Parallelizing Reinforcement Learning

Jonathan T. Barron
barron@eecs.berkeley.edu

Dave S. Golland
dsg@eecs.berkeley.edu

Nicholas J. Hay
nickjhay@eecs.berkeley.edu

Abstract

Reinforcement learning is an important family of algorithms that have been extremely effective in fields such as robotics, economics, and artificial intelligence. Current algorithms become increasingly expensive as the state space of the problem increases in size. Additionally, computer architectures are becoming increasingly parallelized, and existing algorithms need to be reworked to fit within this parallelization paradigm. We will present a general framework for parallelizing such algorithms.

Many reinforcement learning problems (such as robot navigation) have state-spaces that correspond to real, physical spaces, in which states are physical locations and actions transition between neighboring locations. We will demonstrate how problems of this nature can be decomposed into smaller subproblems that can solved in parallel. We built two implementations of our framework, a MATLAB implementation for rapid prototyping (in which all parallelism is simulated) and a Java multicore implementation for which our reported runtimes are actual *parallel runtime. Our framework improves the runtime of policy iteration by up to* $3\times$ *on an simulated* 8 *core processor (MATLAB), and up to* $203\times$ *on an actual* 8 *core processor (Java).*

1. Introduction

Our work will be demonstrated on a simple navigation-style problem, similar to the classic grid-world problem, in which states are locations in a space. The agent nondeterministically transitions to one of several nearby locations following a distribution determined by the action it selects. There are absorbing states, some of which are assigned positive rewards, and some of which are assigned negative rewards by the reward function. Examples of such problems can be seen in Figures 2(a) and 2(d), and a (correctly) solved problem, in which we have estimated the utilities of all states and the optimal policy for all states, can be seen in Figure 1(b). In general, states near positive absorbing states have high utility in the solution, whereas states near negative absorbing states have low utility. The optimal policy directs the agent to the nearest positive absorbing state.

Problems where states lie in a low-dimensional space and adjacency is determined by a maximum distance can be nicely decomposed. Consider Figure 2(a), which we will call our "room/hallway" problem. In order to facilitate parallelization, we propose to solve this problem by decomposing the graph into densely connected "rooms", which we solve independently, and sparse "hallways", through which rooms can communicate as the algorithm converges on the correct answer. Even in Figures 2(d) (our "uniform random" problem), which is not as structured as the "room/hallway" problem, intuition dictates that parallelization can still benefit from decomposing the problem into subproblems.

In the below we will cover two implementations of this parallel algorithm. The first is a MATLAB prototype we used to simulate the performance of the parallel algorithm under various conditions. The second is a full parallel Java implementation.

Our Java implementation uses thread-based parallelism, and is aimed at multicore processors. Our experiments are on 8-core machines, although we could easily use massively multicore processors. This can naturally be extended to a distributed setting, with different machines being assigned different subregions of the state space (collections of rooms), machines whose subregions are connected by hallways communicating by message passing over the network.

2. Problem Decomposition

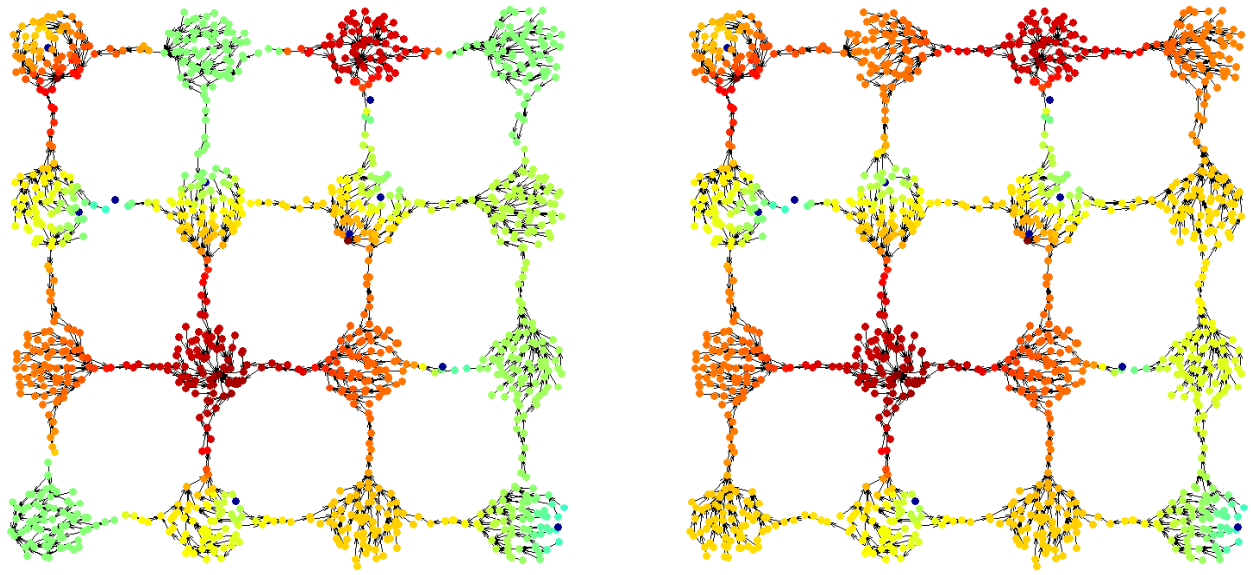
To decompose our problem, we will explore Normalized Cuts [5], a spectral clustering method that has seen widespread use in Computer Vision, and K-means [4], a standard clustering algorithm.

Normalized Cuts minimizes:

$$Ncut(A,B) = \frac{cut(A,B)}{assoc(A,V)} + \frac{cut(B,A)}{assoc(B,V)} \quad (1)$$

which roughly means that we seek to partition V into sets A and B such that the association between groups is minimized and the association within groups is maximized. A real-valued solution y is attained by solving the following generalized eigenvalue for the second smallest eigenvector:

$$(\mathbf{D} - \mathbf{W})y = \lambda \mathbf{D} y \quad (2)$$



(a) Naive parallel policy iteration

(b) Our parallel policy iteration method (virtually identical to non-parallelized policy iteration)

Figure 1. On the left, we see the failure of a naive attempt to parallelize reinforcement learning by simply solving the subproblems of a decomposed problem individually. On the right, we see the correct results of our parallelized reinforcement learning algorithm. The color of the node is proportional to utility (red = positive, blue = negative), and the arrows show the optimal policy (which is greedy with respect to the utilities).

Where $\mathbf{W}(i,j) = 1$ if state i can transition to state j, and 0 otherwise, and $\mathbf{D}$ is a diagonal matrix such that $D(i,i) = \sum_j W(i,j)$. We then find the binary cut along the direction of y such that $Ncut(A,B)$ is maximized. This can be done recursively $\log_2(k)$ times to produced k clusters. The resulting decomposition of our example problem into 16 clusters can be seen in Figures 2(b) and Figure 2(e).

For comparison, we also partition the problem using K-means. K-means clusters data by minimizing the Euclidian distance between the data and a set of cluster centroids. Results with $k = 16$ can be seen in Figures 2(c) and Figure 2(f).

Normalized Cuts does a much better job of finding the clustered structure in the problem because (unlike K-means) it takes advantage of the graph adjacency structure. K-means performs poorly because it frequently partitions the graph into subgraphs containing disjoint nodes or partitions the graph such that there are many inter-subgraph edges. On the other hand, Normalized Cuts aggressively finds room/hallway-like patterns, even if they are subtle. Examples of this behavior can be found in Figure 2.

In practice, parallelized policy iteration on a problem that has been decomposed with Normalized cuts is 10% to 200% faster than one decomposed with K-means, depending on the structure of the problem. We therefore use Normalized Cuts for all future experiments.

Note that our decomposition is based entirely on the graph structure and not on the reward function. Hence, the decomposition is fixed prior to seeing the reward function.

3. Reinforcement Learning

A Markov Decision Process is defined by five items: a set of states S, a set of actions A, transition probabilities $T(s,a,s')$, a reward function R, and a discount factor γ. Since in our problem setup, each state is assigned a reward, the standard Bellman optimality equation for calculating the optimal utility of a state $V^*(s)$ (the expected reward for starting in state s and behaving optimally) can be written as follows:

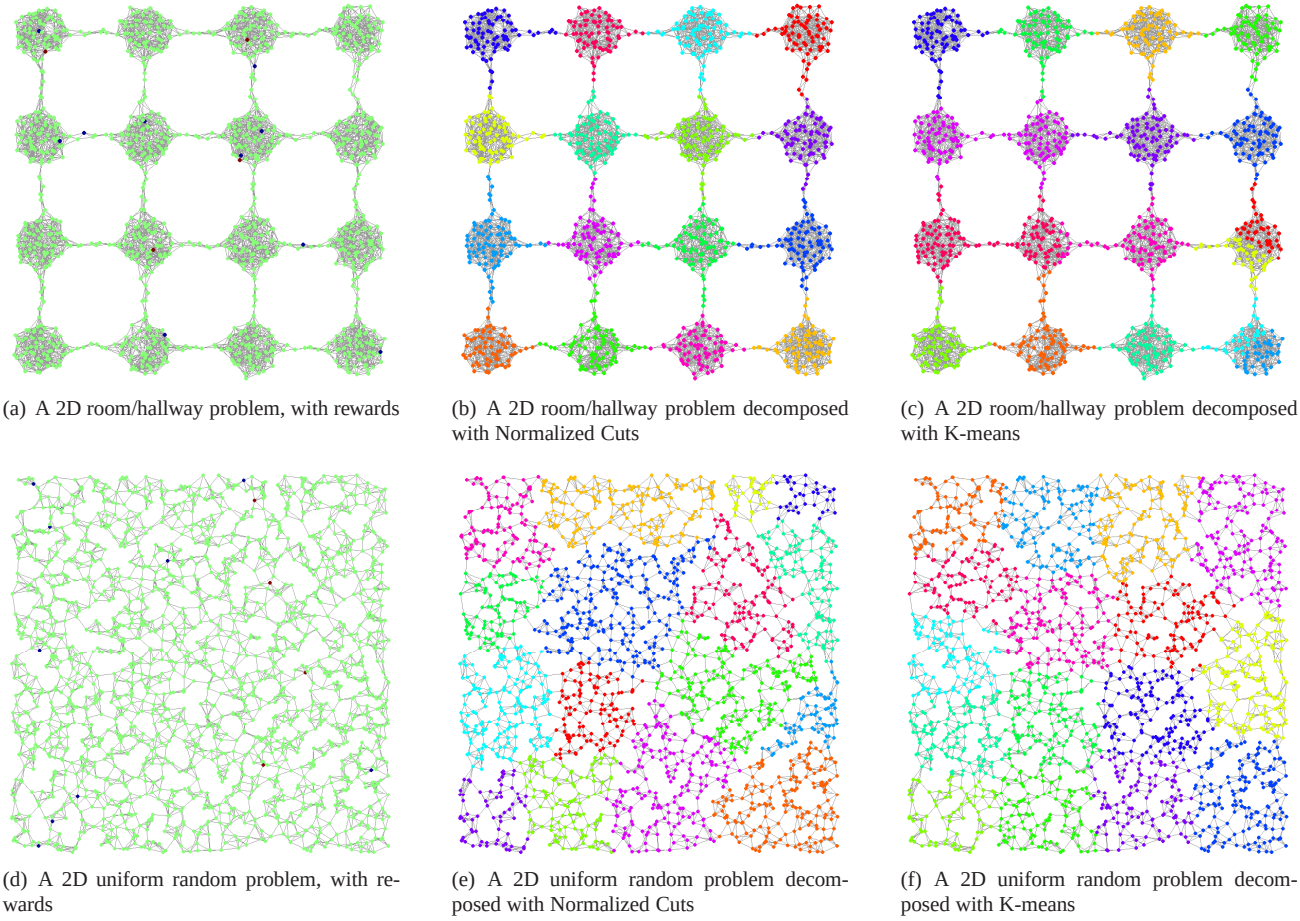
$$V^*(s) = R(s) + \gamma \max_a \sum_{s'} T(s,a,s')V^*(s') \tag{3}$$

The optimal policy π is defined accordingly:

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')V^*(s') \tag{4}$$

In this formulation, the agent receives a reward for the current state, and then transitions to a new state. This formulation has the nice quality that the reward of a "goal" state (an absorbing state that has a non-zero reward) is *equal to its utility*, which makes explaining our parallelized algorithm easier.

To compute V^* (and therefore π^*) we use policy iteration[3], which consists of two operations: policy estimation and policy improvement. In policy estimation, we



(a) A 2D room/hallway problem, with rewards

(b) A 2D room/hallway problem decomposed with Normalized Cuts

(c) A 2D room/hallway problem decomposed with K-means

(d) A 2D uniform random problem, with rewards

(e) A 2D uniform random problem decomposed with Normalized Cuts

(f) A 2D uniform random problem decomposed with K-means

Figure 2. Two example problems. The first row is an example "room/hallway" problem, and the second row is a "uniform random" problem in which state positions are generated randomly. The left column shows the initial problem, where color indicates reward (green = 0, red = $+1$, blue = -1). On the right, we see two decompositions of the problem.

find V^π, the expected reward for starting in state s and following the current policy π as follows:

$$V^\pi(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') V^\pi(s') \tag{5}$$

In policy improvement, we update π using greedy one-step lookahead:

$$\pi(s) = \arg\max_a \sum_{s'} T(s, a, s') V^\pi(s') \tag{6}$$

Unlike the Bellman optimality equation, the policy estimation step simply involves solving a linear system of equations. We first construct the transition matrix for the current policy:

$$T^\pi = \begin{bmatrix} T(1, \pi(1), :) \\ T(2, \pi(2), :) \\ \cdots \\ T(n, \pi(n), :) \end{bmatrix} \tag{7}$$

Equation 5 then reduces to the linear equation:

$$V^\pi = R + \gamma T^\pi V^\pi \tag{8}$$

$$(I - \gamma T^\pi) V^\pi = R \tag{9}$$

Because T^π is sparse in our example problems, this equation can be solved efficiently with a sparse linear algebra package.

4. Parallelizing Reinforcement Learning

The naive solution to parallelizing reinforcement learning is simple: solve each subproblem in isolation and combine together the solutions. As shown in Figure 1(a), this produces terrible results because information does not propagate between subproblems. Our algorithm incorporates subproblem communication while still solving each subproblem in parallel. The output of our parallel algorithm (which is virtually identical to the solution produced by the non-parallelized algorithm) can be seen in Figure 1(b).

Figure 3. A small, toy problem for explaining our algorithm, consisting of a state-space decomposed into two non-overlapping subproblems (A and B), shown as colored nodes. The dashed line represents the division between the two subproblems, found by Normalized Cuts. The encircled regions are the expanded subproblems (A' and B'), in which the "fringe" nodes on either side of the division are included in both subproblems. The node on the far left is a "goal" state in the problem, and is therefore an absorbing state (indicated by the double circle) and has a reward of 1 (indicated by the red 1).

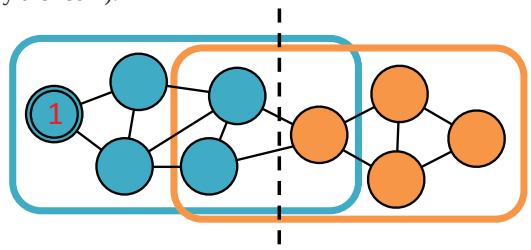
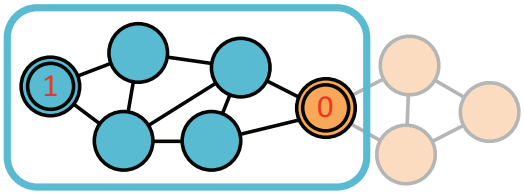
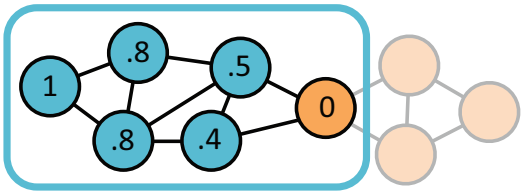


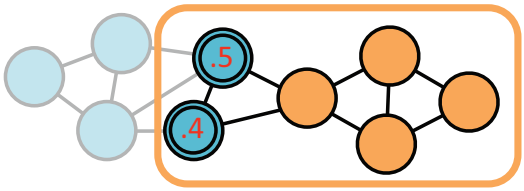
Figure 4. A "serialized" walkthrough of our algorithm on a toy problem containing two subproblems. Absorbing states are indicated by double circles. Rewards (left column) are written in red text, utilities (right column) are written in black text. Note that our definition of utility means that the utility of an absorbing state is equal to its reward.



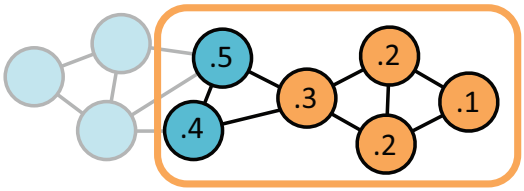
(a) We first partially solve subproblem A, by doing one step of policy iteration on the states in A' (the union of the states in A with the "fringe" states in B). The goal/absorbing state on the left is part of the initial problem, while the absorbing state on the right is artificial, and encodes the current estimate of the utility of transferring into subproblem B. Because we have not yet partially solved subproblem B, that estimate of utility is currently 0.



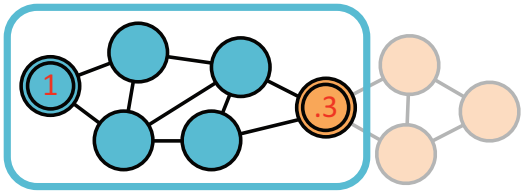
(b) After partially solving ("iterating on") subproblem A, we have estimates of the utilities (rendered in black text) of each state in A' (though the utility of the shared node that belongs to subproblem B is—by construction—unchanged). These estimates will be used when iterating on subproblem B.



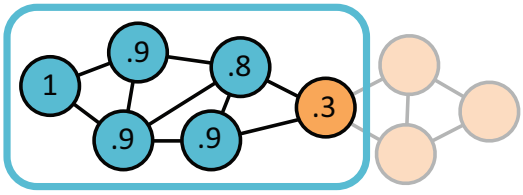
(c) We then iterate on subproblem B, just as we iterated on subproblem A, by partially solving the states in B'. The utility estimates of the fringe states in subproblem A that are also in subproblem B' are, again, used as the rewards of those states, and are treated as absorbing states. Note that, had we iterated on subproblem B before subproblem A, the rewards of those states would have been 0.



(d) After iterating on subproblem B, we have estimates of the utilities (rendered in black text) of each state in B'. Had we iterated on B before A, all of these utilities would have been estimated to be 0, thus demonstrating how information has propagated from subproblem A to subproblem B.



(e) Now, when we iterate on subproblem A again, the reward we assign to the node on the right is different than it was in step (a). Information is therefore passing from subproblem A, to B, and then *back* to A. This is necessary for accurate convergence, even though B has no inherent reward states, and therefore appears to contain no real information.



(f) After iterating on subproblem A again, we have revised estimates of utilities. Note that these estimates are different than they were in step (b), and that the utilities in the fringe of A that we be used in partially solving subproblem B have therefore changed. In this fashion, our algorithm gradually converges to the correct answer.

The crux of our algorithm lies in efficiently managing the communication between subproblems: communication only occurs between neighboring subproblems. The subproblems estimate the utility of transitioning into neighboring subproblems. As the algorithm executes, the estimates of neighbor utility become more accurate, and the internal estimates converge to their correct values. We explain the operation of the algorithm on two neighboring subproblems.

Take subproblems A and B, which are two non-overlapping adjacent subgraphs (as given by Normalized Cuts). Each subgraph, X, has a set of "fringe" nodes, $fr(X)$, whose neighbors include nodes from the other subgraph. We then define two expanded subgraphs $A' = A \cup fr(B)$ and $B' = B \cup fr(A)$. In Figure 3, we see A and B represented as sets of colored nodes, while A' and B' are shown as sets of nodes within colored rectangles.

We describe the algorithm from the perspective of subproblem A. When iterating on A', perhaps by doing one step of policy iteration or value iteration, we make the following changes to the nodes in A' that are in $fr(B)$: they are treated as absorbing states, and their rewards are set to their utilities that were found when last operating on B' (if B' has not yet been operated on, these nodes' rewards are set to 0). All nodes $A \in A'$ are completely unchanged. By changing $fr(B)$ in this way, information from subproblem B is propagated onto subproblem A. This modification solves a slightly different Markov Decision Process, in that A treats the nodes, s' in subproblem B as absorbing states. Hence, upon transferring into state s', A receives some reward, r, and then immediately terminates, while in reality, s' could potentially transition to other nodes. However, the reward r is our current estimate of $V(s')$, which is, by construction, *the expected reward of being in state s' in subproblem B*. This is the same assumption that is made in value iteration, and thus our algorithm achieves the correct solution. A demonstration of this algorithm can be seen in Figures 3 and 4.

5. Optimizations

In planning for our fully parallelized implementation, we experimented with a number of optimizations that we simulated in our MATLAB implementation. These optimizations are all aimed at minimizing the total number of iterations. Their contributions can be seen in Table 2.

5.1. Convergence

Though we do not offer a formal proof, our parallel reinforcement learning framework seems to *always converge* to the optimal estimate of utility V^*. However, it appears that finding the true optimal utility with our framework requires infinite time (and infinite numerical precision), so we need some heuristic for estimating how close we are to V^*.

A well known property of value iteration is that the distance from the estimate of utility at time t to the optimal utility ($||V^* - V^t||_\infty$) is a function of the change in utility from $t-1$ to t, and γ. Since our parallel framework seems similar to asynchronous value iteration between subproblems, this suggests that placing a threshold ϵ on $\left|\left|V^t - V^{t-1}\right|\right|_\infty$ is a reasonable method for determining convergence. In Figure 5, we see how these two quantities correlate as subproblems are iterated.

Our baseline convergence algorithm is to simply record the most recent change in utility $\left|\left|V^t(X) - V^{t-1}(X)\right|\right|_\infty$ for each subproblem X, until all such differences are within ϵ. We also developed an "asynchronous" convergence algorithm, as follows:

- If $\left|\left|V^t(A) - V^{t-1}(A)\right|\right|_\infty < \epsilon$, sleep subproblem A.
- If the reward of any state in $A' \backslash A$ (the expanded set of states around A) has changed by at least ϵ since the last iteration of subproblem A, awaken subproblem A.

This asynchronous convergence algorithm is well suited to a parallel implementation, as convergence is detected in a distributed fashion by each subproblem, rather than by some centralized process that must monitor each subproblem.

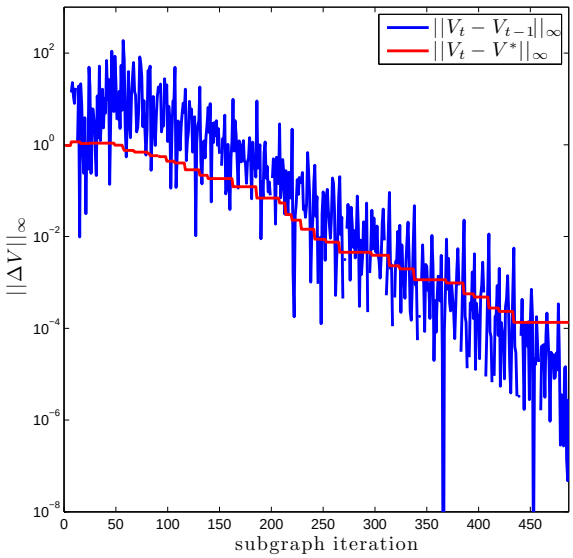


Figure 5. A plot of changes in V, and the error of V with respect to V^*, as our parallelized policy iteration algorithm ($\gamma = 0.99$) progresses until convergence (at which point each subproblem's most recent change was less than 10^{-6}). We can see that the two are indeed correlated.

5.2. Scheduling

We now address the question of selecting which (non-sleeping) subproblem to iterate next. A simple choice is

to pick a subproblem randomly. Another choice is to directly encourage fairness by iterating through non-sleeping subproblems in a round-robin fashion. The results of both strategies can be seen in Table 2.

In practice, random scheduling tends to cause "starvation", which dramatically hurts performance, and often wastes computation as the same subproblem is iterated repeatedly. Repeated iteration is often suboptimal because, once a neighboring subproblem is iterated, the circumstances of the first subproblem may be dramatically different, and so much of the past computation may effectively be wasted. The round-robin scheduler seems like it would address this problem.

5.3. Conflict Prevention

We have observed that, if two adjacent subproblems are solved simultaneously, they will often produce worse results than if they ran serially. In contrast, if two non-adjacent subproblems are solved simultaneously, they produce equivalent results as they would if ran serially. Hence, all other things being equal, we prefer to run non-adjacent subproblems simultaneously. Of course, strictly preventing the running of adjacent subproblems may mean we do not take advantage of parallelism, i.e. if the number of sleeping subproblems surpasses the number of available threads, so we instead simply *discourage* the concurrent running of adjacent subproblems.

The algorithm for "conflict prevention" is as follows: when selecting a subproblem in a round-robin fashion, we first scan the list of subproblems for one whose neighbors are not currently being run. If that scan fails, we re-scan the list, and take the first non-sleeping subproblem.

We more thoroughly analyze these scheduling and conflict prevention ideas in Section 6.4.

6. Parallel Implementation

Our parallelized Java implementation decouples state from iteration scheduling.

The state of the parallelized problem consists of partial solutions for each subproblem along with the messages currently being passed between subproblems. The partial solution records the current approximation to the optimal value function and policy for the subproblem. The messages pass the current utility of fringe states between neighboring subproblems.

Our system performs iterations on subproblems until reaching convergence. Each iteration reads in messages sent from neighboring subproblems updating the reward function, runs one iteration of the reinforcement learning algorithm, then sends messages to neighboring subproblems containing the current utility of the fringe nodes.

The scheduler is the heart of our implementation. It starts a number of worker threads which perform iteration and message passing in parallel.

Each of the following subsections details a component of the system.

6.1. Subproblems and Message Passing

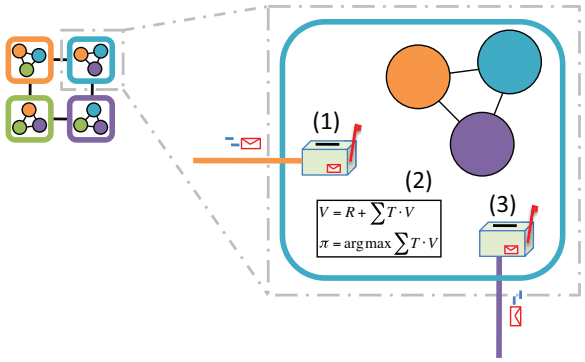


Figure 6. Subproblems and message passing. The numbers describe steps in a worker thread's execution cycle; see section 6.3.

The decomposition of the problem into a graph of connected subproblems is represented by a number of objects (see figure 6).

Each subproblem has an object (the little boxes in the diagram) storing its partial solution: the current approximation to the optimal value function and policy over the subproblem's states along with the most recent utility values of the fringe states of its neighbors as received by messages. For each pair of neighboring subproblems we have a pair of `Communicator` objects (drawn as mailboxes) which handle message passing along the link. Each subproblem sends messages through its `Communicator`, and receives messages by polling its `Communicator`. Since our experiments were all run on the same physical machine, a shared memory implementation of the `Communicator` was sufficient. However, the modular design easily allows for extending to a networking implementation to support communication between subproblems on different physical machines without modifying the rest of the code.

To ensure consistency of parallelized problem state, we only allow a subproblem's state to be modified by a single worker thread. For each pair of neighboring subproblems the two `Communicators` store the last delivered messages. Messages are atomically added and removed from the mailboxes, using a Java `AtomicReference`. The mailbox only needs to store a single message as later messages are more accurate and therefore can override earlier ones.

6.2. Scheduling Architecture

The scheduler determines the order subproblems are iterated by worker threads. One approach would be to assign

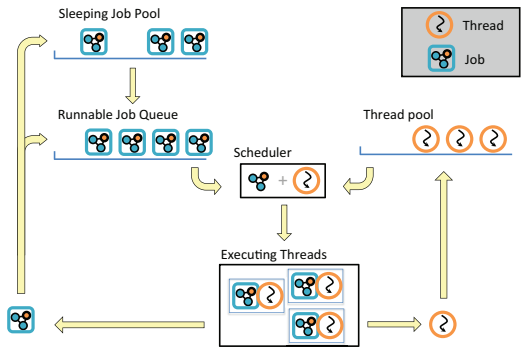


Figure 7. Scheduling architecture.

one thread per subproblem to repeatedly perform iterations. Additional optimizations could sleep threads of subproblems with low priority, as described in Section 5 above.

Instead (see Figure 7) we have a pool of generic worker threads which the scheduler assigns subproblems (jobs). Jobs which have locally converged (as defined in Section 5.1), are put into a pool of sleeping jobs. When all jobs are sleeping the problem is solved and the scheduler can shut the system down. This architecture allows flexible implementation of different scheduling policies for jobs (see Section 6.4 below).

This subsumes the one-thread-per-subproblem scheduler yet allows much greater flexibility. Since most of the program time is spent in the iteration rather than the scheduling code (90% of the runtime is spent within the linear algebra package alone), this flexibility does not come at a great cost.

To ensure consistency of scheduling state, we use Java monitors to deny concurrent access to the scheduler. This does not compromise throughput because the scheduling code is lightweight, iterating a subproblem is relatively expensive, and subproblems tend to finish iterating asynchronously. When there are fewer jobs than threads, threads wait on the scheduler to free more jobs.

6.3. Worker Thread Lifecycle

For a different perspective on the system structure, consider the lifecycle of a worker thread. The thread is created when the scheduler is initialized. The thread requests a subproblem from the scheduler, which selects a subproblem based on the scheduling policy. Once the thread receives the subproblem, it processes the messages sent to the subproblem, performs an iteration on the subproblem, then sends messages into the mailboxes of neighboring subproblems (steps (1), (2), and (3) in figure 6 respectively). After processing, the thread notifies the scheduler of the recipients of the messages it has sent (so that it can wake up sleeping jobs

Figure 8. Setup for possible starvation

who have received messages), returns the job to the scheduler and attempts to request another job. If the returned subproblem has not yet locally converged, the scheduler places it in back in the job queue; otherwise, it is kept separate until the scheduler is notified that a new message has been sent to that subproblem. A thread dies when all subproblems are waiting, and the last thread to die shuts down the scheduler.

6.4. Systematic Analysis of Scheduling Policies

Inspired by the results in Sections 5.2 and 5.3, we systematically analyzed different methods for prioritizing subproblems in the scheduler. Starting with four basic priority orders:

- `T` Select a subproblem which has been iterated the least number of *T* times.
- `N` Prefer subproblems whose *N*eighboring subproblems are not currently running.
- `L` Select the subproblem which *L*east recently completed an iteration.
- `R` *R*andomly select a subproblem.

we form all possible lexicographic orders to break ties. For example, the order `L` always selects the least recently iterated subproblem, i.e. round-robin scheduling; the order `NR` selects randomly from those problems whose neighboring subproblems are not running if possible, otherwise selects a random runnable subproblem; `TL` selects a subproblem which has iterated the least number of times, breaking ties by selecting the problem which least recently finished.

Every ordering ends in either `L` or `R`, as these are the only two orders which are guaranteed never to have ties (the former since subproblems complete iterations serially, and the latter by construction). Hence, there are ten possible orders: `R`, the ordering which just selects a random subproblem to iterate, `NR`, `TR`, `NTR`, `TNR`, `L`, `NL`, `TL`, `NTL`, `TNL`. In section 7 below we empirically compare the performance of these ordering schemes.

Starvation could conceivably be a problem for `NR` and all orders extending it (e.g. `NTR`): a subproblem could always have one of its neighbors running and never run itself. However if the algorithm convergences under a necessarily fair ordering (e.g. `R`), which is empirically the case, this cannot happen. This is because the graph with the starved subproblem's subgraphs removed is itself a problem which will eventually converge, freeing up threads to run the previously starved subproblems.

For a simple example, suppose we have three subproblems A, B, C connected in a row (Figure 8), with two threads of execution under the N scheduling order. If subproblem A is being run by thread 1, then thread 2 will be assigned subproblem C to run as B's neighbor A is running whereas none of C's neighbors are running. Subproblem A finishes before C, and subproblem A still has iterations to run, then thread 1 will again be assigned subproblem A. It seems subproblem B might never be iterated, the two threads always begin assigned to A and C, however eventually subproblem A (or C) will converge, and so B will be given a chance to run.

7. Experiments

The parameters for all experiments, unless otherwise mentioned, are: $\gamma = 0.99$, $\epsilon = 10^{-6}$, $n = 3000$, $k = 16$, $d = 2$ (where n is the number of states, k is the number of clusters, and d is the dimensionality of the underlying space). Since our matrices are extremely sparse, all linear algebra is done with sparse matrix libraries.

We prototyped our algorithm in MATLAB before implementing it in Java. Because multithreading is difficult to implement in MATLAB, we simulated parallelism: that is, when iterating on subproblems "in parallel", we actually run them in serial, and then compute a "simulated" runtime by dividing the serial runtime by the number threads we are simulating. We simulate an 8-core CPU, meaning that we iterate on at most 8 simultaneous subproblems. We used the full Java implementation to verify the accuracy of our simulation.

Experiments 1 and 2 show the effect of decomposition parameters on algorithm performance. We ran these in our prototype MATLAB implementation.

Experiment 3 analyzes the performance of different scheduling algorithms in the Java implementation. We use the results of our simulations studies in experiments 1 and 2 to inform our choice of (n, k, d) parameters for this experiment.

7.1. Experiment 1: Varying n and k

In Table 3, we see the effect on the number of states and the number of clusters on performance. As the number of states increases, the speedup from parallelization also increases. The optimal number of subproblems is between 32 and 64. Using too many clusters when n is small hurts performance. This can be attributed too the increased communication overhead between clusters, or possibly because we use too few cores to take advantage of such a large number of clusters.

In policy iteration, one visible trend is that, when n is large, small numbers of subproblems significantly hurt performance. This is because, whenever there is any amount of parallelization, the number of iterations that is required in policy iteration to reach convergence increases rapidly (see Tables 1 and 2 for examples). This is a fundamental property of the message-passing system of our parallel framework, and of policy iteration. We simply cannot expect the performance of policy iteration to improve unless we have enough subproblems to offset this phenomenon.

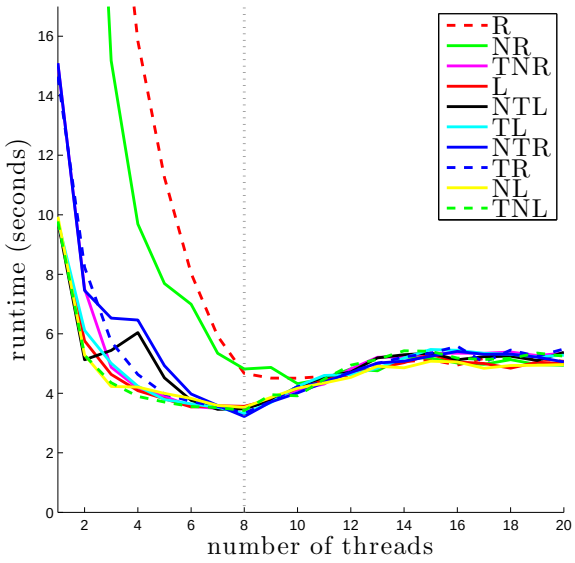


Figure 9. Runtime the different scheduling policies in the Java implementation as a function of thread number on an 8-core machine. The legend orders scheduling policies by the median runtime over 1-8 threads.

7.2. Experiment 2: Varying d

In Figure 10, we see performance relative to the dimensionality of the underlying space from which the problem was generated. As predicted, our algorithm helps performance when d is small (less than 3), but hurts performance when the dimensionality is large (greater than 4). The reason for this is simple: as d increases, the number of subproblems that a given subproblem is adjacent to will also increase, and a greater fraction of states will be shared between subproblems. The utilities of these shared nodes take a very long time to converge, as they depend on message passing between subproblems. Therefore, the more shared nodes we have, the worse performance should be. An illustration of this sharing can be seen in Figure 11, which contains renderings of the graph adjacency structures for random graphs with different underlying dimensionalities.

7.3. Experiment 3: Different Scheduling Policies

We tested the performance of different scheduling policies on 20 uniform random subproblems (the same set of problems tested on different policies) each decomposed into

method	actual-time	simulated-time	iterations
policy iteration	1580 ± 160 ms	1542 ± 155 ms	10 ± 1
policy iteration - parallelized	6186 ± 708 ms	711 ± 86 ms	517 ± 61

Table 1. Performance of policy iteration (actual/serial runtime, simulated/parallel runtime, and the number if iterations on the sub-problems) parallelized and non-parallelized, on 2D uniformly random problems. We present the mean and standard deviation of performance for 20 different random graphs.

	2D Room/Hallway			2D Uniformly Random		
method	simulated-time	iterations	error $\times 10^4$	simulated-time	iterations	error $\times 10^4$
non-parallelized	2315 ± 655 ms	7 ± 1	0.00 ± 0.00	2016 ± 510 ms	10 ± 1	0.00 ± 0.00
parallelized baseline	2348 ± 856 ms	796 ± 527	0.18 ± 0.76	1173 ± 139 ms	780 ± 68	0.04 ± 0.11
+ asynchronous convergence	1830 ± 545 ms	571 ± 306	0.64 ± 0.91	979 ± 113 ms	615 ± 42	0.70 ± 0.95
+ round-robin scheduling	1456 ± 419 ms	449 ± 215	0.21 ± 0.39	785 ± 94 ms	490 ± 43	1.31 ± 4.36
+ conflict prevention	1347 ± 509 ms	397 ± 301	0.24 ± 0.47	845 ± 102 ms	517 ± 61	0.39 ± 0.53

Table 2. Performance (actual/serial runtime, simulated/parallel runtime, and error relative to the true optimal utilities) of policy iteration with respect to the different optimizations in Section 5, for the two different kinds of problems. We present the mean and standard deviation of performance for 20 different random graphs.

16 subproblems, varying the numbers of threads. We ran all experiments on an 8 core system using between 1 and 20 threads. Figure 9 shows the runtime averaged over 20 different graphs as a function of thread count for each of the scheduling orders.

As can be seen in the figure, for each scheduling policy the execution time decreases as we increase the thread count until reaching one thread per core (8 threads), then increases to plateau at 16 threads and beyond. This shows our system makes effective use of additional cores, and that our scheduling policies outperform native Java thread scheduling. The policies plateau to a common execution time at 16 threads because there are 16 subproblems and at most one thread can be used per subproblem.

It is clear from the figure that the schedules fall into one of two groups: the slow group consisting of R and NR, and a fast group consisting of everyone else. R is the most naive scheduler, it is unsurprising that it performs slowly. Since N contains only two classes, it does not significantly partition the threads and therefore performs similarly to R.

The remaining orderings take into account the number of times run (T) or the time of completion (L), therefore introduce more stratification between the subproblems and, moreover, guarantee strong interleaving between different jobs. Because there are many cycles present in the graph, and the values of a subproblem depends on the values of the neighbors, the subproblems must make equal progression towards the goal. The strong interleaving ensures that the jobs progress together towards the goal. By making sure no subproblems lag behind the rest, the system reduces the number of situations in which a subproblem makes a large change that must be propagated through the graph.

Note that as the ratio of number of threads to subproblems increases, the scheduling policies become more alike, as there are fewer choices of subproblems to schedule. In the limit, as we observed above, they converge to the same algorithm: effectively one thread per subproblem. Since the optimal number of threads, one for each of 8 cores, is a significant fraction of the number of subproblems (16), our results understate the difference between scheduling policies. Nonetheless, there is a clear separation between the slow and fast group of policies.

8. Conclusions

We have introduced a technique for decomposing Markov Decision Processes into subproblems, a framework for doing parallel reinforcement learning on these decomposed problems, a number of optimizations for this framework, and many experimental results that show where this parallel framework works well. Our technique is capable of reducing computation for many problems by a factor of $10\times$, and appears to be very robust to the type of problem being solved (as long as it was generated from a low-dimensional space).

We developed a parallel Java implementation of this technique, using a hybrid message-passing job-pool architecture, and systematically analyzed different scheduling policies. The performance of this system validated our MATLAB simulation, whilst highlighting scaling problems with the particular Java linear algebra library used. Given the system design it would be straightforward to implement a distributed networked implementation; this would be a natural area for future work. Our results demonstrate that, even without a distributed implementation, our algorithm achieves competitive performance.

References

[1] R. Bellman. *Dynamic Programming*. Dover Publications, March 2003.

Number of States \ Number of Subproblems	1	2	4	8	16	32	64	128
250	78	180	91	104	121	216	295	-
500	153	340	203	128	133	171	425	-
1000	375	772	460	284	261	213	347	854
2000	974	1,385	1,018	556	480	352	506	854
4000	1,957	4,074	1,949	1,304	943	780	915	1,228
8000	4,698	7,509	3,502	1,900	1,622	1,309	1,339	1,635
16000	9,401	18,694	10,991	4,561	3,806	2,971	2,971	3,313

Table 3. An illustration of the effect of the number of states and the number of subgraphs on simulated runtime (in milliseconds). All experiments were run on 2D, uniformly random problems. Empty squares indicate where Normalized Cuts failed, due to clusters of size 1. Color is proportional to $\log(time)$. Note that the leftmost column is equivalent to non-parallelized policy iteration.

Number of States \ Number of Subproblems	1	2	4	8	16	32	64	128
250	259	443	322	141	185	363	711	-
500	693	881	563	282	268	353	663	569
1000	2,232	6,229	2,408	2,082	847	677	576	-
2000	16,533	26,807	7,389	4,633	1,672	1,353	680	688
4000	95,495	103,268	40,733	13,157	5,073	3,177	1,655	1,395
8000	293,980	330,742	126,899	36,026	7,930	8,631	3,706	2,638
16000	1,514,571	2,656,038	631,055	166,226	57,049	27,098	7,431	8,579

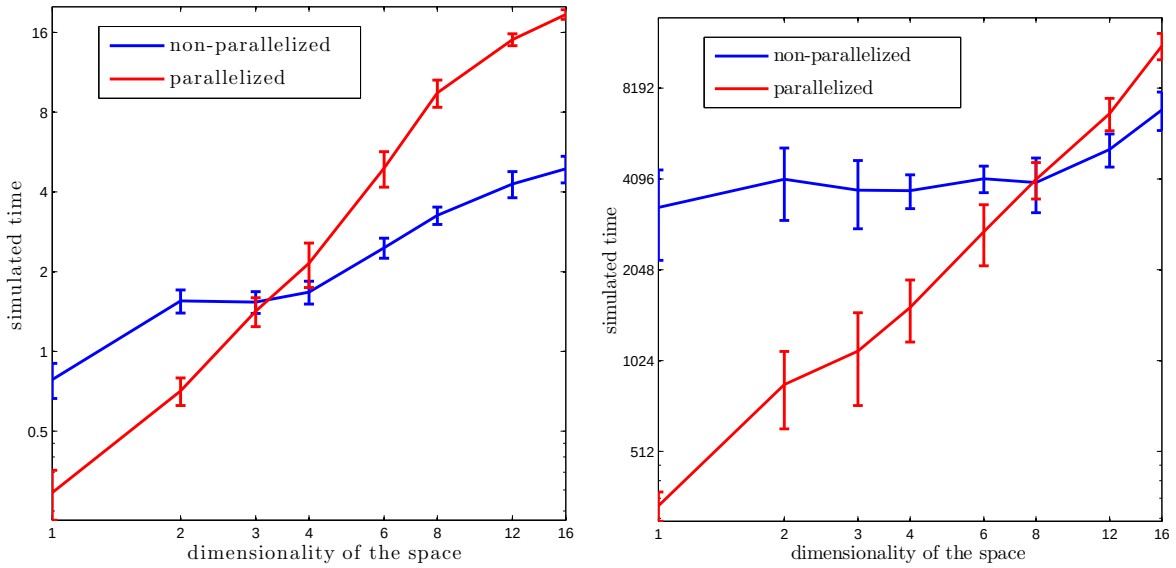
Table 4. The analog of Table 3 for policy iteration in the Java implementation. Note the poor scaling as states increase without parallelization, but comparable performance with parallelization. This is due to poor scaling of the particular Java linear algebra library compared with the native and highly optimized Matlab implementation of the same.

[2] D. P. de Farias and B. Van Roy. The linear programming approach to approximate dynamic programming. *Oper. Res.*, 51(6):850–865, 2003.

[3] R. A. Howard. *Dynamic Programming and Markov Process (Technology Press Research Monographs)*. The MIT Press, first edition edition, June 1960. 2

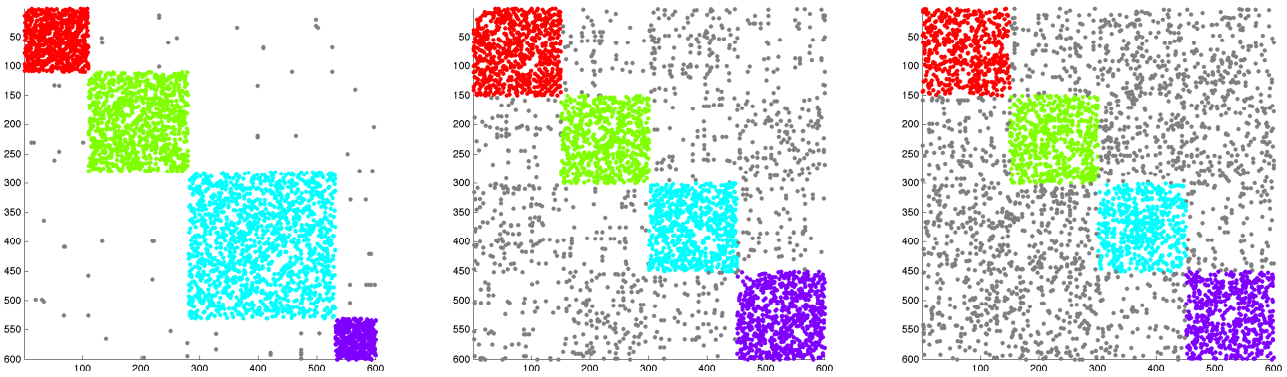
[4] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. L. Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967. 1

[5] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000. 1



(a) MATLAB / simulated time (b) Java / actual time

Figure 10. A plot of performance (averaged over 20 random graphs) as the dimensionality of the underlying (uniform random) problem increases



(a) 2D world (b) 6D world (c) 16D world

Figure 11. Renderings of the state adjacency matrix for uniformly random problems of different dimensionalities (each with the same number of states), after Normalized Cuts has been run. Each dot represents a connection between states i and j. Colored dots indicate that an edge is within a given subgraph, while gray dots are edges between subgraphs. As the dimensionality of the underlying state increases, the number of within-subgraph edges decreases, while the number of between-subgraph edges increases.