P3VI: A Partitioned, Prioritized, Parallel Value Iterator

David Wingate WINGATED@CS.BYU.EDU
Kevin D. Seppi KSEPPI@CS.BYU.EDU
Computer Science Department, Brigham Young University, Provo, UT 84602 USA

Keywords: Asynchronous dynamic programming, value iteration, reinforcement learning.

Abstract

We present an examination of the state-of-the-art for using value iteration to solve large-scale discrete Markov Decision Processes. We introduce an architecture which combines three independent performance enhancements (the intelligent prioritization of computation, state partitioning, and massively parallel processing) into a single algorithm. We show that each idea improves performance in a different way, meaning that algorithm designers do not have to trade one improvement for another. We give special attention to parallelization issues, discussing how to efficiently partition states, distribute partitions to processors, minimize message passing and ensure high scalability. We present experimental results which demonstrate that this approach solves large problems in reasonable time.

1. Introduction

This paper combines several previous research ideas related to high-performance value iteration (VI) into a single algorithm capable of efficiently solving very large reinforcement learning problems. The algorithm has been named "P3VI," which stands for *Partitioned, Prioritized, Parallel Value Iterator*.

Many problems in RL are well modeled as Markov Decision Processes. Computing the value function of MDPs is one way to generate an optimal policy, and is generally a non-trivial task. Obviously, the ability to compute the value function quickly enables larger, more complicated, and more relevant problems to be solved.

Appearing in *Proceedings of the 21st International Conference on Machine Learning*, Banff, Canada, 2004. Copyright 2004 by the authors.

In addition, many techniques rely on accurate value function estimates to make *other* decisions: Munos and Moore [7] use the value function to guide discretization decisions, and Kearns and Singh [5] use it to decide between exploration and exploitation. Value iteration is also used as part of larger algorithms: RTDP [2] performs some value iteration off-line between executing controls, and Modified Policy Iteration [9] performs some value iteration between policy evaluation steps. P3VI can be an effective companion to these other algorithms, and can improve their performance by reducing off-line computation overhead. In addition, P3VI can enhance algorithms that propagate different forms of information (not just *value* information). For example, Munos and Moore [7] propagate both *influence* and *variance* throughout a problem using VI. In a more abstract sense, the principle of propagating knowledge throughout a space as quickly and efficiently as possible is applicable to continuous systems as well as more conceptual systems.

Value iteration is a robust and simple algorithm for solving MDPs, but is not usually considered a viable on-line algorithm because of its slow convergence. Recent research has suggested that this is due to *inefficient computation*: many backups can be redundant, useless or suboptimal [11]. There have been several successful attempts to alter the basic VI algorithm to enhance performance. Previously, researchers have considered *prioritized* value iteration [6], *partitioned* value iteration [11], and *asynchronous* VI [3]. Asynchronous VI sometimes implies partial sweeps through the state space [10], but it sometimes implies *parallel* VI. However, this has traditionally been more of a theoretical construct, as opposed to actual implementations which run on supercomputers or clusters.

This work combines the three ideas of partitioning, prioritization, and parallelization into a single algorithm. A central point of the paper is that algorithm designers do not need to trade one enhancement for another:

all three are compatible, and even complimentary; any combination improves performance while maintaining convergence and optimality guarantees.

There are practical and theoretical problems with the design of such a composite algorithm. Since both prioritization and partitioning have been thoroughly studied as independent enhancements [11], the focus of this work is on the parallelization, and the unique issues resulting from a combination of all three ideas. This research answers questions related to scalability and efficiency by contributing insights into the design and implementation issues associated with parallelization. The most significant insights relate to an effective domain decomposition: we note that naive block-decomposition methods are unlikely to be effective, because of the way in which prioritization focuses computation on an "information frontier." We therefore analyze a heuristic decomposition designed to balance parallelization and prioritization. Other questions addressed include the following: can the idea of prioritized VI be efficiently implemented in parallel? Can the gains of the parallelization be quantified? How does a parallel, partitioned, and prioritized value iterator compare to a parallel naive value iterator? The paper presents experimental results designed to quantify these answers.

The goal of this research is to create solution engines which are fast, efficient, and scalable. The existence of such engines could have profound benefits: it could enable new applications that depend on real-time reinforcement learning of large problems; it could enable new engineering methodologies (by automatically computing and validating control policies); and it could move reinforcement learning to a higher level of applicability by commoditizing the solution of more complicated problems than are currently feasible.

2. The P3VI Algorithm

As noted, the P3VI algorithm is *prioritized*, *partitioned*, and *parallelized*. Prioritization is the primary performance enhancement, because it ensures that computation is focused on regions of the problem space which are expected to be maximally productive. Partitioning helps to reduce overhead by providing an approximate prioritization, and leads to data and code organization which is suitable for parallelization. Parallelization allows the entire system to scale far beyond the capabilities of comparable serial algorithms. All three enhancements to basic value iteration are complementary, and will be discussed in this section; the overall algorithm is shown in Figure 1.

```
Initialization

  1. Partition the state space
  2. Assign partitions to processors
  3. Coordinate dependencies between processors
  4. Construct a priority queue for partitions local
     to each processor

Repeat (in parallel)

  1. Select the highest priority partition $p$ from
     the local queue
  2. Value iterate over the states within $p$, until
     the maximum change $< \epsilon$
  3. Recompute the priorities of local partitions
     depending on $p$
  4. Inform foreign processors about new values of
     states in $p$
  5. Process incoming messages: for each foreign
     partition $f$ that has changed, recompute the
     priorities of local partitions that depend on
     states in $f$

Until stopping criteria are met
```

Figure 1. Pseudocode for the the P3VI algorithm.

The seminal work on the subject of prioritizing updates in value iteration and Q-learning was done by Moore and Atkeson [6] in their work on Prioritized Sweeping (PS). In PS, a priority metric is defined, and a priority queue is constructed and maintained which allows the algorithm to process backups in priority order. Typically, Bellman error is used as the priority metric, although unpublished research by Wingate and Seppi has demonstrated that another equally simple metric (called the *H2* metric) generally yields better performance. The PS algorithm proceeds by extracting a state s from the queue, updating it, and then recomputing the priority of s and any states depending on s. Naturally, PS must store a full model inverse in order to correctly update dependent states.

Unfortunately, demanding perfect prioritization creates prohibitive overhead: for every backup, every dependent state must be extracted from the queue, reprioritized, and reinserted into the queue. This overhead can be substantially reduced through the use of partitioning, which creates the possibility of *approximate* prioritization. In partitioned VI, states are aggregated into sets such that states within each set are mutually dependent (or such that they depend on as few things outside the set as possible). Partitioned, prioritized VI can then be performed on the aggregated problem, with the same core ideas. Each partition can be prioritized with a metric that represents the maximum priority of any state within the partition. The algorithm begins by selecting the highest priority partition p. States within p are repeatedly backed up until they converge. The priorities of p and any partition which depends on any state in p are then recomputed. The priority queue is updated to reflect the new priorities, and the process repeats until stopping criteria are met.

There are several benefits to partitioning. The first is a dramatic reduction in priority queue overhead. The second is that the use of partitioning can reduce the storage requirements of the algorithm, because only a partial model inverse needs to be stored: instead of storing the dependents of every state, it is only necessary to store dependents that cross partition boundaries. In addition, partitioning can block off certain unreachable regions of the problem space, so that they never need to be processed at all [11].

The final benefit is that a partitioned VI algorithm naturally enables an efficient parallel VI algorithm. Partitions can be assigned to different processors, which allows them to operate on different parts of the problem in parallel. Since partitions are designed to minimize the amount of storage and computation required to prioritize backups, the same partitioning can be used to reduce the amount of inter-processor communication in a parallel algorithm.

In the P3VI algorithm, a set of partitions is created (the number of which is greater than the number of processors, as discussed in the next section) and divided among the processors. The processors then execute the algorithm shown in Figure 1 asynchronously. Each processor maintains a local priority queue which prioritizes locally assigned partitions. The processor selects the local partition p with the highest priority, and value iterates over the states within it until they converge. The priorities of local partitions depending on p are then updated, and the new values of states within p are communicated to processors that need to be informed. Naturally, there may be some partitions which do not contain any states whose values need to be communicated to a foreign processor. When run on one processor, the local priority queue contains all of the partitions in the entire problem.

In a parallel value iterator, the issue of cross-processor dependencies is important. It is possible that states owned by one processor depend on the value of states owned by another processor. In fact, the *priority* of a local partition will often depend on the value of states owned by foreign processors. This necessitates important steps which are unnecessary in serial versions. First, in the initialization phase of the algorithm, all dependencies are coordinated between processors: each processor must inform foreign processors that they wish to be informed when the value of a dependency changes. The value of a state will only change once a processor has selected a local partition and backed up the states within it; thus, after processing a partition, a processor must communicate the values of states in the partition to all foreign processors that depend on those states. This implies that processors must maintain a cache of the last known value of a foreign state, which adds some space complexity. Cross-processor transitions must be minimized for several reasons: first, to simplify the coordination phase; second, to minimize the number and size of messages communicated after processing a partition; and third, to minimize the size of the foreign state cache.

The processor then processes incoming messages, which are of two types. The first is a "partition update" message, which contains the values of states in foreign partitions that have changed. The second is a "termination" message, which will be explained shortly. If no incoming messages are waiting, the processor may proceed to select another partition, and work on it normally. Once a processor is finished (all local partitions have a priority that is below some threshold), it informs a designated master processor and blocks on incoming messages. This is part of the distributed stopping algorithm, which is explained in the next section. To avoid starvation, the maximum number of incoming messages that will be processed is equal to the number of processors; once those messages have been processed, the processor returns to working on the local priority queue.

This architecture has the effect of allowing processors to work independently whenever possible, but forcing synchronization when necessary. As with any asynchronous algorithm, care must be taken to avoid deadlock.

3. Algorithm Design Issues

There are several issues involved in the design and implementation of the P3VI algorithm, which may be divided into two broad categories: first, there are theoretical issues related to optimality, convergence, and appropriate decompositions. Second, there are low-level issues related to stopping, deadlock avoidance, and efficient internal representation. Since low-level issues are largely implementation dependent, and discussion of them would detract from the focus of the paper, they will not be discussed in detail.

Convergence is established by appealing to Bertsekas

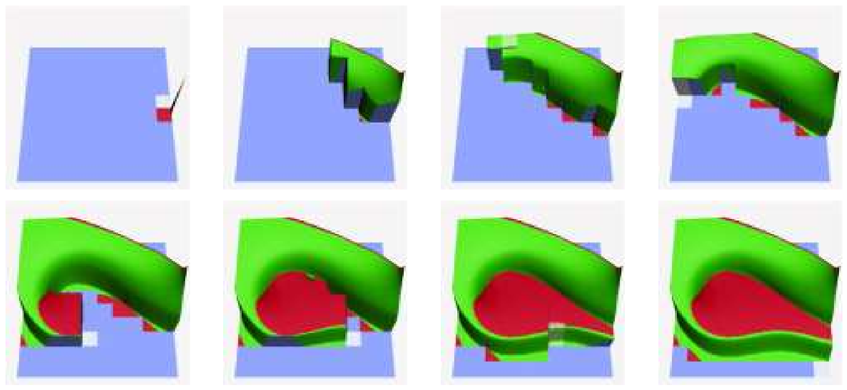

Figure 2. Moving from left to right: the "information frontier" of the Mountain Car (as discussed in Section 4) value function propagates outward from the primary reward. Red (dark) and green (light) colors indicate different controls.

[3], who provides proof that asynchronous VI converges without the assistance of a common clock. The scenario described in his paper exactly matches the scenario of the P3VI algorithm; the addition of partitioning and prioritization does not affect convergence, because the convergence guarantees are provided for arbitrary backup orderings. The P3VI algorithm simply selects one of many backup orderings, which happens to be principled and efficient.

The optimality of the final solution is also uncompromised: it is well-known that the optimal solution of a value-iteration process is a unique fixed point [8], and that if the maximum Bellman error $\|V_t - V_{t+1}\| = \epsilon$ then $\|V_t - V^*\| \leq \epsilon/(1-\gamma)$. (Here, V_t denotes the value function at time t, V^* denotes the optimal value function, $\|\cdot\|$ is the max-norm operator, and $\gamma \in [0,1)$ is the discount factor). P3VI stops when all processors report that the maximum Bellman error is less than ϵ, which satisfies the condition. Since the fixed-point is unique, the policy computed is within $\epsilon/(1-\gamma)$ of optimal. Determining exactly when all processors report a maximum Bellman error less than ϵ is essentially a distributed stopping algorithm. P3VI computes this by maintaining a counter of processes that are done, combined with some additional checking to avoid a mild race condition. Termination messages are sent when all processors report that they are finished.

The most interesting problems confronting the P3VI algorithm involve appropriate domain decompositions. There are two broad issues: first, how are states allocated to partitions? Second, how are partitions allocated to processors? The issue of allocating states to partitions is not treated in this work, for the following reason. The focus of the work is exploring the general benefit of parallelization, and not in tuning the many specific design choices involved. The barrier to entry of creating principled partitions is quite high, and an adequate partitioning can be constructed simply by using the geometric coordinates of each state. This is a consequence of our experimental setup: the MDPs are derived from continuous state optimal control problems, and the states have associated coordinates in the original state space, meaning that partitions of highly related states can be generated by gridding the state space. Solving more general MDPs for which geometric information is not available is an important issue that has been left for future research. We anticipate that existing k-way minimum-cut graph partitioning algorithms, such as recursive spectral bisection [1] or parallel multilevel partitioning [4] will be useful in partitioning such problems.

The issue of allocating partitions to processors is more interesting, and is one of the focuses of this work. The difficulty is finding a balance between parallelism and prioritization, and is best explained through an example. Consider Figure 2. Shown are frames in the evolution of the Mountain Car value function, demonstrating the backpropagation of value information throughout the state space, which travels as an "information wave." The prioritization metrics are designed to focus computation on the crest of the wave, between the region of not-yet-processed and already-converged. The problem lies in the fact that a traditional block decomposition will not yield good parallelization. As an example, consider solving the Mountain Car problem with two processors. Assume that the problem was divided into roughly equal blocks named A and B (as shown in the upper-left image of Figure 3), and that block A (on the left) was assigned to processor one, and that block B was assigned to assigned to processor two. This decomposition would perform quite poorly:

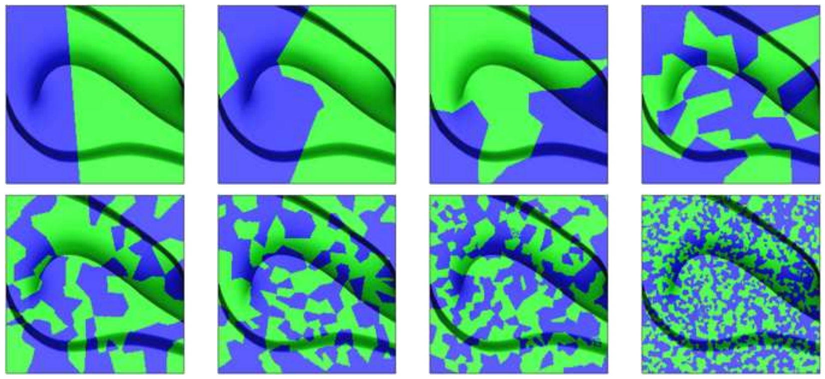

Figure 3. The assignment of partitions as a function of attractors. Shown is a top view of the Mountain Car value function. Blue (dark) and green (light) colors indicate assignment to processor one and two, respectively. Shown are the resulting assignments for (upper-left to lower-right) 1, 5, 10, 20, 50, 100, 200, and 1000 attractors per processor. The more attractors are added, the more random the partition assignment appears to be.

processor one would start off idle, while processor two would drive the information wave from the primary reward until it exited block B and entered block A. Processor two would then be largely idle, while processor one drove the information wave around the curve, and back into block B. Finally, processor one would be idle, as processor two completed pushing the information wave through the state space.

The example illustrates two points. First, it is clear that the ideal scenario is to have all processors work on the information frontier in parallel, but this is difficult to do, since it is not known in advance how the information frontier will progress through the state space. Second, it seems clear that creating more partitions than processors may allow the processors to always be working in parallel. This suggests that a fully random allocation of partitions to processors may be a viable solution, but intuitively, it seems that this would create little cohesion between partitions, and would result in prohibitive inter-processor communication. This is one of the central issues our experiments were designed to test. We leave the idea of *dynamically* allocating partitions to processors to future research.

At one extreme, therefore, is a full block decomposition, where the state space is divided equally among processors in large contiguous regions. At the other extreme is a fully random allocation. To explore this continuum, we introduce the idea of "attractors" in the partitioning, which are essentially nodes in a radial basis function. We allocate n attractors per processor, and scatter them randomly throughout the state space. Partitions are assigned to processors by computing the partition's distance to all of the attractors and assigning it to the owner of the nearest attractor. Adding more attractors makes the assignment more random, as shown in Figure 3.

4. Experimental Setup

The P3VI algorithm was validated by solving several problems of varying complexity. Here, we consider only discrete, model-based, stationary, positive-bounded, non-deterministic MDPs. We selected the Mountain Car (MCAR), Single-Arm Pendulum (SAP), Double-Arm Pendulum (DAP), and Triple-Arm Pendulum (TAP) problems. MCAR is a traditional two-dimensional RL problem; SAP is also two-dimensional, but DAP is four-dimensional and TAP is six-dimensional.

To quantify the relative benefits of partitioning, prioritization and parallelization, we tested three different algorithms. Each algorithm combines some of the proposed enhancements: the *P-EVA* algorithm [11] is partitioned and prioritized, the *PSVI* algorithm is partitioned and parallelized, and the *P3VI* algorithm is partitioned, prioritized and parallelized.

The naive parallel implementation of standard VI (or PSVI) is a non-prioritized version of P3VI. States are aggregated into partitions, and partitions are assigned to processors. Instead of prioritizing partitions, however, each processor sweeps over all of its partitions repeatedly. After processing a partition, the processor communicates new state values to foreign processors in the same way that P3VI does, and moves to the next partition it is responsible for.

Several sets of experiments were run. First, we tested all three algorithms on increasingly larger versions of

DAP. Second, scalability tests were run to determine the efficiency of the algorithms as the number of processors was increased. Finally, experiments were run to evaluate different partition-to-processor mappings.

The problems tested are continuous time, and involve continuous action and state dimensions. These problems were selected because the number of states used in the discretization process could be changed at will, allowing us to smoothly vary the size of the problem (thus generating families of highly related MDPs), while allowing us to easily generate partitions. To discretize the space, we use the same approach described by Munos and Moore [7], except that no *variable* discretization is used. Instead, the space is discretized once in the initialization phase. We refer the reader to their work for a complete description of the technique, but we note that the discretization process is tangential to the research focus of this paper. There are many other methods which could have been used to discretize the problems; naturally, this particular method introduces a bias with respect to the original problem, but since the solution engine simply expects a discrete MDP, the details of where it came from are somewhat irrelevant. We point out that these problems could perhaps be solved using function approximators. The point of this research is not to solve these control problems per se, but to solve very large MDPs quickly; we have selected these problems as inexpensive sources of MDPs .

Partitions are generated by leveraging the geometrical information available for each state. Naive block partitioning of the state-space was performed by gridding uniformly in each dimension. However, we stress that existing k-way graph partitioning techniques can be effectively used to generate partitions on problems for which geometric information is not available [1, 4]. In all experiments except the attractor series, partitions were allocated to processors randomly.

Different partition-to-processor mappings were tested by using partition attractors, as described in Section 3. Recall that attractors do not *define* partitions, but that they *assign* existing partitions to processors. In our attractor experiments, a constant number of partitions are generated a priori (the number of which is far greater than the number of processors), and then assigned to the processor which owns the nearest attractor. Many attractors can be assigned to each processor; the number of attractors per processor is a tunable parameter.

We also report the efficiency of the parallel algorithms, which is computed as $e = T_1/(p * T_p)$. Here, T_1 is the amount of time required to solve the task on one processor, p is the number of processors, and T_p is the amount of time required to solve the task using p processors. Higher efficiencies are better; efficiencies greater than 1.0 represent superlinear speedup. The efficiency on one processor is always 1.0.

In all experiments involving MCAR, a slightly modified version of the reward function was used. The traditional MCAR reward function is continuous and involves positive and negative rewards. In the modified reward function, the agent only receives a reward upon exiting the state space with a position of +1.0 and a velocity of 0.0.

All experiments used the same problem configuration: MCAR, SAP and TAP had 1,000,000 states and 10,000 partitions. DAP revealed a more complex MDP at this discretization; to facilitate the exposition of the results, it was discretized at 456,976 states and 4,096 partitions. In experiments with attractors, all tests used the P3VI algorithm on 8 processors.

All forms of VI used Gauss-Seidel backups [8]. All priority queues used the $H2$ priority metric (which is defined as the value of the state plus the Bellman error if the Bellman error is $> \epsilon$, and 0 otherwise). For all experiments, epsilon was set to 0.0001 and gamma was set to 0.9. All code was implemented in C, using MPI[1]. Experiments were run on a fully connected cluster of dual processor 2.4GHz Pentium 4s, with 2G RAM and Myrinet interconnects.

5. Results

Figure 4 shows that P3VI consistently outperforms VI, PSVI and P-EVA. This demonstrates that the three enhancements of prioritization, parallelization and partitioning can be effectively combined into a single algorithm that outperforms other algorithms that only combine one or two of the enhancements.

The results of attractor-based partition assignments are shown in Figure 5. As predicted, using only a few attractors (resulting in extremely large, contiguous blocks) did not perform well at all, and increasing the number of attractors almost always improved performance. Often, using about 50 attractors per processor halved the time required to solve the problem. The benefits of more attractors quickly diminished after this number, however. In fact, a random assignment works equally well as a large number of attractors. The attractor-based experiments supported our intuition: that a domain decomposition in which all processors can compute on the information frontier gives better

[1]Source is available at http://aml.cs.byu.edu/code/

performance than a naive block decomposition.

P3VI scales moderately well as the number of processors is increased, as shown in Figure 6. The efficiency is consistent, approaching 0.4 as the number of processors increases (this efficiency leaves room for improvement). Figure 7 indicates that P3VI scaled superlinearly on the MCAR problem, which deserves some investigation. The canonical explanation for superlinear scaling is *cache coherency*: for certain problems, and for a fixed problem size, increasing the number of processors increases the cache-to-data ratio. To explore this, we computed the average number of times each partition was processed. For this experiment, the partitions in the MCAR problem were processed an average of 9 times each – low enough to exhibit excellent cache behavior. The other possible explanation for this behavior is that concurrently processing partitions sometimes results in a backup order that is more optimal than the update order imposed by the priority metric, but this theory remains to be validated.

The results also make it clear that a naive parallelization of value iteration (the PSVI algorithm) performs quite poorly. Figures 8 and 9 indicate that PSVI works well for MCAR and SAP, and is as efficient as P3VI (approaching an average efficiency of 0.40). The results on DAP and TAP were very different: adding processors to the PSVI algorithm degraded performance. At 32 processors, it finally performed slightly better than with one processor. Since the number of cross-processor transitions increases with the dimensionality of the underlying problem, this may imply that communication overhead is prohibitive.

6. Conclusions and Future Research

Based on the results of our experiments, we can draw a very strong conclusion about the central theme of the paper: that the benefits to be had from parallelization, partitioning and prioritization are real and compelling, and they can be effectively implemented in a holistic algorithm. Algorithm designers do not need to trade one improvement for another; in fact, there is preliminary evidence to suggest that the combination of the three can be synergistic.

Perhaps the most positive conclusion is that the limiting factor of the algorithm is no longer time, but RAM. Although P3VI adds some space complexity to normal VI, it is not prohibitive. Indeed, any problem that fit into RAM was easily solved, which motivates future research into caching and approximation methods.

The most pressing direction for future research is to allocate states to partitions in a more principled way,

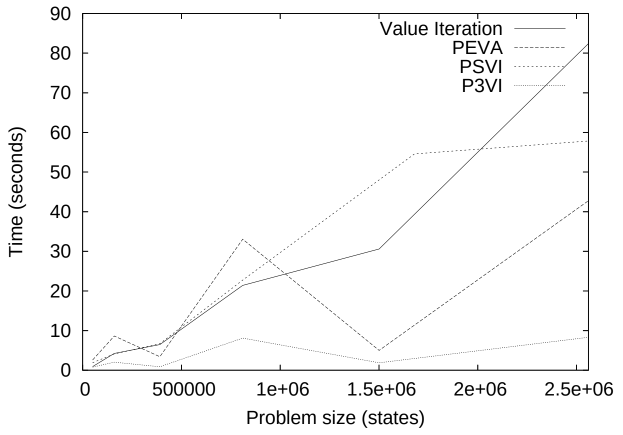

Figure 4. DAP performance of various algorithms as a function of problem size.

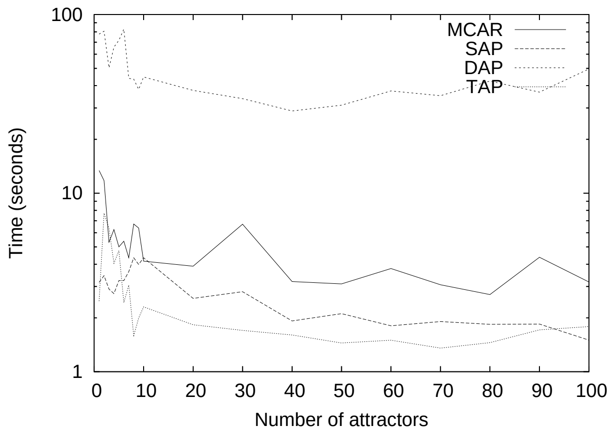

Figure 5. Performance of P3VI as a function of the number of attractors used. Note the log scale.

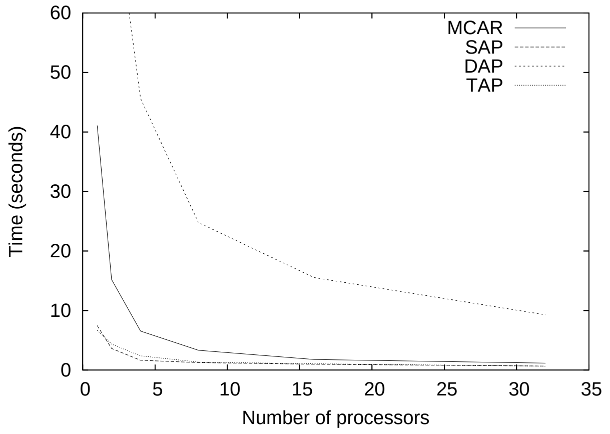

Figure 6. Performance of P3VI as a function of number of processors.

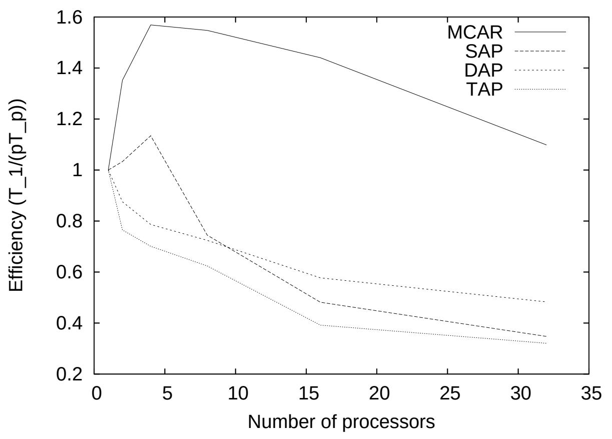

Figure 7. Efficiency of P3VI. Higher efficiency is better.

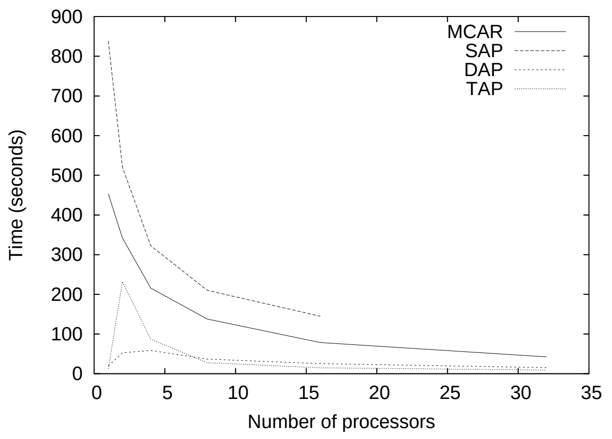

Figure 8. Performance of PSVI as a function of number of processors.

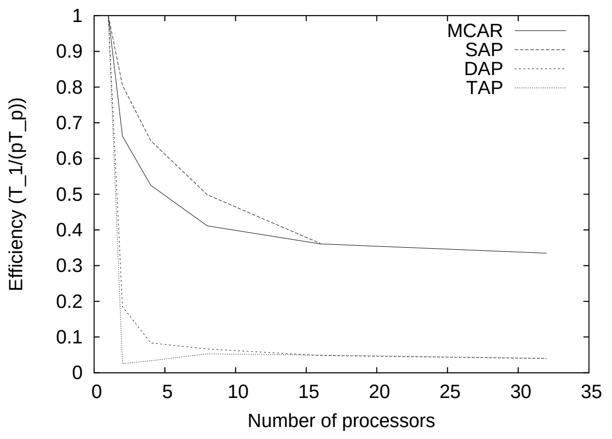

Figure 9. Efficiency of PSVI. Higher efficiency is better.

and quantify the performance impacts of different edge cut criteria. The second most pressing issue is to develop a more theoretically sound method of allocating partitions to processors, possibly even exploring the idea of a dynamic allocation of partitions to processors. Finally, although P3VI outperformed all other algorithms, the efficiency measures indicate that there is room for further improvement.

References

[1] Charles J. Alpert. *Multi-way graph and hypergraph partitioning.* PhD thesis, UCLA, Los Angeles, California, 1996.

[2] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.

[3] Dimitri P. Bertsekas. Distributed dynamic programming. *IEEE Transactions on Automatic Control*, 27:610–616, 1982.

[4] George Karypis and Vipin Kumar. Parallel Multilevel k-way Partitioning Schemes for Irregular Graphs. Technical Report 036, Minneapolis, MN 55454, May 1996.

[5] Michael J. Kearns and Satinder P. Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49:209–232.

[6] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.

[7] Remi Muños and Andrew W. Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49 (2-3):291–323, 2002.

[8] Martin L. Puterman. *Markov Decision Processes–Discrete Stochastic Dynamic Programming.* John Wiley and Sons, Inc., New York, NY, 1994.

[9] Martin L. Puterman and M.C. Shin. Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, pages 1127–1137, 1978.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[11] David Wingate and Kevin Seppi. Efficient value iteration using partitioned models. In *International Conference on Machine Learning and Applications*, pages 53–59, 2003.