Stochastic Reactive Production Scheduling by Multi-agent Based Asynchronous Approximate Dynamic Programming

Balázs Csanád Csáji[1] and László Monostori[1,2]

[1] Computer and Automation Research Institute,
Hungarian Academy of Sciences
[2] Faculty of Mechanical Engineering,
Budapest University of Technology and Economics
{csaji, monostor}@sztaki.hu

Abstract. The paper investigates a stochastic production scheduling problem with unrelated parallel machines. A closed-loop scheduling technique is presented that on-line controls the production process. To achieve this, the scheduling problem is reformulated as a special Markov Decision Process. A near-optimal control policy of the resulted MDP is calculated in a homogeneous multi-agent system. Each agent applies a trial-based approximate dynamic programming method. Different cooperation techniques to distribute the value function computation among the agents are described. Finally, some benchmark experimental results are shown.

1 Introduction

Scheduling is the allocation of resources over time to perform a collection of tasks. Near-optimal scheduling is a prerequisite for the efficient utilization of resources and, hence, for the profitability of the enterprise. Therefore, scheduling is one of the key problems in a manufacturing production control system. Moreover, much that can be learned about scheduling can be applied to other kinds of planning and decision making, therefore, it has general practical value.

The paper suggests an agent-based closed-loop solution to a stochastic scheduling problem that can use information, such as actual processing times, as they become available, and can control the production process on-line. For this reason, the stochastic scheduling problem is reformulated as a Markov Decision Process. Machine learning techniques, such as asynchronous approximate dynamic programming (namely approximate Q-learning with prioritized sweeping), are suggested to compute a good policy in a homogeneous multi-agent system.

Using approximate dynamic programming (also called as reinforcement learning) for job-shop scheduling was first proposed in [12]. They used the $TD(\lambda)$ method with iterative repair to solve a static scheduling problem, namely the NASA space shuttle payload processing problem. Since then, a number of papers have been published that suggested using reinforcement learning for scheduling problems. However, most of them investigated static and deterministic problems, only, and the suggested solutions were mostly centralized. A reinforcement

M. Pěchouček, P. Petta, and L.Z. Varga (Eds.): CEEMAS 2005, LNAI 3690, pp. 388–397, 2005.

learning based centralized closed-loop production scheduling approach was first briefly described in [10]. Recently, several machine learning improvements of multi-agent based scheduling systems were proposed, for example [2] and [3].

2 Production Scheduling Problems

First, a static deterministic scheduling problem with unrelated parallel machines is considered: an instance of the problem consists of a finite set of *tasks* $\mathcal{T}$ together with a partial ordering $\mathcal{C} \subseteq \mathcal{T} \times \mathcal{T}$ that represents the *precedence constraints* between the tasks. A finite set of *machines* $\mathcal{M}$ is also given with a partial function that defines the *durations* (or processing times) of the tasks on the machines, $d : \mathcal{T} \times \mathcal{M} \to \mathbb{N}$. The tasks are supposed to be non-preemptive (they may not be interrupted) thus a *schedule* can be defined as an ordered pair $\langle \varrho, \mu \rangle$ where $\varrho : \mathcal{T} \to \mathbb{N}_0$ gives the starting (release) times of the tasks ($\mathbb{N}_0 = \mathbb{N} \cup \{0\}$), and $\mu : \mathcal{T} \to \mathcal{M}$ defines which machine will process which task. A schedule is called *feasible* if and only if the following three properties are satisfied:

(s1) Each machine processes at most one operation at a time:
$\neg\exists (m \in \mathcal{M} \wedge u, v \in \mathcal{T}) : \mu(u) = \mu(v) = m \wedge\ \varrho(u) \leq \varrho(v) < \varrho(u) + d(u, m)$
(s2) Every machine can process the tasks which were assigned to it:
$\forall v \in \mathcal{T} : \langle v, \mu(v) \rangle \in dom(d)$
(s3) The precedence constraints of the tasks are kept:
$\forall \langle u, v \rangle \in \mathcal{C} : \varrho(u) + d(u, \mu(u)) \leq \varrho(v)$

Note that $dom(d) \subseteq \mathcal{T} \times \mathcal{M}$ denotes the domain set of the function d. The set of all feasible schedules is denoted by S, which is supposed to be non-empty (thus, e.g., $\forall v \in \mathcal{T} : \exists\, m \in \mathcal{M} : \langle v, m \rangle \in dom(d)$). The objective of scheduling is to produce a schedule that minimizes a *performance measure* $\kappa : S \to \mathbb{R}$, which usually depends on the task completion times, only. For example, if the completion time of the task $v \in \mathcal{T}$ is denoted by $C(v) = \varrho(v) + d(v, \mu(v))$ then a commonly used performance measure, which is often called total production time or make-span, can be defined by $C_{max} = \max\{C(v) \mid v \in \mathcal{T}\}$.

However, not *any* function is allowed as a performance measure. These measures are restricted to functions which have the property that a schedule can be uniquely generated from the order in which the jobs are processed through the machines, e.g., by semi-active *timetabling*. *Regular* measures, which are monotonic in completion times, have this property. Note that all of the commonly used performance measures (e.g., maximum completion time, mean flow time, mean tardiness, etc.) are regular. As a consequence, S can be safely restricted to these schedules and, therefore, S will be finite, thus the problem becomes a *combinatorial optimization* problem characterized by the 5-tuple $\langle \mathcal{T}, \mathcal{M}, \mathcal{C}, d, \kappa \rangle$.

It is easy to see that the presented parallel machine scheduling problem is a generalization of the standard *job-shop* scheduling problem which is known to be *strongly NP-hard* [7], consequently, this problem is also strongly NP-hard. Moreover, if the used performance measure is C_{max}, there is no good polynomial time approximation of the optimal scheduling algorithm [9]. Therefore, in practice, we have to satisfy with sub-optimal (approximate) solutions.

The stochastic variant of the presented problem arises, when the durations are given by independent finite random variables. Thus, $d(v, m)$ denotes a random variable with possible values $d_{vm1}, \ldots, d_{vmk}$ and with probability distribution $p_{vm1}, \ldots, p_{vmk}$. Note that $k = k(v, m)$, it can depend on v and m. If the functions ϱ and μ are given, we write d_{vi} and p_{vi} for abbreviation of $d_{v\mu(v)i}$ and $p_{v\mu(v)i}$. In this case, the performance of a schedule is also a random variable.

In stochastic scheduling there are some data (e.g. the actual durations) that will only be available during the execution of the plan. According to the usage of these information, we consider two basic types of scheduling techniques.

A *static* (*open-loop*, *proactive* or *off-line*) scheduler has to make all decisions before the schedule actually being executed and it cannot take the actual evolution of the process into account. It has to build a schedule that can be executed with high probability. For a *dynamic* (*closed-loop*, *reactive* or *on-line*) scheduler it is allowed to make the decisions as the scheduling process actually evolves and more information becomes available. In this paper we will focus on dynamic techniques and will formulate the stochastic scheduling problem as a Markov Decision Process. Note that a dynamic solution is not a simple $\langle \varrho, \mu \rangle$ pair, but instead a scheduling *policy* (defined later) which controls the production.

3 Markov Decision Processes

Sequential decision making under uncertainty is often modeled using MDPs. This section contains the basic definitions and some preliminaries. By a (finite state, discrete time, stationary, fully observable) *Markov Decision Process* (MDP) we mean a 8-tuple $\langle \mathbb{S}, \mathbb{T}, \mathbb{A}, \mathcal{A}, p, g, \alpha, \beta \rangle$, where the components are:

(m1) $\mathbb{S}$ is a finite set of discrete *states*.
(m2) $\mathbb{T} \subseteq \mathbb{S}$ is a set of *terminal states*.
(m3) $\mathbb{A}$ is a finite set of control *actions*.
(m4) $\mathcal{A} : \mathbb{S} \to \mathcal{P}(\mathbb{A})$ is an *availability function* that renders each state a set of control actions available in that state. Note that $\mathcal{P}$ denotes the power set.
(m5) $p : \mathbb{S} \times \mathbb{A} \to \Delta(\mathbb{S})$ is a *transition function*, where $\Delta(\mathbb{S})$ is the space of probability distributions over $\mathbb{S}$. We denote by $p_{ss'}(a)$ the probability of arriving to state s' after executing control action $a \in \mathcal{A}(s)$ in a state s.
(m6) $g : \mathbb{S} \times \mathbb{A} \times \mathbb{S} \to \mathbb{R}$ is an *immediate cost* (or reward) function, $g(s, a, s')$ is the cost of moving from state s to state s' with control action $a \in \mathcal{A}(s)$.
(m7) $\alpha \in [0, 1]$ is a *discount rate* or also called *discount factor*. If $\alpha = 1$ then the MDP is called *undiscounted* otherwise it is *discounted*.
(m8) $\beta \in \Delta(\mathbb{S})$ is an *initial probability distribution*.

An interpretation of a MDP can be given if we consider an agent that acts in a stochastic environment. The agent receives information about the state of the environment $s \in \mathbb{S}$. At each state s the agent can choose an action $a \in \mathcal{A}(s)$. After the action is selected the environment moves to the next state according to the probability distribution $p(s, a)$ and the decision-maker collects its one-step penalty (cost). The aim of the agent is to find an optimal control policy

that minimizes the expected cumulative costs over an infinite horizon or until it reaches an absorbing terminal state. The set of terminal states can be empty. Theoretically, the terminal states can be treated as states with only one available control action that loops back to them with probability 1 and cost 0.

A (stationary, randomized, Markov) control *policy* $\pi : \mathbb{S} \to \Delta(\mathbb{A})$ is a function from states to probability distributions over actions. We denote by $\pi(s, a)$ the probability of executing control action $a \in \mathcal{A}(s)$ in the state $s \in \mathbb{S}$.

The initial probability distribution β, the transition probabilities p together with a control policy π completely determine the progress of the system in a stochastic sense, namely, it defines a homogeneous Markov chain on $\mathbb{S}$.

The *cost-to-go* or *value* function of a policy is $J^\pi : \mathbb{S} \to \mathbb{R}$, where $J^\pi(s)$ gives the expected costs when the system is in state s and it follows π thereafter:

$$J^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \alpha^t g(s_t, a_t, s_{t+1}) \;\middle|\; s_0 = s \right], \tag{1}$$

whenever this expectation is well-defined. Naturally, it is always well-defined if $\alpha < 1$. Here, we consider problems with expected total [un]discounted cost, only.

A policy $\pi_1 \leq \pi_2$ if and only if $\forall s \in \mathbb{S} : J^{\pi_1}(s) \leq J^{\pi_2}(s)$. A policy is called (uniformly) *optimal* if it is better than or equal to all other control policies.

There always exits at least one optimal stationary deterministic control policy. Although, there may be many optimal policies, they all share the same unique optimal cost-to-go function, denoted by J^*. This function must satisfy the (Hamilton-Jacoby-) *Bellman optimality equation* [1] for all $s \in \mathbb{S}$:

$$J^*(s) = \min_{a \in \mathcal{A}(s)} \sum_{s' \in \mathbb{S}} p_{ss'}(a) \left[g(s, a, s') + \alpha J^*(s') \right] \tag{2}$$

Note that from a given cost-to-go function it is straightforward to get a control policy, for example, by selecting in each state in a deterministic and greedy way an action that produces minimal costs with one-stage lookahead. The problem of finding a good policy will be further investigated in Section 5.

4 Stochastic Reactive Scheduling as a MDP

In this section a dynamic stochastic scheduling problem is formulated as a Markov Decision Process. The actual task durations will be only incrementally available during production and the decisions will be made on-line.

A state $s \in \mathbb{S}$ is defined as a 6-tuple: $s = \langle t, \mathcal{T}_S, \mathcal{T}_F, \varrho, \mu, \varphi \rangle$, where $t \in \mathbb{N}_0$ is the actual time, $\mathcal{T}_S \subseteq \mathcal{T}$ is the set of tasks which have been started before time t and $\mathcal{T}_F \subseteq \mathcal{T}_S$ is the set of tasks that have been finished, already. The functions $\varrho : \mathcal{T}_S \to \mathbb{N}_0$ and $\mu : \mathcal{T}_S \to \mathcal{M}$, as previously, give the starting times of the tasks and the task-machine assignments. The function $\varphi : \mathcal{T}_F \to \mathbb{N}$ stores the task completion times. We also define a starting state $s_0 = \langle 0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$, that corresponds to the situation at time 0 when none of the tasks have been started. The initial probability distribution β renders 1 to the starting state s_0.

We introduce a set of terminal states, as well. A state $s = \langle t, \mathcal{T}_S, \mathcal{T}_F, \varrho, \mu, \varphi \rangle$ is considered as a terminal state if and only if $\mathcal{T}_F = \mathcal{T}$ and it can be reached from a state $\hat{s} = \langle \hat{t}, \mathcal{T}'_S, \mathcal{T}'_F, \hat{\varrho}, \hat{\mu}, \hat{\varphi} \rangle$ where $\mathcal{T}'_F \neq \mathcal{T}$. If the system reaches a terminal state (all tasks are finished), then we treat the control process completed.

At every time t the system is informed which tasks have been finished, and it can decide which unscheduled tasks it starts (and on which machines).

The control action space contains task-machine assignments $a_{vm} \in \mathbb{A}$, where $v \in \mathcal{T}$ and $m \in \mathcal{M}$, and a special a_{wait} control that corresponds to the action when the system does not start a new task at the present time.

In a non-terminal state $s = \langle t, \mathcal{T}_S, \mathcal{T}_F, \varrho, \mu, \varphi \rangle$ the available actions are:

(a1) $a_{wait} \in \mathcal{A}(s) \Leftrightarrow \mathcal{T}_S \setminus \mathcal{T}_F \neq \emptyset$
(a2) $\forall v \in \mathcal{T} : \forall m \in \mathcal{M} : a_{vm} \in \mathcal{A}(s) \Leftrightarrow (v \in \mathcal{T} \setminus \mathcal{T}_S \wedge \forall u \in \mathcal{T}_S \setminus \mathcal{T}_F :$
$m \neq \mu(u) \wedge \langle v, m \rangle \in dom(d) \wedge \forall u \in \mathcal{T} : (\langle u, v \rangle \in \mathcal{C}) \Rightarrow (u \in \mathcal{T}_F))$

If an $a_{vm} \in \mathcal{A}(s)$ is executed in a state $s = \langle t, \mathcal{T}_S, \mathcal{T}_F, \varrho, \mu, \varphi \rangle$, the system moves with probability 1 to a new state $\hat{s} = \langle t, \mathcal{T}'_S, \mathcal{T}'_F, \hat{\varrho}, \hat{\mu}, \hat{\varphi} \rangle$, where $\mathcal{T}'_F = \mathcal{T}_F$, $\mathcal{T}'_S = \mathcal{T}_S \cup \{v\}$, $\hat{\varrho}\,\big|_{\mathcal{T}_S} = \varrho$, $\hat{\mu}\,\big|_{\mathcal{T}_S} = \mu$, $\hat{\varrho}(v) = t$, $\hat{\mu}(v) = m$ and $\varphi = \hat{\varphi}$.

The effect of the a_{wait} action is that it takes from $s = \langle t, \mathcal{T}_S, \mathcal{T}_F, \varrho, \mu, \varphi \rangle$ to a state $\hat{s} = \langle t + 1, \mathcal{T}_S, \mathcal{T}'_F, \varrho, \mu, \hat{\varphi} \rangle$ where $\mathcal{T}_F \subseteq \mathcal{T}'_F \subseteq \mathcal{T}_S$ and for all $v \in \mathcal{T}_S \setminus \mathcal{T}_F$: the task v will be in $\mathcal{T}'_F$ (v terminates) with probability as follows:

$$\mathbb{P}(v \in \mathcal{T}'_F \mid s) = \mathbb{P}(F(v) = t \mid F(v) \geq t) = \frac{\sum_{i=1}^{k} p_{vi}\, \mathbb{I}(f_i(v) = t)}{\sum_{i=1}^{k} p_{vi}\, \mathbb{I}(f_i(v) \geq t)}, \qquad (3)$$

where $F(v)$ is a random variable that gives the finish time of task v (according to $\langle \varrho, \mu \rangle$), $f_i(v) = \varrho(v) + d_{vi}$ and $\mathbb{I}$ is an indicator function, viz. $\mathbb{I}(A) = 1$ if A is true, otherwise it is 0. Recall that $p_{vi} = p_{vmi}$ and $d_{vi} = d_{vmi}$, where $m = \mu(v)$; k can also depend on v and m; $\hat{\varphi}\,\big|_{\mathcal{T}_F} = \varphi$, $\forall v \in \mathcal{T}_F \setminus \mathcal{T}'_F : \varphi(v) = t$.

The cost function, for a given κ performance measure (which depends only on the task completion times), is defined as follows. Let $s = \langle t, \mathcal{T}_S, \mathcal{T}_F, \varrho, \mu, \varphi \rangle$ and $\hat{s} = \langle \hat{t}, \mathcal{T}'_S, \mathcal{T}'_F, \hat{\varrho}, \hat{\mu}, \hat{\varphi} \rangle$. Then $\forall a \in \mathcal{A}(s) : g(s, a, \hat{s}) = \kappa(\varphi) - \kappa(\hat{\varphi})$.

It is easy to see that the MDPs defined by this way have finite state spaces and their transition graphs are acyclic. Therefore, these MDPs have a *finite horizon* and, thus, the discount rate α can be safely set to 1, without risking that the expectation in the cost-to-go function becomes not well-defined. Note that these type of problems are often called *Stochastic Shortest Path* (SSP) problems. For the effective computation of a control policy it is important to try reducing the number of states. Domain specific knowledge can help to achieve this: if κ is non-decreasing in the completion times (which is mostly the case in practice), then an optimal policy can be found among those policies which only start new tasks at times when another task has been finished or at the initial state s_0.

5 Approximate Dynamic Programming

In the previous section we have formulated a dynamic production scheduling task as an acyclic stochastic shortest path problem (a special MDP). Now, we

face the challenge of finding a good control policy. We suggest a homogeneous multi-agent system in which the optimal policy is calculated in a distributed way. First, we describe the operation of a single adaptive agent that tries to learn the optimal value function with Watkins' Q-learning. Next, we examine different cooperation techniques to distribute the value function computation.

In theory, the optimal value function of a finite MDP can be computed exactly by dynamic programming methods, such as value iteration or the Gauss-Seidel method. Alternatively, an exact optimal policy can be directly calculated by policy iteration. However, due to the "curse of dimensionality" (viz. in practical situations both the needed memory and the required amount of computation is extremely large) calculating an exact optimal solution by these methods is practically infeasible. We should use Approximate Dynamic Programming (ADP) techniques to achieve a good approximation of an optimal control policy.

The paper suggests using the Q-learning algorithm to calculate a near optimal policy. Like most ADP methods, the aim of Q-learning is also to learn the optimal value function rather than directly learning an optimal control policy. The Q-learning method learns state-action value functions, which are defined by:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \alpha^t g(s_t, a_t, s_{t+1}) \;\middle|\; s_0 = s, a_0 = a\right] \tag{4}$$

An agent can search in the space of feasible schedules by simulating the possible occurrences of the production process with the model. The trials of the agent can be described as state-action pair trajectories. After each episode the agent makes updates *asynchronously* on the approximated values of the visited pairs. Only a subset of all pairs are updated in each trial. Note that the agent does not need a uniformly good approximation on all possible pairs, but instead on the *relevant* ones which can appear with positive probability during the executing of an optimal policy. Therefore, it can always start the simulation from s_0.

The general version of the one-step Q-learning rule can be formulated as:

$$Q_{t+1}(s,a) = Q_t(s,a) + \gamma_t(s,a)\left[g(s,a,s') - Q_t(s,a) + \alpha \min_{b \in \mathcal{A}(s')} Q_t(s',b)\right], \tag{5}$$

where s' and $g(s,a,s')$ are generated from the pair (s,a) by simulation, that is, according to the transition probabilities $p_{ss'}(a)$; $\gamma_t(s,a)$ are sequences that define the *learning rates* of the system. Q-learning can also be seen as a Robbins-Monro type stochastic approximation method. Note that it is advised to apply *prioritized sweeping* during backups. Q-learning is called an *off-policy* method, which means that the value function converges almost surely to the optimal state-action value function *independently* of the policy being followed or the starting Q values. It is known [1], that if the learning rates satisfy: $\sum_{t=1}^{\infty} \gamma_t(s,a) = \infty$ and $\sum_{t=1}^{\infty} \gamma_t^2(s,a) < \infty$ for all s and a, the Q-learning algorithm will converge with probability one to the optimal value function in the case of lookup table representation (namely, the value of each pair is stored independently).

However, in systems with large state spaces, it is not possible to store an estimation for each state-action pair. The value function should be approximated

by a parametric function. We suggest a *Support Vector Machine* (SVM) based regression for maintaining the Q function, as in [4], which then takes the form:

$$Q(s,a) \approx \tilde{Q}(x,w,b) = \sum_{i=1}^{n} w_i K(x,x_i) + b, \tag{6}$$

where $x = \phi(s,a)$ represents some peculiar features of s and a, x_i denotes the features of the training data, b is a bias, K is the *kernel* function and $w \in \mathbb{R}^n$ is the parameter vector of the approximation. As a kernel we choose a Gaussian type function $K(x_1,x_2) = \exp(-\|x_1 - x_2\|^2/\sigma^2)$. Basically, an SVM is an approximate implementation of the *method of structural risk minimization*. Recently, several on-line, incremental methods have been suggested that made SVMs applicable for reinforcement learning. For more details, see [8].

Now, we give some ideas about the possible features that can be used in the stochastic scheduling case. Concerning the environment: expected relative ready time of each machine with their standard deviations and the estimated relative future load of the machines. Regarding the chosen action (task-machine assignment): its expected relative finish time with its deviation and the cumulative estimated relative finish time of the tasks, which succeeds the selected task.

In order to ensure the convergence of the Q-learning algorithm, one must guarantee that each state-action pair is continue to update. An often used technique to balance between *exploration* and *exploitation* is the *Boltzmann formula*:

$$\pi(s,a) = \frac{\exp(\tau/Q(s,a))}{\sum_{b \in \mathcal{A}(s)} \exp(\tau/Q(s,b))}, \tag{7}$$

where $\tau \geq 0$ is the Boltzmann (or Gibbs) temperature. Low temperatures cause the actions to be (nearly) equiprobable, high ones cause a greater difference in selection probability for actions that differ in their value estimations. Note that here we applied the Boltzmann formula for minimization, viz. small values mean high probability. Also note that it is advised to extend this approach by a variant of *simulated annealing*, which means that τ should be increased over time.

6 Distributed Value Function Computation

In the previous section we have described the learning mechanism of a single agent. In this section we examine cooperation techniques in homogeneous multi-agent systems to distribute the computation of the optimal value function. Our suggested architectures are *heterarchical*, in which the agents communicate as peers and no master/slave relationships exist. The advantages of these systems include: self-configuration, scalability, fault tolerance, massive parallelism, reduced complexity, increased flexibility, reduced cost and emergent behavior [11].

An agent-based (holonic) reference architecture for manufacturing systems is PROSA [5]. The general idea underlying this approach is to consider both the machines and the jobs (sets of tasks) as active entities. There are three

types of standard agents in PROSA: order agents (internal logistics), product agents (process plans), and resource agents (resource handling). In a further improvement of this architecture the system is extended with mobile agents, called ants. As we have shown in [2], it is advised to extend the ant-colony based approach with ADP techniques. Another way for scheduling with PROSA is to use some kind of market or negotiation mechanism. We have presented a market-based scheduling approach with competitive adaptive agents in [3].

Now, we return to our original approach and present ways to distribute the value function calculation. The suggested multi-agent architectures are homogeneous, therefore, all of the agents are identical. The agents work independently by making their trials in the simulated environment, but they share information.

If a common (global) storage is available to the agents, then it is straightforward to parallelize the value function computation: each agent searches independently by making trials, however, they all share (read and write) the same value function. They update the value function estimations asynchronously.

A more complex situation arises when the memory is completely local to the agents, which is realistic if they are physically separated (e.g. they run on different computers). For that case, we suggest two cooperation techniques. A way of dividing the computation of a good policy among several agents is when there is only one "global" value function, however, it is stored in a distributed way. Each agent stores a part of the value function and it asks for estimations which it requires but does not have from the other agents. The applicability of this approach lies in the fact that the underlying MDP is acyclic and, thus, it can be effectively partitioned among the agents, for example, by starting each agent from a different starting state. Partitioning the search space can be very useful for the other distributed ADP approaches, as well. The policy can be then computed by using the aggregated value function estimations of the agents.

Another approach is, when the agents have their own completely local value functions and, consequently, they could have widely different estimations on the optimal state-action values. In that case, the agents should count that how many times did they update the estimations of the different pairs. Finally, the values of the global Q-function can be combined from the estimations of the agents:

$$Q(s,a) = \sum_{i=1}^{n} w_i(s,a)\, Q_i(s,a), \qquad w_i(s,a) = \frac{\exp(h_i(s,a)/\eta)}{\sum_{j=1}^{n} \exp(h_j(s,a)/\eta)}, \tag{8}$$

where n is the number of agents, Q_i is the state-action value function of agent i, $h_i(s,a)$ contains the number of how many times did agent i update its estimation for the (s,a) pair and $\eta > 0$ is an adjustable parameter. Naturally, for large state spaces, the counter functions can be parametrically approximated, as well.

The agents can also help each other by communicating estimation information, episodes, policies, etc. A promising way of cooperation is, when the agents periodically exchange a fixed number of their best episodes after an adjustable amount of trials and, by this way, they help improving each others value functions. After an agent receives an episode (a sequence of states), it updates its value function estimation as if this state trajectory was produced by itself.

7 Experimental Results

We have tested our ADP based approach on Hurink's benchmark dataset [6]. It contains flexible job-shop scheduling problems with 6-30 jobs (30-225 tasks) and 5-15 machines. These problems are "hard", which means, for example, that standard dispatching rules or heuristics perform poorly on them. This dataset consists of four subsets, each subset contains about 60 problems. The subsets (sdata, edata, rdata, vdata) differ on the ratio of machine interchangeability, which are shown in the "parallel" column in the table (left part of Figure 1). The columns with label "x es" show the global error after carrying out altogether "x" episodes. The execution of 10000 simulated trials (after on the average the system has achieved a solution with less than 5% global error) takes only a few seconds on a computer of our day. In the tests we have used a decision-tree based state-aggregation. The left part of Figure 1 shows the results of a single agent.

dataset	parallel	1000 es	5000 es	10000 es
sdata	1	8.54 %	5.69 %	3.57 %
edata	1.2	12.37 %	8.03 %	5.26 %
rdata	2	16.14 %	11.41 %	7.14 %
vdata	5	10.18 %	7.73 %	3.49 %
average	2.3	11.81 %	8.21 %	4.86 %

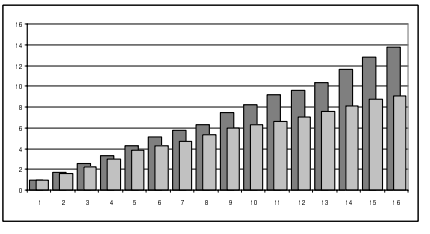

Fig. 1. Benchmarks; left: average global error on a dataset of "hard" flexible job-shop problems; right: average speedup (y axis) relative to the number of agents (x axis); dark grey bars: global value function; light grey bars: local value functions

We have also investigated the speedup of the system relative to the number of agents. The average number of iterations was studied, until the system could reach a solution with less than 5% global error on Hurink's dataset. We have treated the average speed of a single agent as a unit. In the right part of Figure 1 two cases are shown: in the first case, all of the agents could access a global value function. In that case, the speedup was almost linear. In the second case, each agent had its own (local) value function and, after the search was finished, the individual functions were combined. The experiments show, that the computation of the ADP based scheduling technique can be effectively distributed among several agents, even if they do not have a commonly accessible value function.

8 Concluding Remarks

Efficient allocation of manufacturing resources over time is one of the key problems in a production control system. The paper has presented an approximate dynamic programming based stochastic reactive scheduler that can control the production process on-line, instead of generating an off-line rigid static plan. To

achieve closed-loop control, the stochastic scheduling problem was formulated as a special Markov Decision Process. To compute a (near) optimal control policy, homogeneous multi-agent systems were suggested, in which cooperative agents learn the optimal value function in a distributed way by using trial-based ADP methods. After each trial, the agents asynchronously update the actual value function estimation according to the Q-learning rule with prioritized sweeping. For large state spaces a Support Vector Machine regression based value function approximation was suggested. Finally, the paper has shown some benchmark results on Hurink's flexible job-shop dataset, which illustrate the effectiveness of the ADP based approach, even in the case of deterministic problems.

Acknowledgements

This research was partially supported by the National Research and Development Programme (NKFP), Hungary, Grant No. 2/010/2004 and by the Hungarian Scientic Research Fund (OTKA), Grant Nos. T049481 and T043547.

References

1. Bertsekas, D. P., Tsitsiklis J. N.: Neuro-Dynamic Programming (1996)
2. Csáji, B. Cs., Kádár, B., Monostori, L.: Improving Multi-Agent Based Scheduling by Neurodynamic Programming. Holonic and Mult-Agent Systems for Manufacturing, Lecture Notes in Computer Science **2744**, HoloMAS: Industrial Applications of Holonic and Multi-Agent Systems (2003) 110–123
3. Csáji, B. Cs., Monostori, L., Kádár, B.: Learning and Cooperation in a Distributed Market-Based Production Control System. Proceedings of the 5th International Workshop on Emergent Synthesis (2004) 109–116
4. Dietterich, T. G., Xin Wang: Batch Value Function Approximation via Support Vectors. Advances in Neural Information Processing Systems **14** (2001) 1491–1498
5. Hadeli, Valckenaers, P., Kollingbaum, M., Van Brussel, H.: Multi-Agent Coordination and Control Using Stigmergy. Computers in Industry **53** (2004) 75–96.
6. Hurink, E., Jurisch, B., Thole, M.: Tabu Search for the Job Shop Scheduling Problem with Multi-Purpose Machine. Operations Research Spektrum **15** (1994) 205–215
7. Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G., Shmoys, D. B.: Sequencing and Scheduling: Algorithms and Complexity. Handbooks in Operations Research and Management Science (1993)
8. Martin., M.: On-line Support Vector Machine Regression. Proceedings of the 13th European Conference on Machine Learning (2002) 282–294
9. Williamson, D. P., Hall L. A., Hoogeveen, J. A., Hurkens, C. A. J., Lenstra, J. K., Sevastjanov, S. V., Shmoys, D. B.: Short Shop Schedules. Operations Research **45** (1997) 288–294
10. Schneider, J., Boyan, J., Moore, A.: Value Function Based Production Scheduling. Proceedings of the 15th International Conference on Machine Learning (1998)
11. Ueda, K., Márkus, A., Monostori, L., Kals, H. J. J., Arai, T.: Emergent Synthesis Methodologies for Manufacturing. Annals of the CIRP **50** (2001) 535–551
12. Zhang, W., Dietterich, T.: A Reinforcement Learning Approach to Job-Shop Scheduling. IJCAI: Proceedings of the 14th International Joint Conference on Artificial Intelligence (1995) 1114–1120