Beschleunigung eines Reinforcement-Learning-Algorithmus durch Parallelverarbeitung für Robotikanwendungen

Conference Paper · March 2019

CITATIONS
2

READS
105

4 authors, including:

David Jammer
Hochschule Wismar
1 PUBLICATION **2** CITATIONS
SEE PROFILE

Georg Kunert
Hochschule Wismar
7 PUBLICATIONS **18** CITATIONS
SEE PROFILE

Thorsten Pawletta
Hochschule Wismar
122 PUBLICATIONS **562** CITATIONS
SEE PROFILE

Beschleunigung eines Reinforcement-Learning-Algorithmus durch Parallelverarbeitung für Robotikanwendungen

David Jammer[1], Sven Pawletta[1*], Georg Kunert[1], Thorsten Pawletta[1]

[1]Hochschule Wismar – University of Applied Science, Forschungsgruppe CEA
23966 Wismar; *sven.pawletta@hs-wismar.de*

Abstract. Machine-Learning (ML) findet derzeit auch in ingenieur-technischen Anwendungen eine große Beachtung. Als problematisch erweist sich dabei häufig der erforderliche Rechenaufwand. Im Beitrag werden Beschleunigungspotentiale für ein Reinforcement-Learning-Verfahren untersucht. Neben Ansätzen der Parallelverarbeitung wird auch das Beschleunigungspotential von Laufzeitumgebungen und effizienten Suchalgorithmen betrachtet.

Einleitung

In der Forschungsgruppe *Computational Engineering & Automation* (FG CEA) der Hochschule Wismar werden seit mehr als 10 Jahren Projekte zum Einsatz von Industrierobotern in der flexiblen Fertigung und Montage durchgeführt. Konzeptionell basieren die Arbeiten auf dem Framework *Simulation Based Control* (SBC,[1]). Das SBC-Framework unterstützt die Methodik des *Rapid Control Prototypings* (RCP, [2]).

Als primäre Software-Plattform für Prototyping sowie operativen Betrieb stehen beim SBC *Scientific Computing Environments* (SCEs) im Mittelpunkt. Bei den SCEs handelt es sich um interpreter-basierte Systeme, die einen komfortablen Zugriff auf umfangreiche Bibliotheken des numerischen und symbolischen Rechnens in Verbindung mit einer matrix-orientierten Programmiersprache bieten. Der Quasistandard in diesem Bereich wird durch das kommerzielle System Matlab von The Mathworks Inc. definiert. Alternativ stehen freie Systeme wie Octave, Scilab, FreeMat und andere zur Verfügung. Höhere interpretierende Sprachen wie Python können ebenfalls zur Klasse der SCEs gezählt werden.

In [3] wurde über ein Projekt der FG CEA zur Generierung von Robotersteuerungen unter Verwendung eines ML-Verfahrens berichtet. Konkret wurde das Lernverfahren *Reinforcement* (RL) in der Ausprägung *Q-Learning* (QL) in der Variante *Value Based Learning* (VBL) betrachtet und die Integration in das SBC-Framework dargestellt.

Die grundsätzliche Anwendbarkeit von ML-Verfahren im Rahmen des SBC-Frameworks wurde in [3] an einem Beispielproblem geringer Komplexität demonstriert und nachgewiesen (*Türme von Hanoi*, TvH). Für die Anwendung von ML-Verfahren in der Praxis stellt sich jedoch regelmäßig die Frage, welche rechentechnischen Ressourcen für Probleme realer Komplexität erforderlich sind.

Dieser Frage widmet sich der vorliegende Beitrag. Dazu wird eingangs das in [3] gewählte ML-Verfahren soweit eingeführt, dass ein Verständnis der Nachfolgenden Betrachtungen möglich ist.

Anschließend wird anhand des Studienbeispiels TvH die Komplexitätsproblematik dargestellt. Im Kontext realer Anwendungen führt eine hohe Komplexität im Zusammenhang mit ML-Verfahren zwangsläufig zu einem sehr hohen Rechenaufwand und damit zu langen Laufzeiten. In der Praxis ist die maximal erlaubte Rechenzeiten meist begrenzt. Wenn diese Schranke durch eine konventionelle Implementierung nicht eingehalten werden kann, wird heute häufig versucht, eine Lösung durch *Parallelverarbeitung* (PV) auf Basis der Multi-Core-Technologie zu erzielen. Tatsächlich sind die erreichbaren Beschleunigungsfaktoren auf Ein-Prozessor-Computern aber noch relativ begrenzt. Die Core-Zahl der gegenwärtig leistungsstärksten Prozessoren von AMD und Intel ist 32 beziehungsweise 28.

Im vorliegenden Beitrag werden deshalb zuerst Beschleunigungspotentiale untersucht, die einer PV

vorgelagert werden sollten. Das sind (i) der Wechsel der Laufzeitumgebung und (ii) die Implementierung einer effizienten Algorithmik. Erst zum Schluss (iii) wird auf den Einsatz einer geeigneten PV-Technologie eingegangen.

1 Reinforcement Learning

Gegenstand der Untersuchung in [3] war eine Variante des RL, die als Q-Learning bezeichnet wird. Q-Learning kann nach [4, 5] wiederum nach verschieden Ansätzen durchgeführt werden. Der in [3] und hier betrachtete Ansatz ist das Value Based Learning (VBL). Die Grundstruktur des Verfahrens zeigt Abbildung 1.

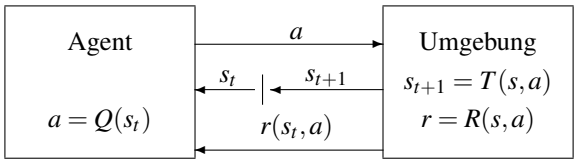


Abbildung 1: Grundstruktur des RL/QL/VBL nach [3]

Der Agent besitzt anfangs kein Wissen über seine Umgebung. Initial kommuniziert die Umgebung ihren momentanen Zustand s_t an den Agenten. Aus Sicht des Agenten kann s_t als Anfangszustand der Umgebung für eine beginnende Explorationsphase interpretiert werden. Die Abbildung $a = Q(s_t)$ steht im Agenten für das erlernte Wissen über die Umgebung. Verfügt der Agent über ausreichendes Wissen, so kann er auf Basis des momentanen Zustands s_t eine Aktion a auswählen und an die Umgebung kommunizieren, die dort zu einer sinnvollen Zustandsänderung im Sinne einer Zielvorgabe führt. In der Regel besteht die Zielvorgabe im Erreichen eines bestimmten Zielzustandes über den Weg möglichst weniger Zustandsänderungen.

Anfangs kann der Agent Aktionen a nur zufällig generieren, weil er noch kein Wissen besitzt. Die in der Umgebung auf a folgende Zustandsänderung wird durch $s_{t+1} = T(s,a)$ beschrieben. Da es sich bei den betrachteten Umgebungen jeweils um Simulationen einer realen Umgebung handelt, kann T auch als Zustandsübergangsmodell bezeichnet werden. Die auf a folgende Änderung in den Zustand s_{t+1}wird wieder an den Agenten kommuniziert. Auf diese Weise erkundet der Agent den Zustandsraum der Umgebung und lernt dabei, welche Aktionen zulässig sind. Auf unzulässige Aktionen reagiert T mit $s_{t+1} = s_t$, der Zustand bleibt also unverändert. Darüber hinaus besitzt die Umgebung ein Belohnungsmodell $r = R(s,a)$. Die Belohnungswerte r (*reward*) werden ebenfalls an den Agenten kommuniziert. Welche Werte r zum Einsatz kommen, hängt vom jeweiligen Anwendungsproblem ab. Übliche Belohnungswerte sind:

- -∞: Aktion a ist unzulässig
- 0: Aktion a ist zulässig. Der resultierende Zustand s_{t+1} ist noch kein Zielzustand.
- 1: Aktion a ist zulässig. Der resultierende Zustand s_{t+1} ist der gewünschte Zielzustand.
- -1: Aktion a ist zulässig. Der resultierende Zustand s_{t+1} ist aber unerwünscht.

Die erste Exploration endet, wenn der gewünschte Zustand in der Umgebung erreicht ist ($r = 1$). Ein solcher Durchlauf wird als Episode bezeichnet. Alle in einer Episode erreichten Zustände sowie die empfangenen Belohnungen werden im Agenten gespeichert.

Durch wiederholte Explorations-Episoden wird ein immer größerer Teil des Zustandsraums der Umgebung nach dem Trial-and-Error-Prinzip erkundet. Abhängig von der konkreten Anwendung besitzt der Agent nun Wissen über mindestens eine Zustandsfolge, die von s_t nach s_{Ziel} führt. Nun kann ein zweiter Mechanismus zur Wirkung kommen, der als Exploitation bezeichnet wird. Damit ist das Lernen auf Basis bereits erworbenen Wissens gemeint.
Grundsätzlich ist das Verhältnis von angemessener Exploration zu Exploitation in fortgeschrittenen Episoden nicht trivial. Für einen weitergehenden Einstieg in die Problematik sei an dieser Stelle auf [3] und darüber hinaus auf die umfangreiche Primärliteratur verwiesen.

2 Studienbeispiel TvH

Die in [3] vorgestellte Arbeit verfolgt das Ziel, den Einsatz eines RL-Verfahrens im Kontext der Industrierobotik zu untersuchen. Hier werden Gelenkarmroboter häufig für Transport- , Fertigungs- und Montageaufgaben eingesetzt. Gerade vor dem Hintergrund von Montageaufgaben ist die Wahl des Einspieler-Problems Türme von Hanoi (TvH) sehr anschaulich. In seiner minimalen Komplexität mit zwei Scheiben und drei Stäben ist es sogar vollständig grafisch darstellbar, wie Abbildung 2 zeigt.

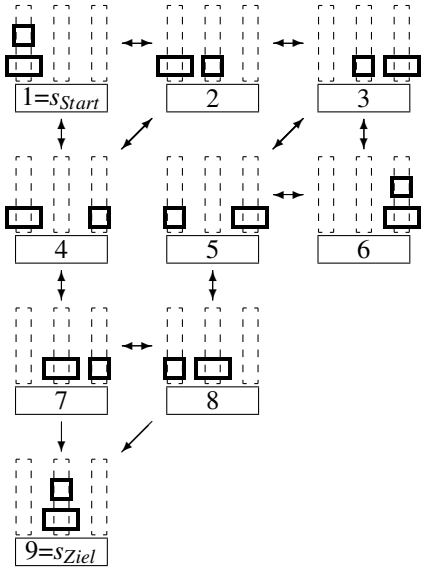


Abbildung 2: Mögliche Spielzüge bzw. Zustände und Übergänge bei TvH minimaler Komplexität nach [3]

Über die Parameter *Anzahl der Scheiben* und *Anzahl der Stäbe* lässt sich eine beliebige Problemkomplexität einstellen. Anhand der Abbildung 2 sind Start, Ziel und Regeln von TvH leicht nachvollziehbar:

1. Größere Scheiben dürfen *nie* über kleineren liegen. Dies gilt auch für den Startzustand.
2. Startzustand: Alle Scheiben liegen auf Stab 1.
3. Zielzustand: Alle Scheiben liegen auf Stab 2.
4. Bei jeder Aktion kann immer nur eine oben liegende Scheibe von einem Stab zu einem anderen Stab transportiert werden (Analogie zu *Pick&Place* in der Robotik).

In seiner minimalen Komplexität besitzt TvH 9 mögliche Zustände und 6 Aktionen.

3 Beschleunigungspotential durch Wechsel der Laufzeitumgebung

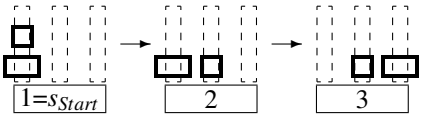
Das Ziel der in [3] publizierten Arbeit bestand im Nachweis einer prinzipiellen Machbarkeit. Der dazu erforderliche Prototyp wurde in einer SCE implementiert und für diesen Beitrag unter folgenden Bedingungen nochmals ausgemessen:

- Hardware-Plattform: Intel Core i7 8th Gen., 6 Cores plus Hyper-Threading Technology (HTT), 32 GB Hauptspeicher
- Betriebssystem: GNU/Linux 4.15.0-43-generic / Ubuntu 18.04 LTS
- SCE: Matlab 2018a
- TvH-Komplexität: 6 Scheiben, 3 Stäbe
- Laufzeit: 6,5 Sekunden im Mittel

Angesichts der noch relativ geringen Problemkomplexität ist eine mittlere Laufzeit von 6,5 Sekunden bereits bedenklich hoch, so dass Überlegungen bezüglich nutzbarer Beschleunigungspotentiale begründet sind. In einem ersten Schritt wurde der SCE-Prototyp ohne struktuelle Änderungen in C reimplementiert. Damit konnte ein Wechsel der Laufzeitumgebung von Matlab 2018a nach GCC 7.3.0 erfolgen. Die anschließend gemessene Laufzeit betrug 2,0 Sekunden, was einem Speedup-Faktor von mehr als 3 entspricht.

4 Beschleunigte Suche durch Binärbäume

Abbildung 3 zeigt die Datenstruktur DS nach zwei ausgeführten Aktionen in der initialen Explorationsepisode.



Datenstruktur DS

S_1	1	2	0	0	0	0
S_2	0	2	0	1	0	0
S_3	0	0	0	1	0	2

Abbildung 3: Speicherung explorierter Zustände im Agenten

Mittels DS werden sämtliche explorierten Zustände der Umgebung im Agenten als jeweils eine Zeile dauerhaft gespeichert. Nach jeder Aktion muss geprüft werden, ob der resultierende Zustand in DS bereits vorhanden ist oder als neue Zeile angefügt werden muss. Der bisherige Prototyp verwendete dafür eine O(n)-Suche. In einer detailierten Laufzeitanalyse wurde festgestellt, dass die O(n)-Suche einen Anteil von 99% an der Gesamtlaufzeit besitzt.

Daraufhin wurde die Suche in DS unter Verwendung eines Binärbaums reimplementiert. Die notwendige Basisfunktionalität steht in der C-Bibliothek der GCC-Umgebung bereits zur Verfügung (tsearch(3)). Die dortige Implementierung basiert auf Donald Knuth [6].

Durch die Umstellung reduziert sich die Komplexität der Suche auf $O(log_2(n))$. Die Laufzeit der bisher untersuchten TvH-Konfiguration sinkt im Mittel von 2 Sekunden auf 0,06 Sekunden, was einem Speedup-Faktor von 33 entspricht.

5 Parallelisierung mittels POSIX-Threads

POSIX-Threads [7] sind eine standardisierte, von fast allen Betriebssystemen unterstützte Technik zur Realisierung mehrerer Kontrollflüsse (Threads) innerhalb eines gemeinsamen Adressraums (Prozess). Demzufolge ist eine sehr effiziente Inter-Thread-Kommunikation nach dem Shared-Memory-Modell (SHM) möglich. Häufig werden Thread-Anwendungen im SPMD-Stil realisiert (*Single Programm Multiple Data*). SPMD-Implementierungen können sowohl sequentiell auf einem Core als auch parallel auf mehreren Cores laufen.

Thread-basierte Programme können auch unter Nutzung nur eines Cores gegenüber einer konventionellen Implementierung einen Speedup aufweisen. Dies wird bei heutigen Prozessoren durch Hyper-Threading und ausgefeilte Cache-Technologien ermöglicht. Bei der bisher untersuchten TvH-Konfiguration ergibt sich dadurch nochmals ein Speedup von 1,2.

Wegen des verwendeten SPMD-Stils kann die Thread-Implementierung ohne Änderungen auch mehrere Cores für PV benutzen. Da die mittlere Laufzeit der untersuchten TvH-Konfiguration aber nur noch 50 Millisekunden beträgt, ist ihr sogenanntes Problemgewicht inzwischen aber so gering, dass sie von PV auf Multi-Core-Technologie nicht mehr profitieren kann.

Wie Tabelle 1 zeigt, können aber komplexere TvH-Konfigurationen durchaus durch PV noch weiter beschleunigt werden.

	Speedup	**Threads**	**Zustände**
8 Scheiben 3 Stäbe	2,3	8	6.564
10 Scheiben 4 Stäbe	2,5	14	1.048.583
10 Scheiben 5 Stäbe	4,3	14	9.765.625

Tabelle 1: Speedup ausgewählter TvH-Konfigurationen

6 Zusammenfassung

In diesem Beitrag wurden verschiedene Möglichkeiten zur Beschleunigung einer ML-Anwendung untersucht. Das größte Einzelpotential wies dabei die Verbesserung der Algorithmik auf. In Kombination tragen jedoch alle Maßnahmen zur Beschleunigung bei. Für die TvH-Konfiguration 6 Scheiben, 3 Stäbe wurde ein Gesamt-Speedup von 130 erreicht. Der PV-Anteil an der Gesamtbeschleunigung wächst mit zunehmender Problemkomplexität.

In weiteren Arbeiten sollen weitere Ansätze zur Beschleunigung von ML-Verfahren wie SIMD-Technologien und NUMA-Architekturen untersucht werden.

Literatur

[1] Freyman B, Pawletta S, Schmidt A, Pawletta T. Design, Simulation and Optimization of Task-Oriented Multi-Robot Applications with MATLAB/Stateflow. *SNE - Simulation News Europe*. 2016;26(2):83–90. doi: 10.11128/sne.26.2.1033.

[2] Abel D, Bollig A. *Rapid Control Prototyping - Methoden und Anwendungen.* Springer Verlag. 2006.

[3] Kunert G, Pawletta T. Generating of Task-Based Controls for Joint-Arm Robots with Simulation-Based Reinforcement Learning. *SNE - Simulation Notes Europe*. 2018;28(4):149–156. 10.11128/sne.28.tn.10442.

[4] Sutter S R, Barton G A. Reinforcement Learning: An Introduction. *Cambridge/MA: MIT Press*. 2012;(2):324.

[5] Akhtar S. Practical Reinforcement Learning. *Birmingham/UK: Packt Publishing Ltd*. 2017;(1):320.

[6] Donald E Knuth. *The Art of Computer Programming Volume 3.* Addison Wesley, 2nd ed. 1998.

[7] Wolf J, Wolf KJ. *Linux-Unix-Programmierung.* Reihnwerk Verlag. 2016.