Fig. 3. Losses of the learned parameter versus learning time. Each row corresponds to a network, while each column corresponds to different ϵ. Star marker represents LSPI, circle marker represents parallel LSPI with four workers, and square marker represents parallel LSPI with eight workers. The 0.5 s.e. error bars are also plotted.

better parallelization since the correlation between workers is smaller. At higher ϵ (left column), which encourages taking random action more during learning, parallel LSPI can significantly accelerate the learning process comparing to taking action more greedily with respect to the current estimated state-action value (right column). For parallel-LSPI with 0.7 or 0.5-greedy, after running three or four iterations, it already reaches the point where the standard LSPI needs to take ten iterations or more. For 0.2-greedy in network 3 and 4, parallel LSPI has no advantage while consumes more computation resources than non-parallelized one each iteration. For persistent and search domain, parallel-LSPI with $\epsilon = 0.1$-greedy already achieves significant speedup than its non-parallel counterpart, increasing ϵ can accelerate further but not much more. This reflects the limitation of the proposed heuristic, yet parallel-LSPI still shows the benefit of parallelization. The overhead of parallelization can also be seen on the figures. We can see that parallel LSPI requires slightly more time to finish the same amount of iterations than standard LSPI does due to communication overhead. Yet, this overhead is tolerable since parallel LSPI usually reaches at the same level of performance as standard LSPI with much fewer iterations.

From the figures, we also observe that the advantage of parallelization decreases as the number of workers increases (4 workers vs. 8 workers). The acceleration by doubling the workers is incremental in most of the cases. A possible



(a) 0.1-greedy (b) 0.2-greedy (c) 0.35-greedy

Fig. 4. Rewards of the learned parameter in persistent search and track domain versus learning time. The 0.5 s.e. error bars are also plotted.

explanation is that the correlation between workers is likely to increase with more workers being added. Similar to the parallel TD [18], the degree of parallelization of our method still has some room for improvement. This limitation is not explicitly implied in our analysis since we just show the ideal case and the worst case. That says, the connection between the degree of possible parallelization and the properties of underlying MDPs and feature space needs to be further explored. We leave it as a future work.

VI. Conclusion

We consider parallel LSPI, which is inspired by recent success in parallel optimization. Our method allows each worker to execute a sufficient number of episodes before parameter averaging. Parameter averaging is conducted upon all workers updating their estimated parameters for each iteration. We also provide some analysis of the method. We show that parallel LSPI can enjoy an improved rate from $\mathcal{O}(1/\sqrt{n})$ towards $\mathcal{O}(1/\sqrt{Mn})$ for each worker. Further theoretical analysis needs to be explored more.

There are a few potential future works. For example, parallelizing LSPI with regularization can be considered. Though the features we used for the domains in our experiments are already sparse, in some situation where high dimensions of features is used, one may resort to l_1 regularization. There are growing number of related works about sparse least squares temporal difference learning [2], [10], [8], [14], [6], [19]. It is interesting to investigate how to combine the different ideas into parallel LSPI. Another improvements comes from reducing the communication overhead during learning. There is some related works in distributed optimization literature that tackles this problem, such as [20]. Moreover, distributed variants of other existing reinforcement learning methods can be considered as well.

VII. Acknowledgement

This work was supported by the Ministry of Science and Technology Taiwan, ROC, under grants MOST-105-2218-E-002-013. This material was also based upon work supported by the Air Force Office of Scientific Research, Asian Office of Aerospace Research and Development (AOARD) under award number FA2386-15-1-4013 and Microsoft Research Asia (MSRA) under award number FY16-RES-THEME-013.

References

[1] Craig Boutilier. “Sequential optimality and coordination in multiagent systems”. IJCAI, 1999.

[2] Amir Massoud Farahmand and Mohammad Ghavamzadeh and Csaba Szepesvri and Shie Mannor. “Regularized Policy Iteration“. Advances in Neural Information Processing Systems 20, 2008 Journal of Machine Learning Research, 15:1111–1133, 2014.

[3] Daniela Pucci de Farias and Benjamin Van Roy. “The linear programming approach to approximate dynamic programming“. Operations Research(51):850–865, 2003

[4] Victor Gabillon and Mohammad Ghavamzadeh and Bruno Scherrer. “Approximate Dynamic Programming Finally Performs Well in the Game of Tetris“. Approximate Dynamic Programming Finally Performs Well in the Game of Tetris, 2013

[5] Carlos Guestrin and Geoffrey Gordon. “Distributed Planning in Hierarchical Factored MDPs”. Uncertainty in Artificial Intelligence 18, 2002.

[6] Matthieu Geist and Bruno Scherrer and Alessandro Lazaric and Mohammad Ghavamzadeh. “A Dantzig Selector Approach to Temporal Difference Learning“. International Conference on Machine Learning 29, 2012

[7] Matthieu Geist and Bruno Scherrer and Alessandro Lazaric and Mohammad Ghavamzadeh. “A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning“. Foundations and Trends in Machine Learning(6)4:375–451, 2013.

[8] Jeffrey Johns and Christopher Painter-wakefield and Ronald Parr. “Linear Complementarity for Regularized Policy Evaluation and Improvement“. Advances in Neural Information Processing Systems 23, 2010.

[9] Matthew Kretchmar. “Parallel reinforcement learning”. The sixth World conference on Systemics, Cybernetics, and Informatics, 2002.

[10] Kolter, J. Zico and Ng, Andrew Y. “Regularization and Feature Selection in Least-squares Temporal Difference Learning“. International Conference on Machine Learning 26, 2009.

[11] Alessandro Lazaric and Mohammad Ghavamzadeh and Rémi Munos. “Finite-Sample Analysis of Least-Squares Policy Iteration “. Journal of Machine Learning Research (JMLR) (13)3041–3074, Oct. 2012.

[12] Lagoudakis, Michail G. and Parr, Ronald. “Least-squares Policy Iteration“. Journal of Machine Learning Research (JMLR)(4):1107–1149, Dec. 2003.

[13] John Langford and Alex. J. Smola and Martin Zinkevich. “Slow learners are fast“. Advances in Neural Information Processing Systems 22, 2009.

[14] Bo Liu and Mahadevan, Sridhar and Liu, Ji. “Regularized Off-Policy TD-Learning“. Advances in Neural Information Processing Systems 25, 2012.

[15] Li, Lihong and Littman, Michael L. and Mansley, Christopher R. “Online Exploration in Least-squares Policy Iteration”. Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems, 2009.

[16] Mu Li and Tong Zhang and Yuqiang Chen and Alex Smola. “ Efficient Mini-batch Training for Stochastic Optimization”. ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2010.

[17] Yucheng Low and Joseph Gonzalez and Aapo Kyrola and Danny Bickson and Carlos Guestrin and Joseph M. Hellerstein “GraphLab: A New Parallel Framework for Machine Learning”. UAI, 2010.

[18] Odalric-Ambrym Maillard and Rémi Coulom and Philippe Preux. “Parallelization of the TD(λ) Learning Algorithm”. The Seventh European Workshop on Reinforcement Learning, 2005.

[19] Zhiwei Qin and Weichang Li and Firdaus Janoos “ Sparse Reinforce-

ment Learning via Convex Optimization“. International Conference on Machine Learning 31, 2014.

[20] Ho, Qirong and Cipar, James and Cui, Henggang and Lee, Seunghak and Kim, Jin Kyu and Gibbons, Phillip B. and Gibson, Garth A. and Ganger, Gregory R. and Xing, Eric P.. “ More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server“. Advances in Neural Information Processing Systems 26, 2013.

[21] Yuxi Li and Dale Schuurmans. “MapReduce for Parallel Reinforcement Learning”. The Ninth European Workshop on Reinforcement Learning, 2011.

[22] Csaba Szepesvári. “Algorithms for Reinforcement Learning”. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.

[23] Sutton, Richard S. and Barto, Andrew G. “Introduction to Reinforcement Learning”. MIT Press. 1998.

[24] Powell, Warren B. “Approximate Dynamic Programming: Solving the Curses of Dimensionality (Wiley Series in Probability and Statistics)”. Wiley-Interscience, 2007.

[25] Yasin Abbasi-Yadkori and Peter Bartlett and Alan Malek. “ Linear Programming for Large-Scale Markov Decision Problems”. International Conference on Machine Learning 31, 2014