MapReduce for Parallel Reinforcement Learning

Yuxi Li[1] and Dale Schuurmans[2]

[1] College of Computer Science and Engineering
Univ. of Electronic Science and Technology of China
Chengdu, China
[2] Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada

Abstract. We investigate the parallelization of reinforcement learning algorithms using MapReduce, a popular parallel computing framework. We present parallel versions of several dynamic programming algorithms, including policy evaluation, policy iteration, and off-policy updates. Furthermore, we design parallel reinforcement learning algorithms to deal with large scale problems using linear function approximation, including model-based projection, least squares policy iteration, temporal difference learning and recent gradient temporal difference learning algorithms. We give time and space complexity analysis of the proposed algorithms. This study demonstrates how parallelization opens new avenues for solving large scale reinforcement learning problems.

1 Introduction

Reinforcement learning (RL) can solve a wide range of problems in science, engineering and economics that are modeled as Markov or Semi-Markov Decision Processes (MDPs) [1, 14, 16]. For large problems, however, one encounters the "curse of dimensionality". In such cases, function approximation, and in particular, linear function approximation, have proved to be a critical strategy for generalizing and scaling-up. Significant research effort is still devoted to developing efficient algorithms for large-scale problems.

For large problems, parallelism appears to be another promising approach to scaling up, particularly since multi-core, cluster and cloud computing are becoming increasingly the norm. Among parallel computing frameworks, MapReduce [5] has recently been attracting much attention in both industry and academia. There are numerous successes of MapReduce [4, 8, 10], including applications for machine learning and data mining problems. Among these, the most prominent is the complete rewrite of the production indexing system for the Google web search service. However, there has yet to be significant research effort on designing MapReduce parallel algorithms for reinforcement learning.

In this paper, we propose MapReduce algorithms to parallelize several important reinforcement learning algorithms. We identify maximum, vector, and matrix operations, such as matrix-vector multiplications, as the core enabling techniques for parallelizing reinforcement learning algorithms in MapReduce. Using these primitives, we show how MapReduce parallel algorithms can be designed for classical dynamic programming (DP) algorithms, including policy evaluation, policy iteration and off-policy updates, as well as tabular reinforcement learning algorithms. To cope with large scale problems via linear function approximation, we show how MapReduce parallel algorithms can be developed for approximate reinforcement learning algorithms, including the model-based projection method, least squares policy iteration, temporal difference (TD) learning, and the recent gradient TD algorithms. We present time and space complexity analysis for the proposed algorithms.

With such MapReduce reinforcement learning algorithms, one can handle complex sequential decision problems more efficiently by exploiting parallelism. MapReduce reinforcement learning algorithms are able to provide solutions to problems that would be infeasible otherwise. For example, it is mentioned in [18] that for the game of Go [15] over one million features were used when building a value function, making least-squares methods infeasible. However, parallelism can address such issues. Moreover, with MapReduce the programmer does not have to explicitly manage the complexity of underlying parallel/distributed issues, such as data distribution, load balancing and fault tolerance. In fact, parallel computing broadens the range of problems that can be feasibly tackled with tabular reinforcement learning methods, athough it enhances function approximation methods as well, as we will demonstrate.

The remainder of the paper is organized as follows. Section 2 first provides a brief introduction of MapReduce and discuss a design of matrix-vector multiplication. Then in Section 3, we discuss MapReduce algorithms for DP algorithms and RL algorithms in the tabular setting. Finally, we discuss MapReduce algorithms for RL with linear function approximation in Section 4, and conclude.

2 MapReduce

MapReduce is a framework introduced by Google for handling large data sets via distributed processing on clusters of commodity computers. It allows one to express parallel computations without worrying about the messy details of parallelism, including data distribution, load balancing and fault tolerance.

In this model, computation is expressed as taking a set of input (key, value) pairs and producing a set of output (key, value) pairs via two functions: Map and Reduce. The Map function takes an input pair and produces a set of intermediate (key, value) pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the Reduce function. The Reduce function accepts an intermediate key and a set of values for that key, and merges these values to form a possibly smaller set of values. Both the Map and Reduce functions are specified by the user. It is possible to further improve performance via partition and combine functions. The partition function divides the intermediate key space and assigns them to reducers. The combine function allows for local aggregation before the shuffle and sort phase.

Iterative MapReduce. Most RL algorithms are iterative by nature. There are several research efforts to design and implement iterative MapReduce, e.g., Haloop [3] and Twister [6]. Haloop extends Hadoop [20] for iterative programs by providing new programming model and architecture, loop-aware scheduling, caching for loop-invariant data and caching to support fixpoint evaluation.

MapReduce for matrix-vector multiplication. In the following, we present a MapReduce implementation for multiplying an $M \times N$ matrix $\{a_{i,j}\}$ with an $N \times 1$ vector $\{b_i\}$. The basic idea follows. We need two MapReduce jobs for the multiplication. In MapReduce Step 1, we multiply the j-th vector element b_j with each element $a_{i,j}$ of the j-th column of the matrix to obtain a set of $(key : i, value : b_j \times a_{i,j})$ pairs, for $i \in \{1, 2, \ldots, M\}$; In Step 2, we sum up values with key i to get the i-th vector element. Figures 1 and 2 present the MapReduce pseudo code for matrix-vector multiplication. The sorting phase dominates the algorithms complexity. Assuming the load is uniformly distributed on P processing units, the running time complexity is $O(\frac{MN}{P} log(\frac{MN}{P}))$, and the space complexity is $O(\frac{MN}{P})$.

```
Input Matrix $M = \{(i, (j, mval))\}$,
        Vector $V = \{(i, vval)\}$
Output Vector $V = \{(i, mval * vval)\}$

MV-Map1(Key $i$, Value $v$)
if ($k$,$v$) is of type Vector then
    Output($i, v$)
end if
if ($k$,$v$) is of type Matrix then
    $(j, mval) \leftarrow v$
    Output($j, (i, mval)$)
end if

MV-Reduce1(Key $j$, Value $v'[1..m]$)
$mv \leftarrow []$
$vv \leftarrow []$
for $v$ in $v'[1..m]$ do
    if ($k$,$v$) is of type Vector then
        $vv \leftarrow v$
    else
        if ($k$,$v$) is of type Matrix then
            Add $v$ to $mv$;
        end if
    end if
end for
for $(i', mval')$ in $mv$ do
    Output($i', mval' * vv$)
end for
```

Fig. 1. Matrix-Vector multiplication, step 1

```
Input Partial vector $V' = \{(i, vval')\}$
Output Result vector $V = \{(i, vval)\}$

MV-Map2(Key $i$, Value $v$)
Output($i, v$)

MV-Reduce2(Key $i$, Value $v[1..m]$)
$sum \leftarrow 0$
for $v'$ in $v[1..m]$ do
    $sum \leftarrow sum + v'$
end for
Output($i, v$)
```

Fig. 2. Matrix-Vector multiplication, step 2

Such implementation assumes that a processing unit can not multiply a row of the matrix and the vector. Otherwise, a single map function can handle the matrix-vector multiplication. In the sequel, we decompose matrix operations into matrix-vector multiplications; and we use vector and maximum operations directly, since they are straightforward to parallelize in MapReduce. See more discussions about designing matrix operations in [8, 11].

3 MapReduce for Tabular DP and RL

First, we introduce the notation we will use. An MDP is defined by a finite set of states S, a finite set of actions A, an $|S||A| \times |S|$ transition matrix P and an $|S||A| \times 1$ reward vector $\mathbf{r}$. The entry $P_{(sa,s')}$ specifies the conditional probability of transiting to state s' starting from state s and taking action a. The entry $\mathbf{r}_{(sa)}$ specifies the reward obtained when taking action a in state s. A standard objective is to maximize the infinite horizon discounted reward $\sum_{t=1}^{\infty} \gamma^{t-1} r_t$. In this case, one can always have an optimal stationary, deterministic policy [1], denoted as $\boldsymbol{\pi}$. The entry $\pi_{(sa)}$ specifies the probability of taking action a in state s. A policy is stationary if the selection probability does not change over time. In a deterministic policy, there is an optimal action for each state, i.e., the probabilities are either 0 or 1. For convenience, let Π denote an $|S| \times |S||A|$ matrix with $\Pi(s,(s,a)) = \pi(s,a)$.

3.1 Policy evaluation

We have $\mathbf{v} = \Pi(\mathbf{r} + \gamma P \mathbf{v})$ for state value function $\mathbf{v}$. A state-based policy evaluation algorithm can be defined by repeatedly applying an operator $\mathcal{O}$ defined by $\mathcal{O}\mathbf{v} = \Pi(\mathbf{r}+\gamma P\mathbf{v}) = \Pi\mathbf{r}+\gamma\Pi P\mathbf{v}$. For a fixed policy $\boldsymbol{\pi}$, a MapReduce policy evaluation algorithm can first calculate $\tilde{\mathbf{v}} = \Pi\mathbf{r}$ and $H = \Pi P$ using parallel matrix-vector and matrix-matrix multiplications respectively. Then $\mathbf{v} \leftarrow \tilde{\mathbf{v}}+\gamma H\mathbf{v}$ can be iteratively calculated until convergence, which includes iterative calls of parallel operations for matrix-vector multiplication and vector-vector addition. State-action policy evaluation can be dealt with similarly.

For calculating $H = \Pi P$, the time complexity is $O(|S|\frac{|S|^2|A|}{P} log(\frac{|S|^2|A|}{P}))$, and the space complexity is $O(\frac{|S|^2|A|}{P})$. For the iteration, $\mathbf{v} \leftarrow \tilde{\mathbf{v}} + \gamma H\mathbf{v}$, the time complexity is $O(T\frac{|S|^3}{P} log(\frac{|S|^3}{P}))$, where T is the number of iterations; and the space complexity is $O(\frac{|S|^3}{P})$. Thus the complexity of the whole policy evaluation depends on which part dominates.

One can also consider an iterative MapReduce implementation of policy evaluation using Haloop [3], as shown in Figure 3. In Haloop, one has the additional functionality: AddMap and AddReduce to specify Map and Reduce function(s) in each iteration; SetInput to specify the input to each iteration; AddInvariantTable to specify loop-invariant data; SetDistanceMeasure to specify a distance for results for fixed-point check; and SetFixedPointThreshold and/or SetMaxNumOfIterations to specify the loop termination condition. Thus, for policy evaluation one can first calculate $H = \Pi P$ using MapReduce matrix-matrix multiplication; this remains invariant across iterations. Then we introduce a map function (without reduce) to update $\mathbf{v}$ after the matrix-vector multiplication $\mathbf{v}' = H\mathbf{v}_{i-1}$

```
**IterationInput**
**Input**: int i
return the $i$-th column of matrix $P$
**Main**
Job job = new Job()
add job for matrix-vector multiplication
job.AddInvariantTable($\Pi$)
job.SetInput(IterationInput)
job.setMaxNumOfIteration($|S|$)
job.Submit()

(Above: Calculating $H = \Pi$ P)

**IterationInput**
**Input**: int i
return the $(i-1)$-th step of $\mathbf{v}$

**UpdateV-Map**
**Input**: Key k, vector $\mathbf{r}$, constant $\gamma$, $\mathbf{v}'$
Output(0, $\mathbf{v} = \tilde{\mathbf{v}} + \gamma \mathbf{v}'$)
**ResultDistance**
Input: $\mathbf{v}_{i-1}$, $\mathbf{v}_i$
return $\|\mathbf{v}_{i-1} - \mathbf{v}_i\|$
**Main**
Job job = new Job()
add job for matrix-vector multiplication
job.AddMap(UpdateQ-Map)
job.AddInvariantTable($M$)
job.SetInput(IterationInput)
job.SetDistanceMeasure(ResultDistance)
job.setFixedPointThreshold($\epsilon$)
job.setMaxNumOfIteration($T$)
job.Submit()
```

Fig. 3. Haloop-based policy evaluation

in the main function.[3] IterationInput specifies that the most recent $\mathbf{v}_{i-1}$ as the input to each iteration; see Figure 3. We use the L_2-norm to calculate the difference of consecutive $\mathbf{v}$'s to test convergence, as specified in ResultDistance. We set the number of iterations and the fixed point threshold respectively as T and ϵ, two predefined numbers.

3.2 Policy iteration

Policy iteration is a standard MDP planning algorithm that iteratively conducts policy evaluation and policy improvement. In policy improvement, given a current policy $\boldsymbol{\pi}$, whose state value function $\mathbf{v}$ or state-action value function $\mathbf{q}$ have already been determined, one can obtain an improved policy $\boldsymbol{\pi}'$ by setting: $a^*(s) = \arg\max_a \mathbf{q}_{(sa)} = \arg\max_a \mathbf{r}_{(sa)} + \gamma P_{(sa,:)}\mathbf{v}$, $\boldsymbol{\pi}'_{(sa)} = 1$, if $a = a^*(s)$; or, 0, if $a \neq a^*(s)$. The policy improvement theorem verifies that the above update leads to an improved policy; that is, $\Pi \mathbf{q} \leq \Pi' \mathbf{q}$ implies $\mathbf{v} \leq \mathbf{v}'$.

It is straightforward to design parallel algorithms for policy improvement in MapReduce. For example, given $\mathbf{q}$, for each state s, one finds the maximum state-action values $\mathbf{q}$ for all actions. Similarly, given $\mathbf{v}$, for each state s and action a, one calculates $\mathbf{r}_{(sa)} + \gamma P_{(sa,:)}\mathbf{v}$, which involves a vector-vector multiplications and an addition. Then, for state s, we find the maximum over the resulting values for all actions.

Vector-vector multiplication, vector-vector addition and maximum operators are linear in both time and space. Thus, policy evaluation dominates time and space complexity in policy iteration.

[3] Note, to make the implementation more efficient, one should integrate this update step with the step for matrix-vector multiplication. We present it in this separated manner for clarity.

3.3 Off-policy updates

Off-policy updates form the basis for many RL algorithms, in particular, value iteration and Q-learning. For a state value function $\mathbf{v}$, the off-policy operator $\mathcal{M}$ is defined as $\mathcal{M}\mathbf{v} = \Pi^*(\mathbf{r}+\gamma P\mathbf{v})$, where,$\Pi^*(\mathbf{r}+\gamma P\mathbf{v})_{(s)} = \max_a(\mathbf{r}_{(sa)} + \gamma P_{(sa,:)}\mathbf{v})$ For a state-action value function $\mathbf{q}$, the off-policy operator $\mathcal{M}$ is defined as $\mathcal{M}\mathbf{q} = \mathbf{r} + \gamma P\Pi^*\mathbf{q}$, where, $\Pi^*\mathbf{q}_{(s)} = \max_a \mathbf{q}_{(sa)}$. The above off-policy updates aim at making the corresponding value functions closer to their respective Bellman equations; namely, $\mathbf{v} = \Pi^*(\mathbf{r}+\gamma P\mathbf{v})$ and $\mathbf{q} = \mathbf{r}+\gamma P\Pi^*\mathbf{q}$. It is straightforward to design parallelized algorithms for off-policy updates using MapReduce's vector-vector multiplication and max operation, which have linear complexity.

3.4 Tabular online algorithms

There are several tabular online algorithms, such as, TD(λ), Sarsa(λ), and Q(λ)-learning, for $\lambda \in [0,1]$, where each online sample updates one entry of the value function, and the current update is affected by previous updates. For simplicity, we give updates for TD(0), Sarsa(0) and Q(0)-learning respectively.

$$\begin{aligned} V(s) &\leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)] \\ Q(s,a) &\leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)] \\ Q(s,a) &\leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \end{aligned}$$

For large-scale problems, we can take advantage of parallel computing: in Q-learning for picking the action that gives the largest state-action value given the state; and in Sarsa and Q-learning for selecting an action given the current state and state-action value function, for example, with ϵ-greedy policy. Both involve a maximum operation. These operations have linear complexity.

4 MapReduce for RL: Linear Function Approximation

In the linear architecture, the approximate value function is represented by: $\hat{Q}(s,a;\mathbf{w}) = \sum_{i=1}^k \phi_i(s,a)w_i$, where $\phi_i(\cdot,\cdot)$ is a basis function, w_i is its weight, and k is the number of basis functions. Define

$$\begin{aligned} \phi^{\mathsf{T}}(s,a) &= \{\phi_1(s,a), \phi_2(s,a), \ldots, \phi_k(s,a)\}, \\ \Phi^{\mathsf{T}} &= \{\phi(s_1,a_1), \ldots, \phi(s,a), \ldots, \phi(s_{|S|}, a_{|A|})\}, \\ \mathbf{w}^{\mathsf{T}} &= \{w_1, w_2, \ldots, w_k\}. \end{aligned}$$

where T denotes matrix transpose. Then we have $\hat{Q} = \Phi\mathbf{w}$.

4.1 Model-based Projection

Define an operator $\mathcal{P}$ that projects a state-action value function $\mathbf{q}$ to the column span of Φ, $\mathcal{P}\mathbf{q} = \mathrm{argmin}_{\hat{q}\in span(\Phi)} \|\mathbf{q} - \hat{\mathbf{q}}\|_{\mathbf{z}} = \Phi(\Phi^{\mathsf{T}}Z\Phi)^{-1}\Phi^{\mathsf{T}}Z\mathbf{q}$. Here, $Z = \mathrm{diag}(\mathbf{z})$, where $\mathbf{z}$ is the stationary state-action visit distribution for $P\Pi$.

Approximate dynamic programming composes the on-policy operator $\mathcal{O}$ and the subspace projection operator $\mathcal{P}$ to compute the best approximation of one-step update of the value function representable with the basis functions. Such combined operator is guaranteed to converge to a fixed point.

We discuss how to parallelize such a projection. Since $\Phi(\Phi^\mathsf{T} Z\Phi)^{-1}\Phi^\mathsf{T} Z$ remains fixed, we can calculate it using a matrix inversion and several matrix-matrix multiplications. A matrix inversion can be calculated with singular value decomposition (SVD)[4], whose MapReduce implementation is available in the open source Apache Hama project. In the following, we discuss matrix inversion with MapReduce based on the algorithm in [2].

We first describe briefly the fast iterative matrix inversion algorithm in [2, Section 2.9]. Suppose A is an invertible square matrix. The algorithm is based on the classical Newton method, which is motivated by the desire of low computational complexity and good numerical robustness. The algorithm starts with some B_0 such that $\|I - B_0A\| < 1$. Then it iteratively improve B_k by letting $B_{k+1} = 2B_k - B_kAB_k$. This iteration can be interpreted as solving $X^{-1} - A = 0$ with Newton's method. A choice for B_0 is $B_0 = A^\mathsf{T}/\mathrm{tr}(A^\mathsf{T}A)$, where $\mathrm{tr}(A^\mathsf{T}A)$ is the trace of $A^\mathsf{T}A$, the sum of the diagonal entries of $A^\mathsf{T}A$. It can be shown that such a choice for B_0 is suitable and the above iterative method is efficient.

Each iteration involves matrix multiplications and subtraction, so it is straightforward to parallelize in MapReduce. Since $\Phi^\mathsf{T} Z\Phi$ is a square matrix, we can use this iterative method to invert it. For calculating $\Phi^\mathsf{T} Z\Phi$, the time complexity is $O((k+|S||A|)\frac{k|S||A|}{P}log(\frac{k|S||A|}{P}))$, and the space complexity is $O(\frac{k|S||A|}{P})$, assuming we use matrix-vector multiplication. For each iteration, the time complexity is $O(\frac{k^5}{P}log(\frac{k^3}{P}))$, and the space complexity is $O(\frac{k^3}{P})$. The complexity can be improved by customizing a MapReduce algorithm for matrix multiplication.

4.2 Least-squares policy iteration

LSPI [9] combines the data efficiency of the least squares temporal difference (LSTD) method and the policy search efficiency of policy iteration. As shown in [9], for a given $\boldsymbol{\pi}$, the weighted least squares fixed point solution is: $\mathbf{w} = \left(\Phi^\mathsf{T}\Delta_\mu(\Phi - \gamma P\Pi\Phi)\right)^{-1}\Phi^\mathsf{T}\Delta_\mu R$, where Δ_μ is the diagonal matrix with the entries of $\mu(s,a)$, which is a probability distribution over state-action pairs $(S\times A)$. This can be written as $A\mathbf{w} = b$, where $A = \Phi^\mathsf{T}\Delta_\mu(\Phi - \gamma P\Pi\Phi)$ and $\mathbf{b} = \Phi^\mathsf{T}\Delta_\mu\mathbf{r}$.

Without a model of the MDP—that is, without the full knowledge of P, Π and $\mathbf{r}$—one needs a learning method to discover an optimal policy. It is shown in [9] that $\mathbf{A}$ and $\mathbf{b}$ can be learned incrementally as, for a sample (s,a,r,s'):

$$A \leftarrow A + \phi(s,a)(\phi(s,a) - \gamma\phi(s',\pi(s')))^\mathsf{T}$$
$$\mathbf{b} \leftarrow \mathbf{b} + \phi(s,a)r$$

[4] For a real number matrix A, if its singular value decomposition is $A = U\Sigma V^\mathsf{T}$, the pseudo-inverse is $A^+ = V\Sigma^+U^\mathsf{T}$. To get the pseudoinverse of the diagonal matrix Σ, we first transpose Σ, and then take the reciprocal of each non-zero element on the diagonal, and leave the zeros intact.

```
**Input:** $samples = \{(s, a, r, s')\}$
**Output:** $A$ and $\mathbf{b}$
**LSPI-Ab-Map**(Key $k$, Value $v$)
$A \leftarrow []$; $\mathbf{b} \leftarrow []$
**for** $(s, a, r, s')$ in samples **do**
    $\phi_0 \leftarrow \phi(s, a)$
    $\phi_d \leftarrow \phi_0 - \gamma\phi(s', \pi(s'))$
    **for** $i \leftarrow 1$ **to** $k$ **do**
        $A(i, :) \leftarrow A(i, :) + \phi_0(i)\phi_d^\top$
    **end for**
    $\mathbf{b} = \mathbf{b} + \phi_0 r$
**end for**
Output(1,$\{A, \mathbf{b}\}$)
```

```
**LSPI-Ab-Reduce**(Key $k$, Value $Ab'$)
$A \leftarrow []$; $\mathbf{b} \leftarrow []$
**for** $(A', \mathbf{b}')$ in $Ab'$ **do**
    $A \leftarrow A + A'$
    $\mathbf{b} \leftarrow \mathbf{b} + \mathbf{b}'$
**end for**
Output $(0, \{A, \mathbf{b}\})$

**Input:** Samples $Samples$
**Output:** Weight vector $\mathbf{w}$
**Main**
**while** $\mathbf{w}$ not convergent **do**
    Calculate $A, b$
    Calculate $D^{-1}b, D^{-1}B$
    Calculate $\mathbf{w}$ (Figure 5)
**end while**
```

Fig. 4. Parallel LSPI with MapReduce

With the new weight vector $\mathbf{w}'$ a new policy is obtained. Thus, one can iteratively improve the policy until convergence.

To parallelize LSPI in MapReduce, consider the following. To calculate A and $\mathbf{b}$, one can parallelize the algorithm over samples (s_t, a_t, r_t, s'_t). Moreover, there is a special structure in $\phi(s_t, a_t)(\phi(s_t, a_t) - \gamma\phi(s'_t, \pi(s'_t)))^\top$; that is, it is a vector-vector multiplication, resulting in a matrix. Thus, we can design a parallel algorithm which guarantees that each element in one vector multiplies each element in another vector. In this way, we obtain matrix A. The time complexity is $O(\frac{k^2}{P}log(\frac{k^2}{P}))$, and the space complexity is $O(\frac{k^2}{P})$. It is straightforward to obtain vector $\mathbf{b}$, since it involves only vector-scalar multiplication.

In Figure 4, we provide a MapReduce algorithm to calculate A and $\mathbf{b}$, then present an implementation for LSPI. We divide the (s, a, r, s') samples into groups. Each MapReduce job handles one group of samples. We use $\phi(s, a)$ to denote the value vector of basis functions for state-action pair (s, a), which involves an underlying evaluation of basis functions. We output the key as 0 and value as a pair of $\{A, \mathbf{b}\}$ so that all outputs (that is, the partial results of A and $\mathbf{b}$) will go to a single reduce function. The reduce function collects partial results of A and $\mathbf{b}$ and sums them, respectively For LSPI, we choose not to use Haloop since the current version cannot support iterations with "loop-within-loop". This remains an interesting potential future extension of Hadoop; namely, a framework to support "loop-within-loop" iterations.

To solve $A\mathbf{w} = \mathbf{b}$, we note that the obvious solution of computing $\mathbf{w} = A^{-1}\mathbf{b}$ may be inefficient. Instead, we deploy Jacobi iteration to solve this linear system. This can be done efficiently in the proposed framework as follows. Set $A = B+D$, where D is a diagonal matrix, making it easy to obtain $\mathbf{w} = D^{-1}(\mathbf{b} - B\mathbf{w})$. We then obtain the following iteration:

$$\mathbf{w}_{t+1} = D^{-1}(\mathbf{b} - B\mathbf{w}_t)$$

```
Update-w-Map
Input: Key k, vectors $D^{-1}b$, $\mathbf{w}$
Output(1,$\mathbf{w} = D^{-1}b - \mathbf{w}$)
IterationInput
Input: int $i$
return the $(i-1)$-th step of $\mathbf{w}$
ResultDistance
Input: $\mathbf{w}_{i-1}$, $\mathbf{w}_i$
return $\|\mathbf{w}_{i-1} - \mathbf{w}_i\|$

Main
Job job = new Job()
add job for matrix-vector multiplication
job.AddMap(Update-w-Map)
job.SetInput(IterationInput)
job.SetDistanceMeasure(ResultDistance)
job.setFixedPointThreshold($\epsilon$)
job.setMaxNumOfIteration($T$)
job.Submit()
```

Fig. 5. Solve $Aw = b$

Note that $D^{-1}b$ and $D^{-1}B$ each only need to be calculated once, and each is easy since D is diagonal. The time and space complexity will be linear in k. Thus, the matrix inversion problem $\mathbf{w} = A^{-1}b$ is converted into iterations that require only matrix-vector multiplications. Thus, for each iteration, the time complexity is $O(\frac{k^2}{P}log(\frac{k^2}{P}))$, and the space complexity is $O(\frac{k^2}{P})$.

In Figure 5, we give an iterative MapReduce implementation in the Haloop style for calculating $\mathbf{w}$ after calculating A and b. As above, $D^{-1}b$ and $D^{-1}B$ are given so we can add a map function (without reduce) to update $\mathbf{w}$ after the matrix-vector multiplication $\mathbf{w}' = D^{-1}B\mathbf{w}_{i-1}$. IterationInput specifies that the most recent $\mathbf{w}_{i-1}$ as the input to each iteration. We use the L_2-norm to calculate the difference of consecutive $\mathbf{w}$'s to test the convergence, as specified in ResultDistance. We set the number of iterations as T, and the fixed point threshold as ϵ, two predefined numbers.

An alternative method for solving $Ax = b$ is to use a conjugate gradient method [7]. If A is symmetric and positive semi-definite, then we can apply the conjugate gradient method directly. Since A is usually not symmetric in our problem, we need to solve the normal equations, $A^{\mathsf{T}}Ax = A^{\mathsf{T}}b$. Unfortunately, the condition number $\kappa(A^{\mathsf{T}}A) = \kappa^2(A)$ might be significantly increased, which results in slow convergence. To address such a problem, choosing a good preconditioner would be important.

Such techniques form the basis for parallelizing similar least squares RL methods in MapReduce, e.g., the backward approach and the fitted-Q iteration in [19].

4.3 Temporal difference learning

In TD learning with linear function approximation, with k basis functions, we define $\phi^{\mathsf{T}}(s) = \{\phi_1(s), \cdots, \phi_k(s)\}$. The approximate value function is then given by $\hat{\mathbf{v}}(s) = \boldsymbol{\phi}(s)^{\mathsf{T}}\mathbf{w}$, where $\mathbf{w}$ is the weight vector.

The update procedure for approximate TD(0) is then

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t\phi(s_t)[r_t + \gamma\phi^{\mathsf{T}}(s_{t+1})\mathbf{w}_t - \phi^{\mathsf{T}}(s_t)\mathbf{w}_t]$$

With linear function approximation, TD(λ), $\lambda \in [0, 1]$, can be expressed by the update rule

$$d_t = r + \gamma\phi^{\mathsf{T}}(s_{t+1})\mathbf{w}_t - \phi^{\mathsf{T}}(s_t)\mathbf{w}_t$$

```
Input: Samples Samples
Output: Weight vector $\mathbf{w}$
for sample $(s, a, r, s')$ in Samples do
  Calculate $\phi(s)$, $\phi(s')$, $\phi(s') - \phi(s)$, $[\phi(s') - \phi(s)]\mathbf{w}_t$
  Calculate the scalar $d_t = r + [\phi(s') - \phi(s)]\mathbf{w}_t$
  Calculate $z_t = \gamma\lambda z_{t-1} + \phi(s)$
  Calculate $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t d_t z_t$
end for
```

Fig. 6. MapReduce TD(λ)

$$z_t = \gamma\lambda z_{t-1} + \phi(s_t)$$
$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t d_t z_t$$

where d_t is the temporal difference, z_t is the eligibility trace [16].

TD updates involve vector-vector additions, subtractions and multiplications (resulting in scalars), which are straightforward to parallelize in MapReduce. It is also straightforward to calculate basis functions for a state with MapReduce, which needs only a map function. It is particular interesting to parallelize TD learning algorithms when the feature dimension is huge, that is, when k is very large, for example, at the scale of millions in Go [15].

To design parallel TD(λ) MapReduce, we give the pseudo-code of a driver in Algorithm 6. In each iteration, it calculates the basis functions, the temporal difference d_t, the eligibility trace z_t and updates the weight vector $\mathbf{w}$ with samples (s, a, r, s'). We see that it is not complicated to design parallel TD(λ) in MapReduce. We choose to design a driver to implement the iterations in TD learning, since we realize that the current Haloop or Twister does not support the kind of iteration in TD. That is, in TD, before updating the weight vector $\mathbf{w}$, some operations are needed to calculate basis functions, d_t and z_t, which themselves need MapReduce operations, and the inputs to each of them are different. A future work is to extend Hadoop to support more general iterations, for example, that for TD learning.

Significantly, there are a series of recent papers about gradient TD algorithms, e.g. [17, 13], addressing the instability and divergence problem of function approximation with off-policy TD learning/control. In the following, we give TDC [17](linear TD with gradient correction) algorithm directly.

$$d_t = r + \gamma\phi^{\mathsf{T}}(s_{t+1})\mathbf{w}_t - \phi^{\mathsf{T}}(s_t)\mathbf{w}_t$$
$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t d_t \phi(s_t) - \alpha_t \gamma \phi(s_{t+1})(\phi^{\mathsf{T}}(s_t)\eta_t)$$
$$\eta_{t+1} = \eta_t + \beta_t(d_t - \phi^{\mathsf{T}} s_t)\eta_t)\phi(s_t)$$

We can design parallel TDC algorithm with MapReduce similar to that for TD algorithm, since the updates involve vector-vector additions, subtractions and multiplications (resulting in scalars), so we do not discuss it further. As well, other gradient TD algorithms can be dealt with similarly.

The above parellelization of TD and gradient TD algorithms have linear time and space complexity in the number of basis functions, and thus are suitable for problems with a large dimension or with a large number of features, e.g., for the game of Go [15], over a million features are used when building a value function.

Recently, Zinkevich et al. propose a parallelized stochastic gradient descent algorithm and gave theoretical analysis [21]. Both TD and gradient TD algorithms can exploit such an algorithm for parallelization, so that the parallel learning algorithm collects many sets of samples, then assigns each processing unit to conducting learning with TD or gradient TD update rules, after that, takes the average as the final result. Such an approach of parallelization assumes that a TD or gradient TD can be handled by a processor, and averages results from many processors to achieve high efficiency and accuracy.

5 Conclusions

We have investigated techniques for parallelizing reinforcement learning algorithms with MapReduce. In particular, we have provided parallel dynamic programming algorithms with MapReduce, including policy evaluation, policy iteration and off-policy updates, as well as tabular reinforcement learning algorithms. Furthermore, we proposed parallel algorithms with MapReduce for reinforcement learning, to cope with large scale problems with linear function approximation; namely the model-based projection method, least squares policy iteration, temporal difference learning, and the very recent gradient TD algorithms. We emphasize that iterative MapReduce is critical for parallel reinforcement learning algorithms, and provided algorithms in the Haloop style for tabular policy evaluation and for solving a linear system $Aw = b$. We give time and space complexity analysis of the proposed algorithms. These algorithms show that reinforcement learning algorithms can be significantly parallelized with MapReduce and open new avenues for solving large scale reinforcement learning problems.

We also observe that the current Hadoop and its iterative proposals, including Haloop and Twister, are not general enough to support certain iteration styles in a natrual way, for example, for LSPI and TD(λ). It is desirable to extend Hadoop for more general iterative structures, for example, the "loop-in-loop" iterations. Our preliminary experiments show that a cluster of machines running Hadoop can solve large scale linear systems $Aw = b$ efficiently; while Matlab on a single computer can easily encounter "Out of memory". It is desirable to further study the performance of proposed parallel DP and RL algorithms with MapReduce empiricaly. It would also be interesting to study alternative parallel frameworks, e.g., GraphLab [12], which address potential inefficiency with MapReduce, e.g., for problems with data-dpendancy due to MapReduce's share-nothing feature.

References

[1] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Massachusetts, USA, 1996.

[2] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Athena Scientific, Massachusetts, USA, 1997.

[3] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. In *The 36th International Conference on Very Large Data Bases (VLDB'10)*. Singapore, September 2010.

[4] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19 (NIPS 2006)*, pages 281–288. Vancouver, December 2006.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplied data processing on large clusters. In *OSDI 2004*, pages 137–150. San Francisco, USA, December 2004.

[6] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative mapreduce. In *The First International Workshop on MapReduce and its Applications*. Chicago, USA, June 2010.

[7] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, Maryland, USA, 1996.

[8] U. Kung, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. In *ICDM 2009*, pages 229–238. Miami, December 2009.

[9] M. G. Lagoudakis and R. Parr. Least-squares policy iteration. *The Journal of Machine Learning Research*, 4:1107 – 1149, December 2003.

[10] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool, 2009.

[11] C. Liu, H.-C. Yang, J. Fan, L.-W. He, and Y.-M. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *Proceedings of the 19th International World Wide Web Conference (WWW'10)*, pages 681–690. Raleigh, North Carolina, USA, April 26C30, 2010.

[12] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Uncertainty in Artificial Intelligence (UAI)*. Catalina Island, USA, July 2010.

[13] H. R. Maei and R. S. Sutton. GQ(λ): A general gradient algorithm for temporal-difference prediction learning with eligibility traces. In *Proceedings of the Third Conference on Artificial General Intelligence*. Lugano, Switzerland, 2010.

[14] M. L. Puterman. *Markov decision processes : discrete stochastic dynamic programming*. John Wiley & Sons, New York, 1994.

[15] D. Silver, R. S. Sutton, and M. Muller. Reinforcement learning of local shape in the game of Go. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1053–1058. Hyderabad, India, Jan. 2007.

[16] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[17] R. S. Sutton, H. R. Maei, D. Precup, S. Bhatnagar, D. Silver, C. Szepesvari, and E. Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of The 26th International Conference on Machine Learning (ICML)*, pages 993–1000. Montreal, Canada, June 2009.

[18] C. Szepesvari. *Algorithms for Reinforcement Learning*. Morgan & Claypool, 2010.

[19] J. N. Tsitsiklis and B. Van Roy. Regression methods for pricing complex American-style options. *IEEE Transactions on Neural Networks (special issue on computational finance)*, 12(4):694–703, July 2001.

[20] T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2009.

[21] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. In *Proceedings of Advances in Neural Information Processing Systems 24 (NIPS 2010)*, pages 2217–2225. Vancouver, Canada, December 2010.