2004 年度 修士論文

強化学習並列化による
学習速度の向上

提出日：2005 年 2 月 2 日
指導：山名 早人 助教授

早稲田大学大学院理工学研究科情報・ネットワーク専攻
学籍番号：3603U141-8

森 紘一郎

概要

強化学習は知識がない状態からの試行錯誤によって学習を行う。そのため、学習が遅いという欠点があり、学習の高速化が大きな問題点となっている。このような問題に対して、従来、価値関数を分割して各プロセッサに割り当て、並列に更新する手法が提案されている。しかし、強化学習の性質上、分割された価値関数間で頻繁に経験を交換する必要があり、従来の研究ではプロセッサ間通信のオーバーヘッドが大きいことが問題であった。そこで、本論文では、共有する 1 つの価値関数を複数のエージェントが非同期並列的に更新するオーバーヘッドの少ない手法を提案する。本手法は、共有メモリ型並列計算機を対象としており、従来手法に比べて速度向上が高く、実装も容易という利点がある。共有メモリ型並列計算機 IBM pSeries 690 上で 127×127 の迷路タスクを用いて実行したところ 24 プロセッサで 22.2 倍というほぼ線形の速度向上を達成できた。

目次

1 はじめに

強化学習は知識がない状態から試行錯誤によって学習を行うため学習が遅いという欠点がある。この欠点は強化学習を応用する上で大きな問題点であり、さまざまなところで指摘されている。たとえば、G. J. Tesauro の TD-Gammon [17] [18] というバックギャモンを学習するプログラムでは学習に 2 週間かかったという報告がある[*1] [11]。また、R. S. Sutton は、エレベータディスパッチ問題の学習を取り上げて訓練時間の長さが問題であることを指摘している [15]。これらの指摘から分かるように、強化学習を実問題に応用するためには学習の高速化が必須となっている。

強化学習の高速化に関する従来研究は以下の 2 つの立場に分類できる。

- 学習の収束に要する経験数[*2]の減少を目的とした研究。
- 学習の収束に要する計算時間の短縮を目的とした研究。

経験数の減少を目的とした研究には、適正度の履歴 [12]、プラニング [14]、マルチエージェントによる経験の共有 [16] [5]、教示の導入 [6] などがある。一方、計算時間の短縮を目的とした研究には、利得を近似して計算量を減らす Truncated Temporal Difference [2]、木構造を用いる TD(λ) の対数時間更新算法 [3]、並列計算 [9] などがある。

強化学習の高速化といった場合、経験数の減少を目的とした研究が多い。これは、強化学習をロボットなどの実機に応用したとき経験を得るのが難しく、少ない経験数で効率的に学習する方法が模索されてきたからだと考えられる。一方、シミュレーション上の学習ではシミュレーションによって容易に経験を得ることができるため経験数は問題とならず、シミュレーションおよび学習の計算時間が問題となることが多い。強化学習は、シミュレーションに比べ実機で用いられることが多いため経験数を減少させる研究は重要である。しかし、実機に応用するにしても事前にシミュレーション上で学習を行うことが一

[*1] TD-Gammon が作られたのは 1995 年である。現在に比べハードウェアが貧弱であったため計算時間の比較は難しい。

[*2] 経験数の定義は 2 章で詳しく述べる。

般的である [20]。そのためシミュレーションに必要となる計算時間を短縮させる研究も同様に重要であると考える。

本研究では、計算時間の短縮の立場を取り、並列計算機を用いて学習を高速化する手法を提案する。並列計算機を用いて学習を高速化する研究は、ニューラルネットや遺伝的アルゴリズムでも盛んに行われている[*3][21] [4]。その一方、同様の問題を持つ強化学習では並列化の研究はまだ少ないのが現状である。並列化の効果は明らかなのに研究例が少ないという点は [5] でも指摘されている。

強化学習の並列化手法を提案した研究を 2 例あげる。[9] は分散メモリ型並列計算機を用いて価値関数を各プロセッサに分割し、並列に更新する手法を提案している。しかし、強化学習アルゴリズムの性質上、分割されたプロセッサ間で頻繁な経験の交換が必要であり、プロセッサ間通信のオーバーヘッドが大きいことが問題となっている。また、[5]、[1] は、並列計算機を使った実験はしていないものの、分散メモリ型並列計算機を対象とした価値関数の結合手法を提案している。

従来手法の問題点は、プロセッサ間通信のオーバーヘッドである。強化学習の性質を考えると計算データである価値関数を独立に分割して各プロセッサに割り当てることは難しい（6.3 節を参照)。計算データを独立に分割できないためどうしてもプロセッサ間通信が頻繁に生じてしまうのである。そこで、強化学習には価値関数を分割せずに並列計算する手法、すなわち共有メモリ型並列計算機を対象とした手法の方が適していると考えた。

本研究では、従来の分散メモリ型並列計算機を対象とした手法ではなく、共有メモリ型並列計算機を対象とした手法を提案する。共有メモリ上に存在する価値関数を非同期並列的に更新することによって通信オーバーヘッドを減らし、従来手法に比べ速度向上率を飛躍的に高められたことを実験的に示す。

2 章では、強化学習の概要を説明し、どの部分が学習時間におけるボトルネックであるか示す。また、本手法がどの部分を並列化の対象にしているかを述べる。3 章では、強化学習の高速化に関する従来研究をまとめる。特に本論文と関係のある並列化手法について

[*3] ニューラルネット、遺伝的アルゴリズムでも学習時間が問題になっている。アルゴリズムの並列化は高速化手法の一種として研究分野が成立している。

詳細に述べる。4 章では、提案手法である共有メモリ型並列計算機を用いた共有する 1 つの価値関数を複数のエージェントが非同期並列的に更新する手法について詳細に述べる。5 章では、本手法の実験環境、評価タスク（迷路タスク、車の山登りタスク、アクロバットタスク）について説明し、実験結果をまとめる。6 章では、5 章の結果に基づいて考察を行う。7 章では、まとめを行う。

2　強化学習の概要

本章では、強化学習の概要について述べる。特に本研究で用いた Q-Learning のアルゴリズムを詳説する。また、アルゴリズムのどの部分が学習時間におけるボトルネックになっているかを示し、どの部分を並列化の対象にしているか述べる。

2.1　強化学習の枠組み

強化学習とは、環境との相互作用を通して適切な行動戦略を獲得するタイプの機械学習アルゴリズムである。近年、自律エージェント、自律ロボットに関する研究が活発に行われており、強化学習はそれらの学習制御アルゴリズムとして注目されている。自律エージェント、自律ロボットは閉じた環境ではなく、実世界の複雑な環境で動作しなければならない。そのため、環境との相互作用を通して学習する強化学習の枠組みは適している。また、計算機能力の飛躍的向上、理論面での著しい展開も強化学習が活発に研究されるきっかけとなっている。

強化学習の枠組みを図 1 に示す。強化学習では学習する主体をエージェント、エージェント以外を環境と呼ぶ。環境の状態を $s \in \mathcal{S}$（状態の集合）、エージェントの行動を $a \in \mathcal{A}$（行動の集合）とする。状態 s で行動 a を取ると環境の状態は s から $s' \in \mathcal{S}$ に移り、このとき環境からエージェントに報酬 $r \in \mathbb{R}$ が与えられる。報酬はエージェントにさせたいタスクのゴール状態で与えるのが一般的である[*4]。(s, a, s', r) を経験と呼び、エージェントはこの経験を用いて価値関数を更新していく。価値関数の更新がすなわち学習である。エージェントの目的は、最終的に得られる報酬の総和を最大化する行動戦略（方策と呼ばれる）を見つけることである。この行動戦略は価値関数として表され、知覚した状態に対してどの行動を選ぶかを決める方針となる。経験を通して価値関数が適切に更新されたエージェントはゴール状態へ向かう行動を選べるようになる。

[*4] たとえば迷路を通り抜けるタスクではゴール地点で報酬を与える。

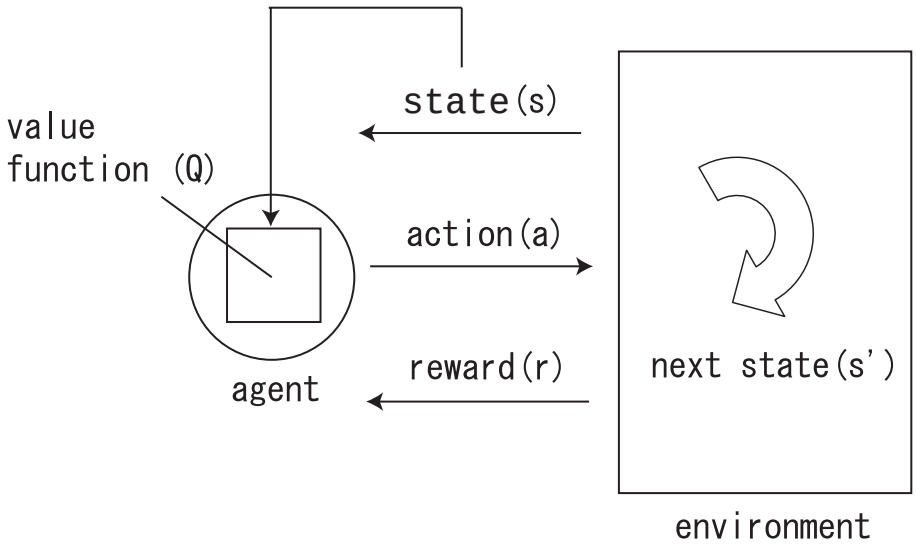

図 1: 強化学習の枠組み

2.2 Q-Learning

強化学習アルゴリズムとして Q-Learning [19] がよく用いられる。Q-Learning は、状態 s、行動 a、次状態 s'、報酬 r から成る経験を用いて価値関数を更新する強化学習アルゴリズムの一種である。

エージェントは価値関数 $Q(s,a)$ を内部に持っている。価値関数 $Q(s,a)$ は、状態 s において行動 a をとったときに得られる報酬の期待値を表している。価値関数が正しく評価されていれば、エージェントは価値の高い状態に遷移するような行動を選択できる。

Q-Learning では、価値関数 $Q(s,a)$ を式 (1) で更新することによって学習が行われる。

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a' \in A} Q(s',a') - Q(s,a)] \tag{1}$$

ここで、α は学習率、γ は割引率と呼ばれるパラメータで、$0 \leq \alpha \leq 1$、$0 \leq \gamma \leq 1$ を満たす実数である。

価値関数 $Q(s,a)$ としてテーブル、線形関数、ニューラルネットが用いられる。テーブル形式は、$Q[s][a]$ という二次元の配列を用意して価値を格納する方式である。実装が容易でよく使われる表現であるが状態・行動数が大きくなるとメモリ使用量・更新に必要

な計算量が指数関数的に大きくなるという欠点がある（次元の呪いと呼ばれる）。この問題に対処するのが価値関数の線形関数・ニューラルネット形式である。線形関数形式は、$Q(s,a)$ を線形関数で表す方式である。状態・行動の汎化が可能であり、テーブル形式とともによく使われる表現である。価値関数の線形関数形式については 3.1.2 項の関数近似で詳細を述べる。ニューラルネット形式は、$Q(s,a)$ をニューラルネットで表す方式である。状態 s と行動 a を入力、$Q(s,a)$ が出力となるニューラルネットを用いる。ニューラルネットは非線形関数であり、強力な汎化能力を持つが実装が難しいという欠点がある。本研究ではテーブル形式と線形関数形式の価値関数を対象にしている。

エージェントは更新された価値関数 $Q(s,a)$ に基づいて行動を選択する。エージェントの行動選択手法には ϵ-greedy がよく使われる。この手法は確率 ϵ でランダムな行動を選択し、確率 $1-\epsilon$ で価値関数に基づいて行動を選択する方法である。ϵ は探査率と呼ばれるパラメータであり、$0 \leq \epsilon \leq 1$ を満たす実数である。エージェントがランダムに行動を選択する可能性を残すことによって局所解に陥らないようにする効果がある。以上が強化学習の大まかな枠組みである。

強化学習が遅いのは、式 (1) の価値関数の更新を学習が収束するまでに大量に繰り返さなければならないからである。強化学習の並列化に関する従来の研究および本研究でも式 (1) の価値関数更新部分を並列化の対象にしている。本研究では、並列計算機を用いて式 (1) の計算を非同期並列的に行うことによって学習を高速化する。

3　強化学習高速化に関する従来研究

本章では強化学習の高速化に関する従来研究について述べる。並列化に限らず、従来の高速化手法全体についてまとめ、本研究の位置づけを明らかにしたい。

強化学習の高速化に関する従来研究は以下の 2 つの立場に分類できる。

- 学習の収束に要する経験数の減少を目的とした研究。
- 学習の収束に要する計算時間の短縮を目的とした研究。

2 章で述べたように Q-Learning における価値関数の更新は経験を用いて式 (1) で行う。価値関数を正しく評価するには環境との相互作用を通して得られた経験（状態 s、行動 a、報酬 r、次状態 s'）が大量に必要となる。

1 つめの立場は学習の収束、つまり価値関数を正しく評価するために必要となる経験の数 [*5]を減らすことである。経験数を減らすことを目的とした研究では、学習に要する計算時間に言及していることはほとんどない。

2 つめの立場は学習が収束するまでに必要となる計算時間を減らすことである。CPU 時間を短くする、計算量を小さくするなどの研究が該当する。

一般的に 1 と 2 は比例する。つまり、学習の収束に必要な経験数を減らせば、計算時間は短くなる場合である。しかし、例外もある。たとえば、3.1.1 項で述べる適正度の履歴は学習に必要な経験数は劇的に減らせるが、計算時間は劇的に増えてしまう。

強化学習の高速化といった場合、経験数の減少を目的とした研究が多い。これは、強化学習をロボットなどの実機に応用したとき経験を得るのが難しく[*6]、少ない経験数で効率的に学習する方法が模索されてきたからだと考えられる。一方、シミュレーション上の学

[*5] 多くの論文では経験数の代わりにエピソード数（試行数）を用いている。エピソードは対象タスクの一回の試行を表しており、多数の経験から構成される。しかし、タスクの中にはエピソードに分割できないものもあるためここではエピソードではなく経験という言葉を用いた。経験数をエピソード数に置き換えても問題はない。

[*6] ロボットなどの実機ではセンサによる状態認識、アクチュエータによる行動を物理的に行わなければならず 1 つの経験を得るのにも時間がかかるためである。

習ではシミュレーションによって経験を容易に得ることができるため、経験数は問題とならず、シミュレーションおよび学習の計算時間が問題となることが多い。

強化学習は、シミュレーションに比べ実機で用いられることが多いため経験数を減少させる研究は重要である。しかし、実機に応用するにしても事前にシミュレーション上で学習を行うことが一般的である [20]。そのためシミュレーションに必要となる計算時間を短縮させる研究も同様に重要であると考える。

以下、経験数の減少に関する研究、計算時間の短縮に関する研究という軸で従来研究を分類し、各手法の概要を説明する。

3.1 経験数減少アプローチ

本節では学習の収束に要する経験数の減少を目的とした研究をまとめる。特に適正度の履歴と関数近似は本研究で用いたアルゴリズムであるため詳細を述べる。

3.1.1 適正度の履歴

適正度の履歴は、エージェントの経験した状態、行動の一時的な記録である。経験した状態、行動には適正度が与えられる。そして、試行が終了すると適正度を持った状態が一斉に更新される。適正度の履歴と Q-Learning を組み合わせたアルゴリズムは Q(λ) と呼ばれる [19] [8]。

適正度は各状態行動対に存在し、時刻 t における状態 s、行動 a の適正度は $e_t(s,a) \in \mathbb{R}$ で表される。各ステップ t において訪問された状態行動対の適正度は 1 に設定される。またすべての状態行動対で 1 ステップおきに適正度は $\gamma\lambda$ だけ減衰される。この減衰は最近訪問した状態・行動ほど報酬への関与が大きいという理由による。

$$e_{t+1}(s,a) = \begin{cases} 1 & s = s_t, a = a_t \\ \gamma\lambda e_t(s,a) & \text{それ以外の s,a} \end{cases} \tag{2}$$

ここで γ は 2.2 節で述べた割引率である。λ は、履歴減衰率と呼ばれ $0 \leq \lambda \leq 1$ を満たす実数である。

適正度の履歴は、各状態がどれだけ頻繁に訪問されたかを表している。そして、適正度

が高い状態ほど報酬への関与が大きいと判断され大幅に更新される。価値関数の更新規則は式 (3) のようになる。

$$\begin{aligned} \delta &= r + \gamma Q(s', a') - Q(s, a) \\ \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \qquad Q(s, a) &\leftarrow Q(s, a) + \alpha \delta e(s, a) \end{aligned} \tag{3}$$

Q-Learning と異なるのは適正度を持つすべての状態が一斉に更新される点である。1 回の試行で多くの状態を更新できるため学習が高速化される。

ただし、適正度の履歴の高速化は経験数に限る。適正度の履歴は学習に必要な経験数は劇的に減らせるが、計算時間は劇的に増えてしまうという欠点がある。Q(λ) の学習時間を短縮する手法として提案されたのが 3.2.1 節で述べる TTD （Truncated Temporal Difference）[2] である。

3.1.2 関数近似

状態空間が大きい、つまり、テーブルで表された価値関数 $Q[s][a]$ が大きい場合の対処法として関数近似がある [15]。関数近似とは一般化を行う手法である。関数近似は、すでに得られている限られた経験を一般化して、まだ経験していない状態、行動の価値を推定する。関数近似を使うと経験を一般化できるためエージェントは巨大な状態行動空間におけるすべての状態を更新する必要がなくなる。

関数近似を用いると価値関数 $Q(s, a)$ は、$\vec{\theta}$ をパラメータベクトルとするパラメータ関数の形で表される。価値関数 $Q(s, a)$ は、パラメータ $\vec{\theta}$ に依存し、$\vec{\theta}$ の変化によって更新される。一般にパラメータベクトル $\vec{\theta}$ の要素数は状態数に比べてはるかに小さく、パラメータを 1 つ変更するだけで多くの状態が一度に更新される。

各状態 s は、$\vec{\theta}$ の要素数と同じ要素数を持つ特徴ベクトル $\vec{\phi}$ で表される。状態を特徴ベクトルに変換する方法としてタイルコーディング [15] がよく用いられている。

価値関数 $Q(s, a)$ は、パラメータベクトル $\vec{\theta}$ と特徴ベクトル $\vec{\phi}$ を用いて式 (4) の線形関数で表される。

$$Q(s, a) = \vec{\theta}^T \vec{\phi} = \sum \theta(i)\phi(i) \tag{4}$$

そして、線形関数形式の Q-Learning では、$Q(s,a)$ を直接更新するのではなく、式 (5) でパラメータベクトル $\vec{\theta}$ を更新する。

$$\begin{aligned} \delta &= r + \gamma Q(s', a') - Q(s, a) \\ \vec{\theta} &\leftarrow \vec{\theta}\delta\vec{e} \end{aligned} \tag{5}$$

ここで、$\vec{e}$ は適正度を表している。線形関数形式の場合は適正度の履歴と組み合わせるのが一般的である。

3.1.3 その他の手法

プラニング [13] は、エージェントの経験から環境のモデル（経験の集合）を作り、モデルを利用して価値関数を更新する手法である。モデルを使って更新を行うため少ない経験で学習できる。プラニングを使うと実際に環境との相互作用から得た経験とモデルから得たシミュレーション上の経験の 2 つの方向から価値関数を更新できるため学習が高速化される。また [13] では環境との相互作用とプラニングが並列計算機で並列化可能なことが示唆されている。本論文では取り上げなかったが、計算時間短縮の手法として利用できると考える。

k-確実探査法 [7] は環境同定手法の一種である。Q-Learning で膨大な経験が必要な原因として探査をランダムに行っている点が挙げられる。k-確実探査法における探査は Q-Learning のようにランダムではなく、まだ十分に試されていない行動をフラグを使って優先して選択する。これにより価値関数を効率的に更新でき、学習が高速化される。

学習の高速化にマルチエージェントを用いる手法もある。マルチエージェント強化学習の目的は複数のエージェントが協調して目標を達成するための方法を学習させることにあるが、Tan [16] は複数のエージェントが情報を共有することによって学習が高速化できることを指摘している。[16] では、知覚、経験、価値関数の交換による学習効率の変化が考察されている。

教示は、エージェントに知識を与えることによって学習を高速化させる手法である。教示の与え方としてさまざまな手法が考案されている。[6] は関数近似のニューラルネットに教師信号を与える手法を提案している。[10] は、強化学習アルゴリズムの一種

Actor-Critic に教師あり学習を導入している。

3.2 計算時間短縮アプローチ

強化学習の計算時間を減らす研究には、アルゴリズムを改良し時間計算量を抑える手法と並列化する手法がある。時間計算量を抑える手法として、Truncated Temporal Difference（TTD）[2]、TTD を改良した TD(λ) の対数時間更新算法 [3] がある。また強化学習の並列化手法として、[9]、[5]、[1] など分散メモリ型並列計算機を対象とした手法が提案されている。しかし、[5] でも指摘されているように強化学習の並列化に関する研究は少ないのが現状である。

3.2.1 Truncated Temporal Difference（TTD）

強化学習の時間計算量を減らす試みとして Truncated Temporal Difference [2] と TD(λ) の対数時間更新算法 [3] が提案されている。TTD は収益を m ステップで打ち切り、m ステップ収益を効率的に計算することによって時間計算量を減らす手法である。従来の TD(λ) の時間計算量は状態数 $|\mathcal{S}|$ に比例した $O(|\mathcal{S}|)$ に対し、TTD は $O(\log m|\mathcal{S}|)$ である。

TD(λ) の対数時間更新算法は、TTD を改良した手法である。状態を木構造で表し、更新を木の節点で行うことによって時間計算量を $O(\log |\mathcal{S}|)$ に抑えている。

3.2.2 価値関数の分割更新手法

[9] は、テーブル型価値関数を分割して各プロセッサに割り当て、並列に更新することによって学習を高速化している。いわゆるデータ分割と呼ばれる並列化手法である。

[9] では図 2 のようにマスター・スレーブ型にプロセッサを配置している。

マスターは、

- 価値関数（Q テーブル）を分割し、各スレーブに割り当てる。

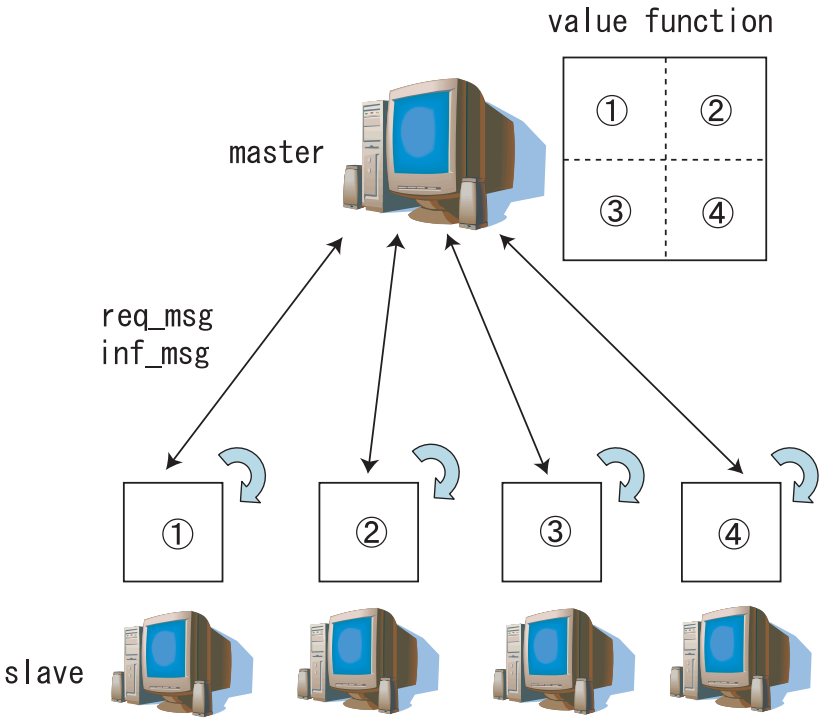

図 2: 価値関数の分割更新手法

- 各スレーブの要求に応じて価値関数の値を通知する。
- 価値関数の更新を管理する。

ことに責任を負う。一方、各スレーブは、マスターから割り当てられた範囲の価値関数を並列に更新する。マスターとスレーブ間は req_msg と inf_msg というメッセージを使って通信する。req_msg はあるスレーブが他のスレーブが担当している領域の価値関数の値を知りたいときに送られる。マスターは要求された価値関数の値をスレーブに返す。このとき、マスターで管理している価値関数の値が最新のものとは限らない。マスターが管理する価値関数の値はスレーブから inf_msg が来たときに最新の値に更新されるからである。inf_msg は価値関数に変更があったとき送られ、スレーブで更新した価値関数の値をマスターに返す。この方法では、各スレーブに割り当てられた範囲以外の値が必要なときにマスタースレーブ間でプロセッサ間通信が生じる。このプロセッサ間通信は頻繁に生じるため [9] ではマスターから得た値を予め決めたサイズ分（4 または 8）だけ保存することによって通信頻度を抑える工夫をしている。

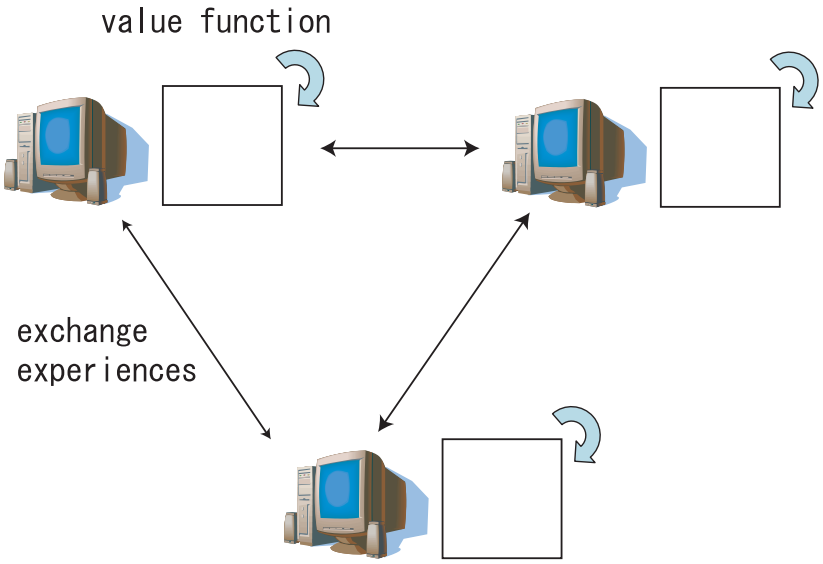

図 3: 価値関数の結合手法

[9] は、分散メモリ型並列計算機 Power Mouse Parallel Machine 上で PVM（Parallel Virtual Machine）メッセージ通信ライブラリを使って実装している。14×14 の迷路タスク[*7]に適用し、4 プロセッサで 2.32 倍の速度向上を達成している。しかし、8 プロセッサでは通信によるオーバーヘッドのため逆に速度は低下してしまっている。

3.2.3 価値関数の結合手法

[5] は、複数のエージェントが独立した価値関数を並列に更新し、学習の途中で得られた価値関数を互いに送信しあい、結合させることによって学習を高速化する手法を提案している（図 3）。

各エージェントは独自の価値関数 $Q(a)$[*8]を持っており、価値関数は並列に更新される。このとき、価値関数が更新された回数を $k(a)$ に保存しておく。ある程度学習が進んだと

[*7] 使われたタスクの規模が小さすぎると誤差の影響が大きくなってしまう。この点は問題があると考える。

[*8] [5] は状態を持たないタスクで実験したため $Q(a)$ と表記している。状態を持ったタスクへの拡張は [1] が行っている。

ころで各々の $Q(a)$ を式 (6) で結合する。

$$Q(a) = \frac{\tilde{Q}(a) * \tilde{k}(a) + \hat{Q}(a) * \hat{k}(a)}{\tilde{k}(a) + \hat{k}(a)} \tag{6}$$

ここで、$\tilde{Q}(a), \tilde{k}(a)$ は自分の価値関数、更新回数であり、$\hat{Q}(a), \hat{k}(a)$ は自分以外のエージェントの価値関数、更新回数である。自分以外のエージェントの価値関数は、式 (6) と同様に更新回数で価値関数を重み付け平均して求めている。更新回数が大きいほどその価値はより正確であるため k による重み付けが必要である。[5] は、この手法を評価し、学習に必要な経験数が減少することを実験的に示している。

[1] は、タスクを複数のエージェントで解くことの利点を明らかにし、価値関数の結合頻度を変えることによって学習に必要となる経験数がどのように変化するかを詳細に調べている。

[1] は、1 個のエージェントが価値関数を 1000 回更新するよりも 10 個のエージェントが 100 回ずつ更新し、学習の途中で結果を式 (6) で結合した方が学習に必要となる経験数が少ないことを実験的に示している。

また、価値関数の結合頻度として

- エピソードの各ステップごとに結合する。
- エピソードごとに結合する。
- タスクに応じて 5, 10, 15 エピソードごとに結合する。
- 価値関数をはじめから共有する。

の 4 パターンをあげて比較している。タスクに応じて結果は変わるものの各ステップごとに結合する手法と価値関数をはじめから共有する手法が学習の高速化に最も有効であると結論付けている。

[5][1] では、学習に要する経験数の減少に焦点を当てており、学習に必要となる計算時間については述べられていない。ただし、[1] はこれらの手法が並列計算機へ実装する点で適していると指摘している。本論文では、[5] の手法を実際に並列計算機で評価して考察も行った。詳細は 5.3.8 項、6.3.3 項で述べる。

3.3 まとめ

強化学習高速化の従来研究をまとめると表 1、表 2 のようになる。

表 1: 経験数減少アプローチ

手法	説明
適正度の履歴 [12]	経験した状態・行動に適正度を割り当て一斉に更新する。収束に必要な経験数は大幅に減らせるが計算量が大きい。
関数近似 [15]	経験した状態・行動を一般化する。収束が保証できないという欠点がある。
プラニング [14]	環境のモデルを作成し、モデル上の経験を用いる。モデルの経験を用いるため環境の変動に弱い。
k-確実探査法 [7]	経験していない状態・行動を優先的に探索する。行動数が大きくなると対応できない。
マルチエージェント [16]	複数のエージェントが経験を共有する。
教示 [6]	エージェントに対して外部の観察者が教示を与える。エージェントの自律性が失われる。

表 2: 計算時間短縮アプローチ

手法	説明
TTD [2]	収益の計算を途中で打ち切ることによって計算量を減らす。
対数時間更新法 [3]	状態を木構造で表し更新の計算量を減らす。価値関数がテーブル形式であることを前提としているため状態数が増えると対応できない。
価値関数の分割更新 [9]	価値関数を分割して各プロセッサに割り当て並列に更新する。プロセッサ間通信のオーバーヘッドが大きいことが問題点である。
価値関数の結合手法 [5]	複数のエージェントが価値関数を並列に更新し、途中で経験を結合する。通信時間が大きな問題点と考えられる。

本論文と関係のある並列化手法の問題点は、

- 価値関数の分割を明示的に行う必要がある。
- プロセッサ間通信によるオーバーヘッドが大きい。
- 価値関数の結合コストが大きい。

などが考えられる。本研究では、これらの問題点を解決する手法を提案する。

4　提案手法

本論文では、共有する一つの価値関数を複数のエージェントが非同期並列的に更新する手法を提案する。なお、プラットフォームとしては共有メモリ型並列計算機を想定する。

複数のエージェントを用いる点は、[5] と同じである。異なるのは、[5] では各エージェントが価値関数を独立に持っていたのに対し、提案手法ではすべてのエージェントが価値関数を共有している点である。

つまり、エージェントは異なるプロセッサ上で並列に動作するが、更新する価値関数は共有メモリ上にあり、すべてのエージェントがアクセスできる。また、同期処理による遅延を避けるため、すべてのエージェントは共有された価値関数を非同期並列的に更新する。つまり、あるエージェントが価値関数を更新している間でも他のエージェントが価値

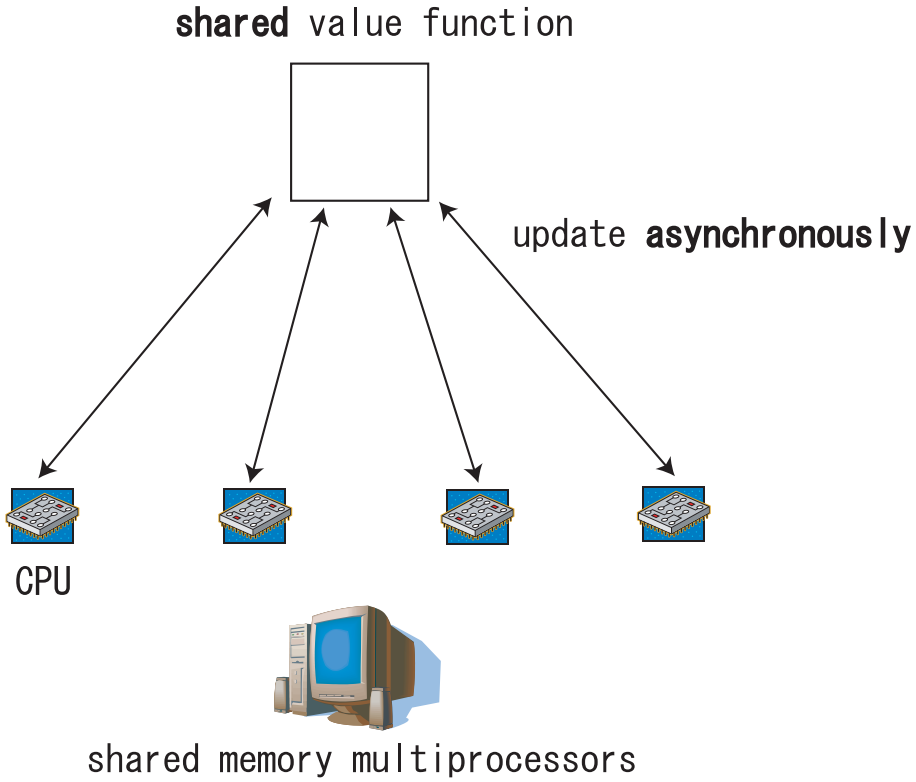

図 4: 共有する一つの価値関数を非同期並列的に更新する手法

```
Q(s, a)        // global (shared) variable

// main method
main() {
    for (i=0; i < NUM_AGENTS; i++)
        Create threads that execute run()
}

// Q-learning
run() {
    Initialize Q(s, a) arbitrarily
    Repeat (for each episode):
        Initialize s
        Repeat (for each step of episode):
            Choose a from s using policy derived from Q(s, a) (e.g., ε-greedy)
            Take action a, observe r, s'
            Q(s, a) ← Q(s, a) + α[r + γ max_a' Q(s', a') − Q(s, a)]
            s ← s'
        until s is terminal
}
```

図 5: 提案手法の擬似コード（Q-Learning の部分は [15] から引用）

関数にアクセスできるようにする。頻繁にメモリへアクセスする必要がある強化学習の性質を考えるとこの手法が一番効率がよいと考える。図 5 に擬似コードを示す。

提案手法の特長をまとめると以下のようになる。

- 価値関数の分割を明示的に行う必要がないため並列化の実装が容易である。
  データ分割を行う [9] の手法はプロセッサ数に応じて価値関数をスレーブに分割し

なくてはならない。価値関数の種類に応じて価値関数を適切に分割するのは手間がかかる。一方、提案手法では全エージェントが価値関数を共有しているためプロセッサ数に応じて価値関数を分割する必要はない。また、プロセッサ数に応じてスレッドを起動するだけで並列化が可能であり実装が容易である。

- 線形関数形式にも対応している。

  従来手法では、価値関数がテーブル形式の場合のみを対象としていた。本来、並列化しなくてはならないようなタスクの価値関数はテーブルで表すことはできず、線形関数やニューラルネットワークが使われる。提案手法は同じ並列化手法でテーブル、線形関数の両形式に適用できるため大規模なタスクにも対応できる。

- オーバーヘッドが小さく、速度向上率が高い。

  従来手法は、プロセッサ間通信に要するオーバーヘッドが大きい点が問題であった。提案手法は、プロセッサ間通信が生じないのでオーバーヘッドがなく、速度向上率は一番高い。

5 評価実験

提案手法の有効性を確かめるため 2 種類の共有メモリ型並列計算機を用いて評価実験を行った。本章では実験環境、評価タスク、評価結果について述べる。

5.1 実験環境

使用した共有メモリ型並列計算機は SUN Ultra 80 と IBM pSeries 690 である。特に断りのない場合は Sun Ultra 80 を用いている。5.3.7 項では IBM pSeries 690、5.3.8 項では分散メモリ型並列計算機（PC クラスタ）を用いている。使用した並列計算機の仕様を表 3 に示す。

表 3: 使用した並列計算機の仕様

機種名	Sun Ultra 80	IBM pSeries 690	PC クラスタの各 PE
CPU	Ultra SPARC II × 4	Power4 × 24	Pentium4
クロック	450MHz	1.1GHz	2.4GHz
L1 キャッシュ	16KB	64KB	8KB
L2 キャッシュ	4MB	1.4MB	512KB
L3 キャッシュ	-	32MB	-
メインメモリ	1GB	8GB	1GB
OS	SunOS 5.8	AIX 5	RedHat Linux 7.3

プログラムの実装には、C 言語を用いた。並列化には共有メモリ型並列計算機では POSIX pthread、分散メモリ型並列計算機では MPI を用いた。コンパイラは gcc を用いた。コンパイラオプションは指定していない。スレッドのスケジューリングポリシーにはラウンドロビンを採用し、各スレッドの実行機会が均等になるようにした。

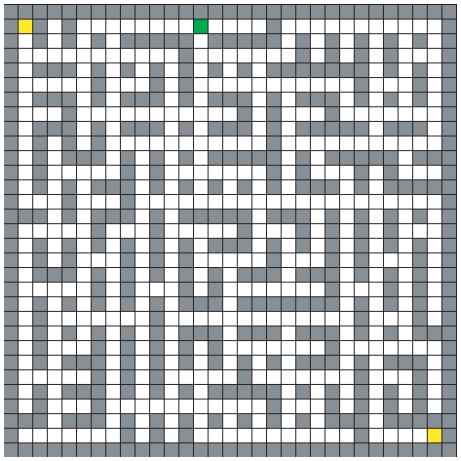

図 6: 迷路タスク

5.2 評価タスク

提案手法の有効性を確かめるため迷路タスク、車の山登りタスク、アクロバットタスクの 3 つを評価タスクに選び、実験を行った。これらのタスクは強化学習の評価タスクとして一般的に使われているものである。

5.2.1 迷路タスク

迷路タスクにおける環境は図 6 に示す迷路である[*9]。迷路の大きさは、63×63 と 127×127 の 2 種類を用意した。左上隅がスタート地点、右下隅がゴール地点である。エージェントの目的はスタート地点からゴール地点への最短経路を学習することである。

状態は、エージェントの迷路中の位置である。各迷路にはそれぞれ 3969（63×63）、16129（127×127）の状態が存在する[*10]。

各状態でエージェントが選択できる行動は、UP（上へ 1 マス進む）、DOWN（下へ 1 マス進む）、LEFT（左へ 1 マス進む）、RIGHT（右へ 1 マス進む）の 4 種類である。エー

[*9] 迷路はランダム迷路生成アルゴリズムの一種である棒倒し法を用いて作成した

[*10] エージェントは壁を通ることができないため探索すべき実際の状態数はこれらの値より少ない

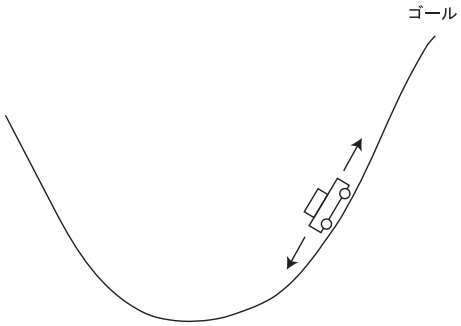

図 7: 車の山登りタスク（[15] から改変）

ジェントは壁には進むことができない。壁に進む行動を選択したとき状態は変わらない（その場にとどまる）ことにする。

報酬はエージェントがゴール状態にいるとき 0、それ以外の状態では-1 とした。

使用した強化学習アルゴリズムは Q-Learning であり、価値関数にはテーブル形式を用いた。価値関数は式 (1) で更新される。強化学習のパラメータは、学習率 α を 0.1、探査率 ϵ を 0、割引率 γ を 0.9 とした。これらのパラメータは収束性と収束後の安定性で評価し、最も適切なものを選んだ。

5.2.2 車の山登りタスク

車の山登りタスクの環境は図 7 に示す凹型の山道である。エージェントは車であり、山道の右端最上部にたどりつくのが目的である。

状態は車の位置 x_t と速度 v_t である。初期状態は $x_0 = -0.5$、$v_0 = 0$ とした。迷路タスクとは異なり、状態は連続値をとる。

エージェントの選択できる行動は、フルスロットル前進（右へ進む)、フルスロットル後退（左へ進む)、ゼロスロットル（惰性で進む）の 3 種類である。エージェントの動きは単純化した物理学モデルにしたがい、エージェントの速度と位置は式 (7) によって更新される。

$$\begin{aligned} v_{t+1} &= bound(v_t + 0.001a_t - 0.0025\cos 3x_t) \\ x_{t+1} &= bound(x_t + v_{t+1}) \end{aligned} \tag{7}$$

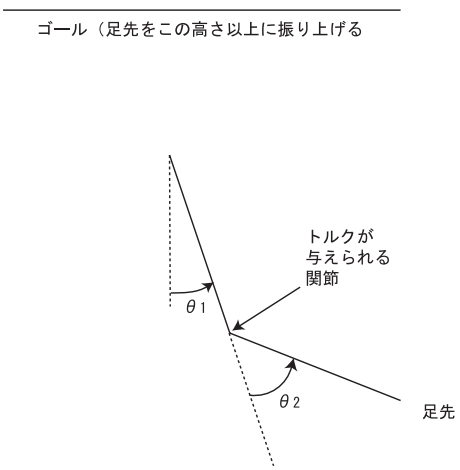

図 8: アクロバットタスク（[15] から改変）

ここで、a_t は行動を表しており、フルスロットル前進のとき 1、フルスロットル後退のとき-1、ゼロスロットルのとき 0 とする。$bound()$ は、位置、速度の範囲をそれぞれ $-1.2 \leq x_{t+1} \leq 0.5$、$-0.07 \leq v_{t+1} \leq 0.07$ に制限する関数である。

報酬はエージェントがゴール状態にいるとき 0、それ以外の状態では-1 とした。

使用した強化学習アルゴリズムは Q(λ) であり、価値関数には線形関数形式を用いた。価値関数は式 (5) で更新される。強化学習のパラメータは、学習率 α を 0.1、探査率 ϵ を 0.1、割引率 γ を 1、履歴減衰率 λ を 0.9 とした。これらのパラメータは収束性と収束後の安定性で評価し、最も適切なものを選んだ。

5.2.3 アクロバットタスク

アクロバットは、図 8 に示す鉄棒で回転を行う体操選手に似たアクチュエータつき 2 リンクロボットである。第 1 関節（鉄棒をつかむ体操選手の手に相当）に力を加えることはできないが、第 2 関節（体操選手の腰の曲げに相当）は可能である。アクロバットタスクの目的は、振り上げ動作によって、足先を第 1 関節を超えてリンク 1 つ分の高さまで最小時間で振り上げることである。

状態は、関節角度 θ_1、θ_2 と関節の各速度 $\dot{\theta_1} \in [-4\pi, 4\pi]$、$\dot{\theta_2} \in [-9\pi, 9\pi]$、計 4 つの連

続値で表される。

エージェントの選択できる行動は、正のトルク（1.0)、同じ大きさの負のトルク（-1.0)、およびゼロトルク（0）の 3 種類である。アクロバットの角度と角速度は式 (8) の運動方程式によって更新される。

$$
\begin{aligned}
\ddot{\theta_1} &= -d_1^{-1}(d_2\ddot{\theta_2} + \phi_1) \\
\ddot{\theta_2} &= \left(m_2 l_{c2}^2 + I_2 - \frac{d_2^2}{d_1}\right)^{-1} \left(\tau + \frac{d_2}{d_1}\phi_1 - m_2 l_1 l_{c2} \dot{\theta_1}^2 \sin\theta_2 - \phi_2\right) \\
d_1 &= m_1 l_{c1}^2 + m_2(l_1^2 + l_{c2}^2 + 2 l_1 l_{c2} \cos\theta_2) + I_1 + I_2 \\
d_2 &= m_2(l_{c2}^2 + l_1 l_{c2} \cos\theta_2) + I_2 \\
\phi_1 &= -m_2 l_1 l_{c2} \dot{\theta_2}^2 \sin\theta_2 - 2 m_2 l_1 l_{c2} \dot{\theta_2}\dot{\theta_1} \sin\theta_2 \\
&\quad + (m_1 l_{c1} + m_2 l_1) g \cos(\theta_1 - \pi/2) + \theta_2 \\
\phi_2 &= m_2 l_{c2} g \cos(\theta_1 + \theta_2 - \pi/2)
\end{aligned}
\tag{8}
$$

ここで、τ はトルク、$m_1 = m_2 = 1$（リンク質量)、$l_1 = l_2 = 1$（リンク長)、$l_{c1} = l_{c2} = 0.5$（リンクの重心までの距離)、$I_1 = I_2 = 1$（リンクの慣性モーメント)、$g = 9.8$（重力加速度）である。

報酬は、ゴール状態で 0、それ以外の状態では-1 とした。

使用した強化学習アルゴリズムは Q(λ) であり、価値関数には線形関数形式を用いた。価値関数は式 (5) で更新される。強化学習のパラメータは、学習率 α を 0.0001、探査率 ϵ を 0.001、割引率 γ を 1、履歴減衰率 λ を 1 とした。これらのパラメータは収束性と収束後の安定性で評価し、最も適切なものを選んだ。アクロバットタスクではエージェントの行動が結果に与える影響が大きいため学習率と探査率を低めに設定する方がより適切である。

5.3 結果

本節では、提案手法を評価した結果とその意味について述べる。これらの結果からわかる考察は 6 章でまとめて扱う。

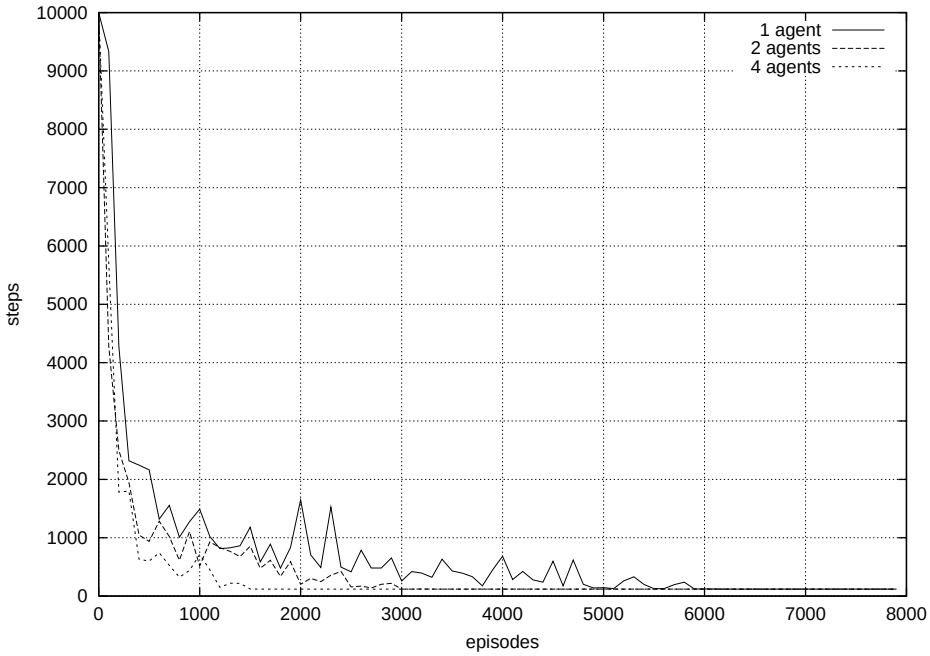

図 9: 迷路タスク（63×63）の学習曲線

5.3.1 学習曲線

図 9 は迷路タスク（63×63)、図 10 は迷路タスク（127×127）の学習曲線であり、並列に動作するエージェント数と学習速度の関係を表している。横軸はエピソード数（スタート地点からゴール地点へたどりつくまでが 1 エピソード)、縦軸はステップ数（エージェントの 1 回の行動が 1 ステップ）である。たとえば、図 9 では、学習初期（1 エピソード）では迷路を抜けるのに 10000 ステップ以上かかっている。しかし、学習が進むにつれて迷路を抜けるためのステップ数が強化学習によって徐々に小さくなっていくことがわかる。学習が収束している点でのステップ数は図 9 では 120、図 10 では 264 であり、これらが迷路を抜けるのに必要な最短ステップ数である。

提案手法は、全エージェント間で価値関数を共有しているためどのエージェントの学習曲線をとってもほぼ一致する。そのため複数のエージェントを用いた場合は、エージェント 1（はじめに起動したエージェント）の学習曲線を記録してグラフにした。

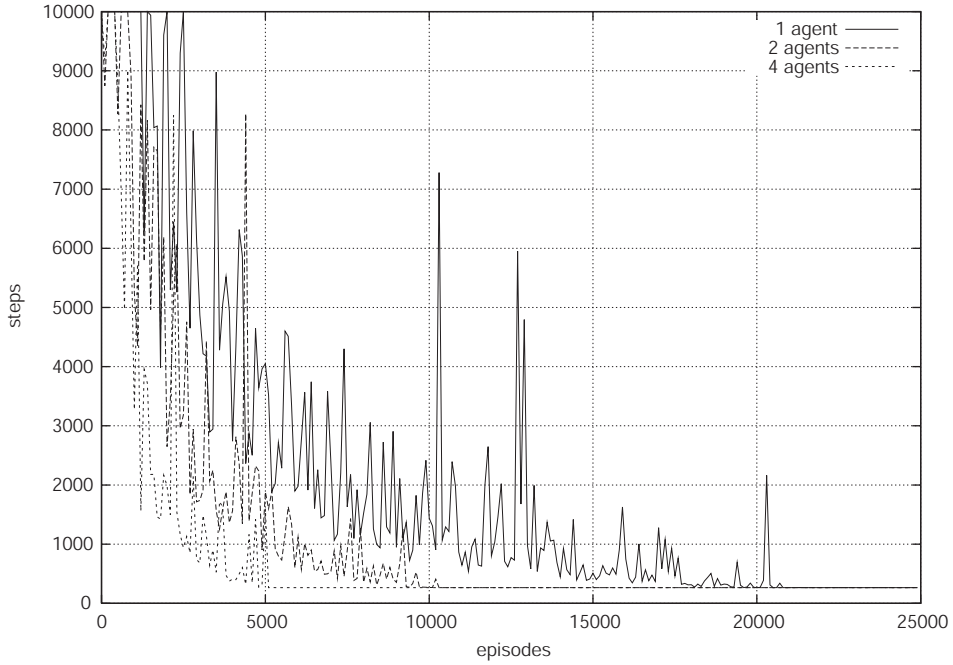

図 10: 迷路タスク（127 × 127）の学習曲線

図 9 と、図 10 からエージェント数と収束エピソード数の関係をまとめると表 4 のようになる。

表 4: エージェント数と収束エピソード数の関係

エージェント数	1	2	3	4
収束エピソード数（63 × 63）	6300	3300	2100	1600
収束エピソード数（127 × 127）	20800	10400	7100	5300

表 4 から価値関数の更新に参加するエージェント数が増えるにしたがい、学習の収束エピソードは早まることがわかる。迷路タスク（127 × 127）においてエージェント数が 1 の場合、20800 エピソードで収束するのに対し、エージェント数が 4 の場合、5300 エピソードで収束しており、ほぼ 4 分の 1 のエピソード数で学習が収束する。

表 4 は、エージェント 1 のみのエピソード数であることに注意しなければならない。実際に必要となるエピソード数は 2 エージェント動かしていた場合は 2 倍、4 エージェント動かしていた場合は 4 倍となる。学習の収束までに必要となる「全」エピソード数は 1

エージェントのときに比べて減ってはいない。つまり、1 章で述べた「学習の収束に要する経験数[*11]の減少」は達成されていない。

本研究の目的は学習の収束に要する経験数を減らすことではなく、学習に要する計算時間を短縮させることである。そのため、エピソード数が増えてしまっても計算時間が短くなればよいとの立場に立っている。

5.3.2 学習時間

本項では学習が収束するまでの計算時間（以下、学習時間）について述べる。学習時間はプログラムの起動時から図 9、図 10 の収束点までの時間を測定した。

表 5、表 6 は、迷路タスクでエージェント数を 1、2、3、4、8 としたときの収束までの時間を測定した結果である。計算機の負荷は一様でないため 10 回の試行でもっとも速かった結果を記録した。

表 5: 迷路タスク（63 × 63）における学習時間

エージェント数	1	2	3	4	8
学習時間（秒）	8.99	5.10	3.65	2.83	2.91
速度向上率	1.00	1.76	2.46	3.18	3.09

表 6: 迷路タスク（127 × 127）における学習時間

エージェント数	1	2	3	4	8
学習時間（秒）	82.82	45.02	31.78	24.65	24.82
速度向上率	1.00	1.84	2.61	3.36	3.34

表 5 と表 6 から価値関数の更新に参加するエージェントが増えるにつれ、学習時間が短縮されることがわかる。エージェント数が 4 のときに比べ、8 のときの方が遅いのは使用マシンのプロセッサ数が 4 台しかないためである。各エージェントはスレッドとして実装

[*11] 経験数は図 9、図 10 において横軸、縦軸、関数値で囲まれた面積で表される

```
// mutex for synchronization
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
// lock mutex
pthread_mutex_lock(&mutex);
// update value function
Q(s,a) ← Q(s,a) + α[r + γ max_a' Q(s',a') − Q(s,a)]
// unlock mutex
pthread_mutex_unlock(&mutex);
```

図 11: Mutex を用いた同期

されており、4 プロセッサでは 4 つのスレッドまでしか並列に動かすことはできない。

また表 5 と表 6 を比べると規模の大きい迷路の方が速度向上率が高いことがわかる。これは、各エージェントが更新する範囲がばらつき、メモリアクセス競合が起きにくくなるためである。メモリアクセス競合については 6.1 節で考察する。

5.3.3 同期更新の学習時間

価値関数の更新時に同期処理を入れて学習時間を測定し、非同期時の結果と比較した。同期処理には pthread_mutex_lock() を図 11 のように用いた。

価値関数 Q[s][a] を更新する前に mutex をロックし、価値関数の更新が多数のスレッドによって同時に行われることを防いでいる。

表 7 に非同期時と同期時の学習時間を測定した結果を示す。127×127 の迷路タスクを使用した。

この結果から更新時に同期を取るよりも取らない場合の方が学習時間が短縮されることがわかる。同期更新の方が非同期更新に比べて遅いのは mutex をロック、アンロックする際のオーバーヘッドおよび複数のスレッドが同時に書き込みを行う場合の同期待ちが原因である。強化学習は価値関数へ頻繁にアクセスするため同期オーバーヘッドは無視でき

表 7: 非同期更新時と同期更新時の学習時間

エージェント数	1	2	4	8
非同期更新時（秒）	83.82	45.02	24.65	24.82
同期更新時（秒）	96.11	71.10	69.83	122.71

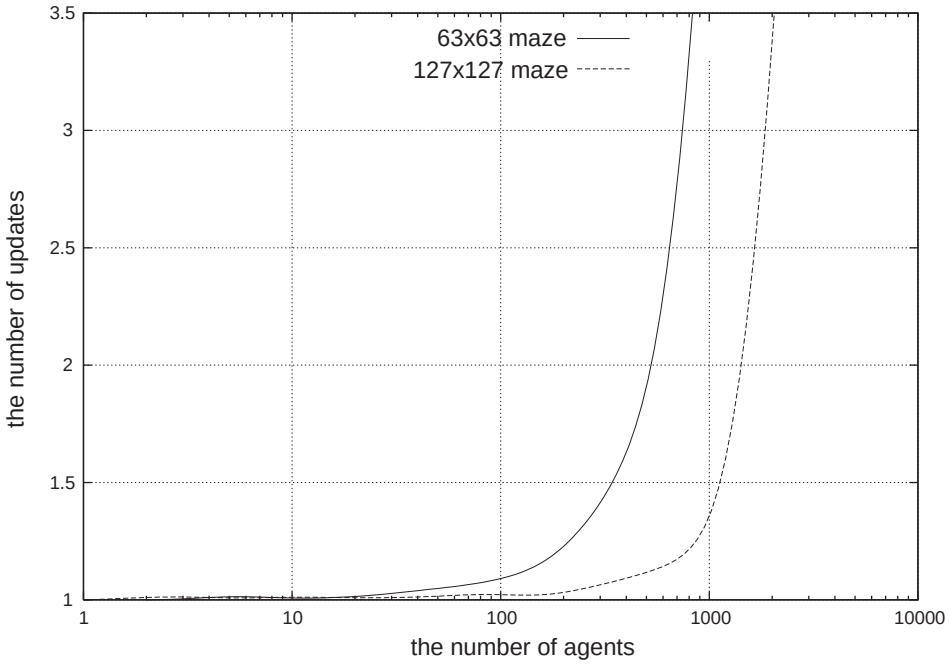

図 12: 収束までの総更新回数

ないほど大きくなってしまう。非同期更新と同期更新については 6.1 節で考察する。

5.3.4 収束までの総更新回数

図 12 は学習の収束までに価値関数を何回更新する必要があるかを表したグラフである。127×127 の迷路タスクを使用した。横軸は更新したエージェント数、縦軸はエージェント数 1 を基準にした総更新回数の比率である。エージェント数 1 のときの総更新回数は約 570 万回である。

図 12 から更新を行うエージェント数の増加にしたがって総更新回数が対数グラフで指数関数的に増加することがわかる。また、大きい迷路（127×127）と小さい迷路（63×63）

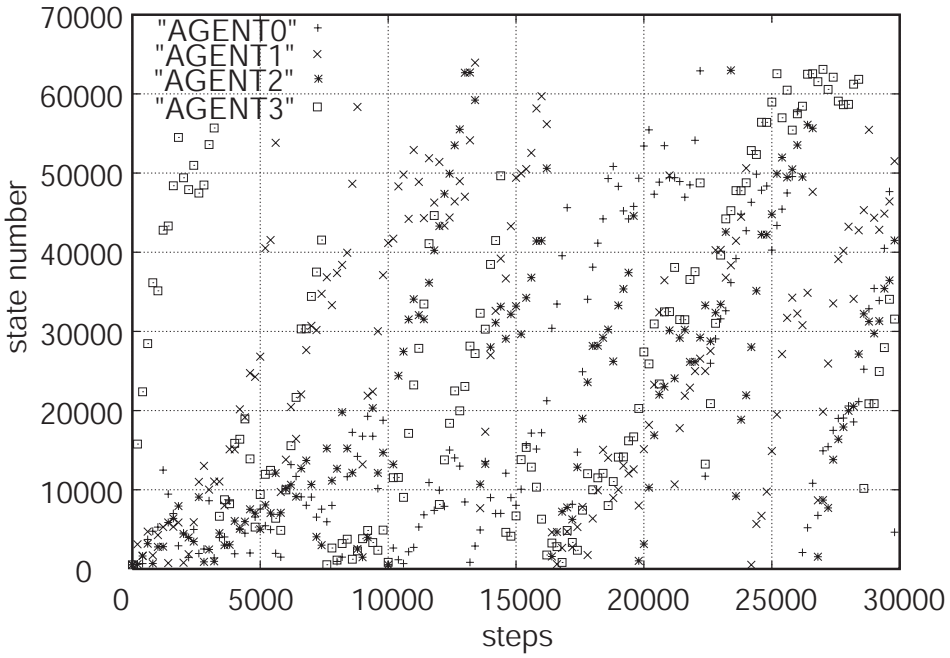

図 13: 各エージェントの価値関数更新分布

を比較すると小さい迷路の方がエージェント数の増加にしたがい急激に総更新回数が増えていることがわかる。

この理由は非同期更新において無駄な更新が生じているためだと考えられる。非同期更新の問題点は 6.1 節で考察する。

5.3.5 各エージェントの価値関数更新分布

図 13 は各エージェントが共有価値関数上のどの部分を更新しているかを表したグラフである。127 × 127 の迷路タスクを使用した。横軸は何回目の価値関数更新かを表すステップ番号、縦軸は更新された状態番号である。状態番号は $s * |\mathcal{A}| + a$ で求めた。ここで $|\mathcal{A}|$ はエージェントが取れる行動の総数である。

図 13 から各エージェントが更新する場所にはばらつきがあることがわかる。つまり、各ステップでエージェントが更新する価値関数の場所は異なっている確率が高いことがわかる。この結果は図 12 でエージェント数が小さいとき総更新回数が増加しない原因と考

えられる。詳しくは 6.1 節で考察する。

5.3.6 連続状態タスクにおける結果

本項では、提案手法を連続状態タスク（車の山登りタスクとアクロバットタスク）に適用した結果をまとめる。迷路タスクとは異なる価値関数形式、アルゴリズムでも同様の並列化手法で学習を高速化できることを示す。

図 14 は車の山登りタスクの学習曲線であり、並列に動作するエージェント数と学習曲線の関係を表している。横軸はエピソード数、縦軸はステップ数である。迷路タスクと同様に学習が進むにつれステップ数が徐々に小さくなっていくことがわかる。1 エピソードのステップ数が 120 以下になった時点で収束と判断した。テーブル形式の Q-Learning が最適解への収束が保証されているのに対し、線形関数形式の Q(λ) は最適解への収束が保証されていない。そのため 120 ステップが最適解とは判断できない。

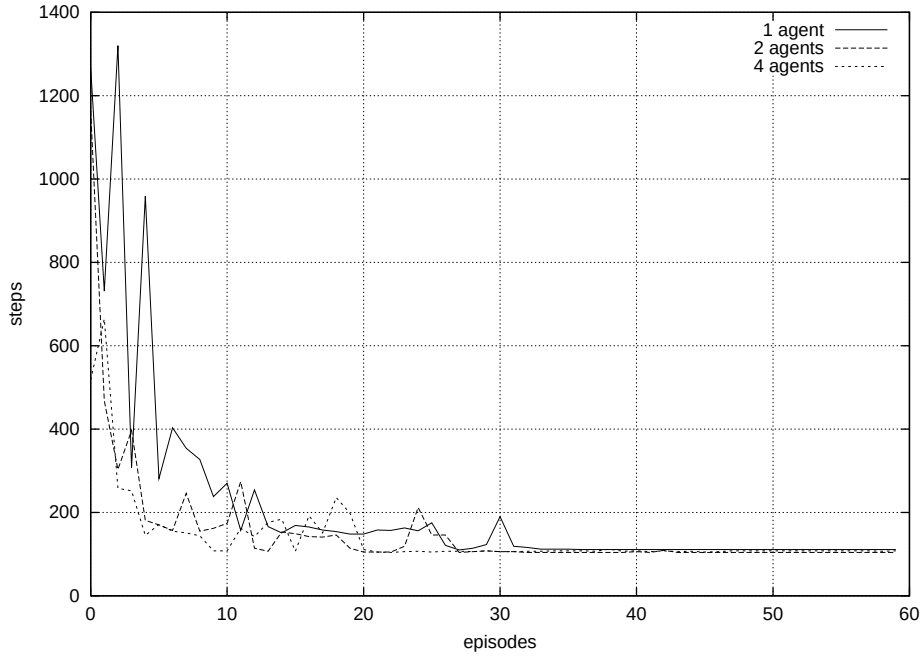

図 14: 車の山登りタスクの学習曲線

図 14 からエージェント数と収束エピソードの関係をまとめると表 8 のようになる。表

8 から価値関数の更新に参加するエージェント数が増えるにしたがい、学習の収束エピソードは早まることがわかる。

表 8: エージェント数と収束エピソード数の関係

エージェント数	1	2	4
収束エピソード数	32	27	20

表 9 は、車の山登りタスクでエージェント数を 1、2、4 としたときの収束までの時間を測定した結果である。計算機の負荷は一様ではないため 10 回の試行でもっとも速かった結果を記録した。

表 9: 車の山登りタスクにおける学習時間

エージェント数	1	2	4
学習時間（秒）	5.87	3.58	1.82
速度向上率	1.00	1.64	3.23

表 9 から価値関数の更新に参加するエージェントが増えるにつれ、学習時間が短縮されることがわかる。

アクロバットタスクでも同様の実験を行った。アクロバットタスクは迷路タスク、車の山登りタスクに比べてシミュレーションが複雑であり、計算に長い時間がかかるため並列化の結果がもっともよく現れるタスクである。

図 15 はアクロバットタスクの学習曲線であり、並列に動作するエージェント数と学習曲線の関係を表している。横軸はエピソード数、縦軸はステップ数である。迷路タスクと同様に学習が進むにつれステップ数が徐々に小さくなっていくことがわかる。1 エピソードのステップ数が 250 以下になった時点で収束と判断した。

アクロバットタスクの学習曲線を見ると分かるように完全に収束してはいない。グラフの右端の方が乱れている。この乱れはランダム行動（探査）によって生じる。アクロバットタスクは連続状態タスクであるためランダムな 1 回の行動によって棒の位置が大きくずれ、失敗してしまうことがあるためである。

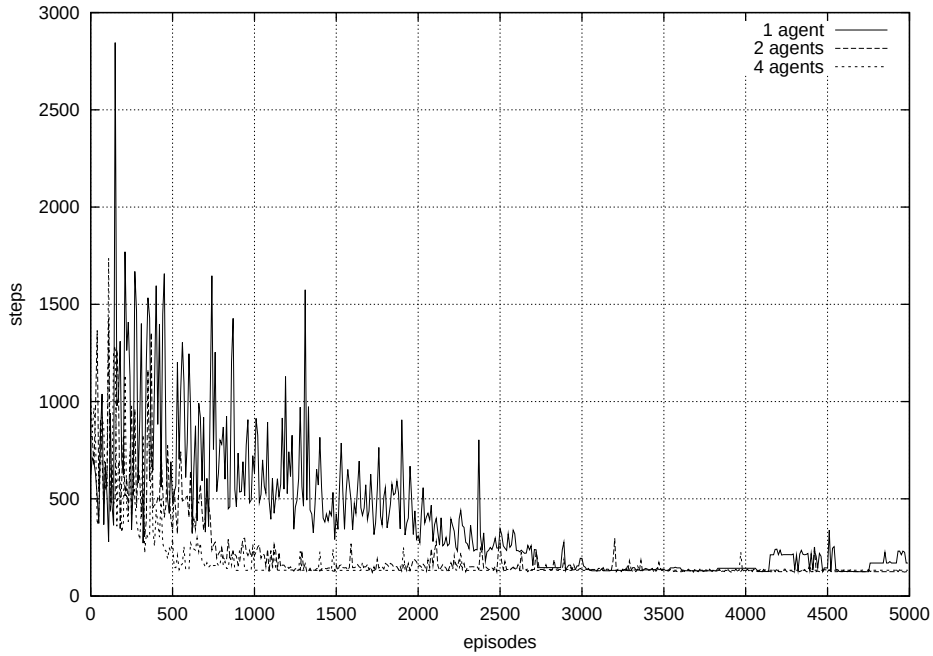

図 15: アクロバットタスクの学習曲線

図 15 からエージェント数と収束エピソードの関係をまとめると表 10 のようになる。表 10 から価値関数の更新に参加するエージェント数が増えるにしたがい、学習の収束エピソードは早まることがわかる。

表 10: エージェント数と収束エピソード数の関係

エージェント数	1	2	4
収束エピソード数	2730	1230	580

表 11 は、アクロバットタスクでエージェント数を 1、2、4 としたときの収束までの時間を測定した結果である。計算機の負荷は一様ではないため 10 回の試行でもっとも速かった結果を記録した。

表 11 から価値関数の更新に参加するエージェントが増えるにつれ、学習時間が短縮されることがわかる。

表 11: アクロバットタスクにおける学習時間

エージェント数	1	2	4
学習時間（秒）	1074.70	552.72	319.89
速度向上率	1.00	1.94	3.36

5.3.7 IBM pSeries 690 での結果

本項では、迷路タスクと車の山登りタスクを IBM pSeries 690 で実験した結果をまとめる。IBM pSeries 690 は 24 プロセッサを持つ共有メモリ型並列計算機である。この実験は本手法の速度向上の限界を調べることが目的である。

表 12、表 13 は、迷路タスクでエージェント数を 1、2、4、8、16、24、32 としたときの収束までの時間を測定した結果である。計算機の負荷は一様でないため 10 回の試行でもっとも速かった結果を記録した。

表 12: 迷路タスク（63×63）における Regatta での学習時間

エージェント数	1	2	4	8	16	24	32
学習時間（秒）	1.92	0.99	0.51	0.27	0.15	0.10	0.11
速度向上率	1.00	1.94	3.76	7.11	12.8	19.2	17.5

表 13: 迷路タスク（127×127）における Regatta での学習時間

エージェント数	1	2	4	8	16	24	32
学習時間（秒）	15.29	7.77	3.95	1.99	1.01	0.69	0.70
速度向上率	1.00	1.97	3.87	7.68	15.1	22.2	21.8

5.3.2 項での結果と同様に表 12 と表 13 から価値関数の更新に参加するエージェントが増えるにつれ、学習時間が短縮されることがわかる。エージェント数が 24 のときに比べ、32 のときの方が遅いのは使用マシンのプロセッサ数が 24 台しかないためである。

また表 12 と表 13 を比べると規模の大きい迷路の方が速度向上比が高いことがわかる。

これは、各エージェントが更新する範囲がばらつき、メモリアクセス競合が起きにくくなるためである。メモリアクセス競合については 6.1 節で考察する。

表 14 は、車の山登りタスクでエージェント数を 1、2、4、8、16、24、32 としたときの収束までの時間を測定した結果である。計算機の負荷は一様でないため 10 回の試行でもっとも速かった結果を記録した。

表 14: 車の山登りタスクにおける Regatta での学習時間

エージェント数	1	2	4	8	16	24	32
学習時間（秒）	5.25	2.77	1.59	0.82	0.53	0.57	0.66
速度向上率	1.00	1.90	3.30	6.40	9.91	9.21	7.93

5.3.5 項での結果と同様に表 14 から価値関数の更新に参加するエージェントが増えるにつれ、学習時間が短縮されることがわかる。しかし、エージェント数が 16 のときに比べ、24 のときの方が遅くなってしまった。これは、使用した Q(λ) の計算量が迷路タスクで使った Q-Learning に比べて大きいからだと考えられる。

以上の結果から IBM pSeries という異なるマシンでもほぼ同様の速度向上が引き出せることが示せた。また、プロセッサ台数にほぼ線形に比例した速度向上が出せ、オーバーヘッドが小さいことも示せた。

5.3.8 価値関数の結合手法の評価

3.2.3 節であげた [5] の手法を迷路タスクを例にして評価した結果を示す。[5] の手法では結合間隔がパラメータになっている。使用した迷路タスクは 20800 エピソードで学習が終了することが表 4 からわかっている。そこで結合間隔を 5000、2000、1 エピソードとした。また簡単のため使用するエージェント数（使用する PE 台数）を 2 とした。エージェント 1 の探査率は 0.01、エージェント 2 の探査率は 0.7 とした[*12]。表 15 に収束エピ

[*12] エージェント 1 とエージェント 2 の探査率を同じにした場合、結合の効果が全く得られなかった。[5] の結合手法は価値関数の加重平均を取っているためエージェント 1 とエージェント 2 の価値関数がほぼ一致する場合は結合（平均）しても価値関数は変わらないからである。エージェント 2 の探査率を 0.7 と高

表 15: 通信時間と結合時間の測定結果（時間の単位は秒）

結合間隔	収束エピソード	通信回数	通信時間	結合時間	残り時間	全体時間
なし	20305	-	-	-	-	17.71
5000	18797	4	3.78	0.01	16.66	20.45
2000	16010	9	4.37	0.02	16.18	20.57
1	8903	8904	397.27	10.18	15.64	423.09

ソード、通信時間、結合時間、全体時間を測定した結果を示す。

通信回数は 収束エピソード ÷ 結合間隔 + 1 である。通信時間は結合間隔おきにエージェント同士が価値関数を送るのに要した時間の合計である。結合時間は送られてきた価値関数と自分の価値関数を式 (6) で結合するのに要した時間である。全体時間は学習が収束するまでに要した時間である。残り時間は全体時間から通信時間と結合時間を除いた時間であり、学習や同期待ちに要した時間が含まれる。「なし」は 1 台の PE のみを使い結合を行わなかった場合の結果である。本研究では実行時間の短縮が目的であるため並列化した場合に 17.71 秒より短くなることが要求される。

この結果から結合間隔が短いほど収束エピソードは小さくなることがわかる。これは [5] が指摘している学習の高速化である。全体時間を見てみると並列化した場合の方がしなかった場合より計算時間が多くなってしまい並列化の効果が出なかった。この計算時間増大の原因は結合にあるのではなく、主に通信時間にある。この結果については 6.3.3 項で考察する。

く設定するとエージェント 2 はエージェント 1 が担当しない価値関数領域を更新するため結合効果が出やすくなる。

6 考察

6.1 共有メモリ型並列計算機における非同期並列更新の有効性

本節では、提案した共有メモリ型並列計算機における非同期並列更新の有効性を考察する。特に価値関数の更新が非同期であることの欠点と利点について述べる。

非同期更新は同期処理のオーバーヘッドがないために高速という利点がある（表 7）が、図 16 のような場合に問題が生じる。すなわち、エージェント 1 が価値関数の値を読み込み (1)、新しい価値を計算し (2)、価値関数へ値を書き戻す (4) 前にエージェント 2 が同じ場所から価値関数の値を読み込んで (3) しまう場合である。このとき、エージェント 1 の更新はエージェント 2 の更新に打ち消されて無駄になる。

この問題は非同期時に特有であり、同期時には生じない。図 16 の現象が発生していることは図 12 でエージェント数を増やしたとき収束までの総更新回数が増えていることからもわかる。エージェント数が増えるにしたがって無駄になる更新が増え余分に更新しなければならないからである。図 16 の現象が頻繁に発生すると価値関数の整合性が取れな

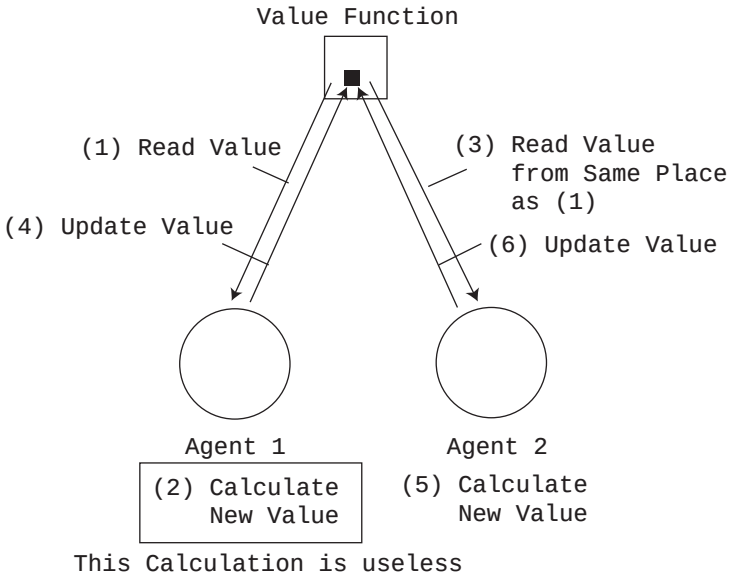

図 16: 非同期更新で無駄が生じる場合

くなり収束性が失われるという問題点もある。図 12 の実験では 4096 エージェントで収束しなくなるという結果が得られた。

本実験環境では 4 エージェントまたは 24 エージェントまでは学習が収束し、かつ学習時間が短縮されることは示されている。一般に共有メモリ型並列計算機に搭載できるプロセッサ数は 2004 年時点では約 100 である[*13]。128 プロセッサでの総更新回数の増加率（非同期更新の無駄によるオーバーヘッド）を図 12 で見てみると小さい迷路環境では高々 11%、大きい迷路環境では高々 2%であることがわかる。並列化したいタスクは迷路に比べてさらに大規模なタスクであることから考えると非同期更新のオーバーヘッドは無視できるほど小さいと考えられる。

よって本手法の速度向上は限界があるにせよ適切なエージェント数の範囲（共有メモリ型並列計算機で並列に動作させることのできるエージェント数の範囲）では図 16 の影響は小さい。

その理由は図 13 のように各エージェントが更新する価値関数の場所が異なっているからだと考えられる。価値関数の更新範囲が広い大規模な問題の場合、並列に動作している各エージェントが同じタイミングで同じ場所を更新してしまう確率は低い。更新する場所が異なれば図 16 の現象は発生しない。

6.2 タスク依存性

提案手法は強化学習の根本的な部分（経験を得て価値関数を更新する部分）を並列化の対象としているためタスクやアルゴリズムによらず並列化の効果が得られるという利点がある。実験では、価値関数の形式とアルゴリズムの組み合わせとして

- テーブル型価値関数× Q-Learning
- 線形関数型価値関数× Q(λ)

の 2 つを評価した。どちらの場合もほぼ同様の並列化の効果が得られた。

[*13] Sun の Sun Fire 15K は 106 プロセッサを搭載できるハイエンド・サーバーである

価値関数の他の形式としてニューラルネットがある。ニューラルネットでは実験を行わなかったが、線形関数と同じく重みパラメータの更新であり、かつその更新方法が [21] と同じであるため並列化の効果は同様に出せると考える。

強化学習アルゴリズムには Q-Learning のほかに TD 法や Profit Sharing といったアルゴリズムもある。これらのアルゴリズムも基本的には Q-Learning と同じく経験による価値の更新である[*14]ため提案手法が適用できると考える。

また、Sun Ultra 80 と IBM pSeries 690 という異なる種類のマシンでもほぼ同様の速度向上を達成できた。ただし、オペレーティングシステムの種類に応じて速度向上が得られないという結果も出た。たとえば、Redhat Linux 9 ではエージェント数（スレッド数）を増やすと逆に速度は低下してしまった。これは Redhat Linux 9 がスレッドスケジューリング時にキャッシュを再利用しない場合があるからである。たとえば、スレッド 1 が CPU1 に割り当てられた場合、このスレッドの処理に使われる価値関数の値はメモリから CPU1 のキャッシュにコピーされる。スレッド 1 がタイムスライスを使い果たし、再スケジューリングされる際に今まで使用していた CPU1 に再び割り当てられればキャッシュを再利用できる。しかし、別の CPU に割り当てられてしまうとキャッシュを利用できないため速度が低下してしまう。OS のキャッシュの扱いには注意する必要がある。

6.3 分散メモリ型並列計算機への実装可能性

本節では強化学習を分散メモリ型並列計算機へ実装する方法について考察する。まず並列計算機の種類とその特徴について簡単にまとめる。次に分散メモリ型並列計算機へ実装した従来手法 [9] [5] について考察を行う。[5] の手法は実際に並列計算機へ実装して速度向上を測定したデータがないため自身で実験を行った。その結果（5.3.8 項）をもとに考察する。

[*14] Profit Sharing では価値という言葉は使わないが状態・行動対の重みを更新するという点では本質的な差はないと考える

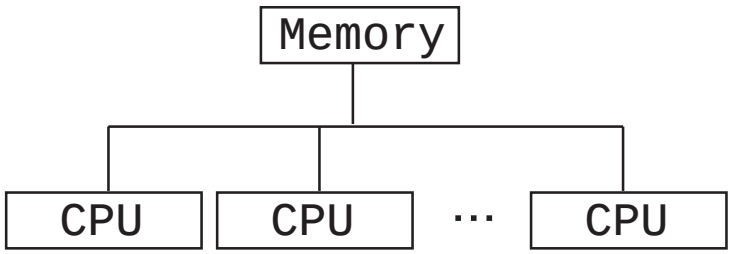

図 17: 共有メモリ型並列計算機の構成

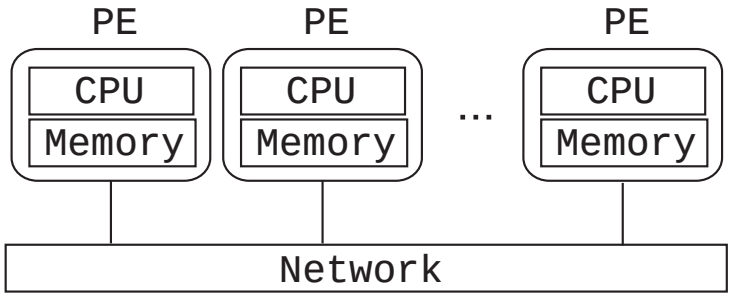

図 18: 分散メモリ型並列計算機の構成

6.3.1 並列計算機の種類

並列計算機には 2 つのタイプがある。共有メモリ型並列計算機と分散メモリ型並列計算機である。共有メモリ型並列計算機は 1 つの PC の中に複数のプロセッサが内蔵されておりメモリを共有するタイプの計算機である（図 17)。一方、分散メモリ型並列計算機は複数の PC がネットワークでつながり、各 PC がメッセージを授受することで並列動作するタイプの計算機であり、PC クラスタとも呼ばれている（図 18)。各並列計算機の特徴をまとめると表 16 のようになる。

提案手法で用いたのは共有メモリ型並列計算機である。共有メモリ型並列計算機は高価であり、24 プロセッサを搭載した並列計算機を用意することは難しい。そこで安価でコストパフォーマンスのよい分散メモリ型並列計算機を用いて強化学習の並列化が可能ならさらに利用価値が高いと考えられる。従来提案されている方法は [9] による価値関数の分割更新手法と [5] による価値関数の結合手法である。

表 16: 各並列計算機の特徴

	共有メモリ型並列計算機	分散メモリ型並列計算機
利点	● データ分割を考慮しなくてよい ● プログラミングが容易 ● プロセッサ数が 1 桁台であれば速度向上比が高い	● 拡張性がある ● コストパフォーマンスが高い
欠点	● メモリアクセス競合が生じる ● 同期コストがかかる ● 価格が高い	● 通信コストがかかる ● データ分割を適切に行う必要がある ● プログラミングが複雑

しかし、筆者の結論では強化学習を分散メモリ型並列計算機を用いて並列化するのは難しいと考える。以下、その理由を交えて考察を行う。

6.3.2 価値関数の分割更新手法

価値関数の分割更新手法は一般的にデータ分割と呼ばれる手法である。データ分割を用いるにはタスクを各々独立になるようにサブタスク分割し、並列に計算できる問題に限られる[*15]。強化学習におけるデータは更新の対象となる価値関数である。価値関数の分割について考えてみると各々が独立になるように分割することは難しいことがわかる。その理由として価値関数の更新には次状態の価値関数の値が必要なことがあげられる。たとえば、図 19 を例に考えてみる。

図 19 は迷路環境における強化学習である。迷路の各マスは 1 つの状態にあたるため価値関数と 1 対 1 の対応関係がある。すなわち、価値関数を分割して各プロセッサに割り当

[*15] たとえば、遺伝的アルゴリズムでは個体の母集団を分割した後は互いに干渉せず独立に評価できるため並列化しやすいと言われている

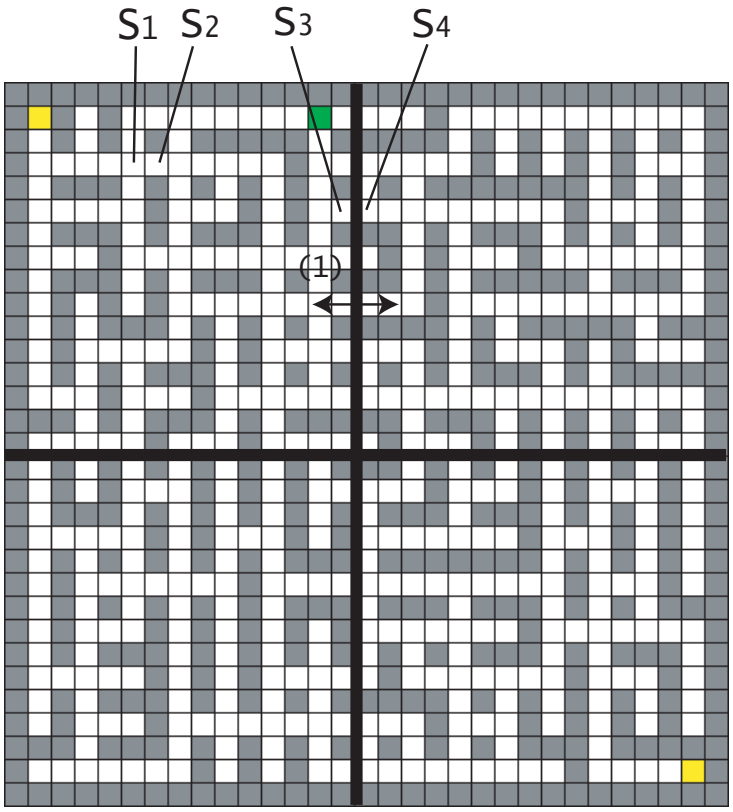

図 19: プロセッサ間通信が頻繁に発生する場合

てることは迷路の範囲を分割して各範囲を複数のプロセッサで並列に学習することに相当する。今回は図 19 のように 4 等分して 4 つのプロセッサに割り当てたと仮定する。価値関数の更新には次状態の価値関数の値が必要である。たとえば、状態 s_1 から状態 s_2 に遷移した場合、状態 s_1 の価値関数の更新には状態 s_2 の価値関数の値を使って、

$$Q(s_1, a) \leftarrow Q(s_1, a) + \alpha[r + \gamma \max_{a' \in A} Q(s_2, a') - Q(s_1, a)] \tag{9}$$

のように行われる。状態 s_1 と状態 s_2 がともに同じ PE 上にあればプロセッサ間通信は生じないが、状態 s_3 と状態 s_4 のように互いに異なる PE 上にある場合はプロセッサ間通信が生じてしまう（s_3 を持つ PE が s_4 を持つ PE から s_4 の価値関数の値を得なければならない)。さらに悪いことに強化学習では ϵ-greedy などを用いて行動選択にランダム性を持たせるため図 19 の (1) のように分割された領域の境界上でエージェントが往復を繰り返す場合がある。この場合、プロセッサ間通信が頻繁に発生し、その通信コストによって

並列化の効果が著しく落ちることが予想される。

6.3.3 価値関数の結合手法

表 15 からわかるように [5] の手法の主なボトルネックは通信時間にある。筆者は価値関数の結合にかかる時間が大部分を占めると考えていたが予想外の結果であった。

この通信時間を抑えるために考えられる方法として

- 通信回数（価値関数の結合回数）を減らす。
- 1 回の通信量を減らす。

が考えられる。価値関数の結合手法は価値関数を結合することによって学習の高速化を達成するのが目的である。通信回数（結合回数）を減らすためには 1 回の結合による学習効果を増やすしかない。これを達成するために必要な新たな問題は次のように定義できる。

> 互いに独立に学習された 2 つの価値関数があるとする。この 2 つの価値関数からより最適価値関数に近い価値関数を再学習なしに作り出すことができるか。

[5] では 2 つの価値関数の加重平均を取るという単純な方法を用いていた。この方法は経験数の減少には効果があるが、計算時間を短縮するほどの効果はない。さらによい方法を考え出す必要がある[*16]。

2 つめの 1 回の通信量を減らすについては使用する価値関数の形式を変えることである程度改善できると考えられる。今回の実験では価値関数の形式にテーブルを用いている。テーブルの大きさは 127×127 の状態に加えそれぞれの状態に 4 つの行動がある。価値関数は double 型なので $127 \times 127 \times 4 \times 8$ で 516,128(Byte) である。さらに各価値関数の更新回数を記録したテーブルも送る必要がある。こちらは int 型なので $127 \times 127 \times 4 \times 4$ で 258,064(Byte) である。よって、1 回の結合の通信量は 774,192(Byte) となる。これは通信量としては大きい。3.1.2 項で述べたように関数近似を使えば価値関数をよりコンパ

[*16] この問題は複数エージェントが持つ経験の効率的な共有という点でも面白い問題だと考えている

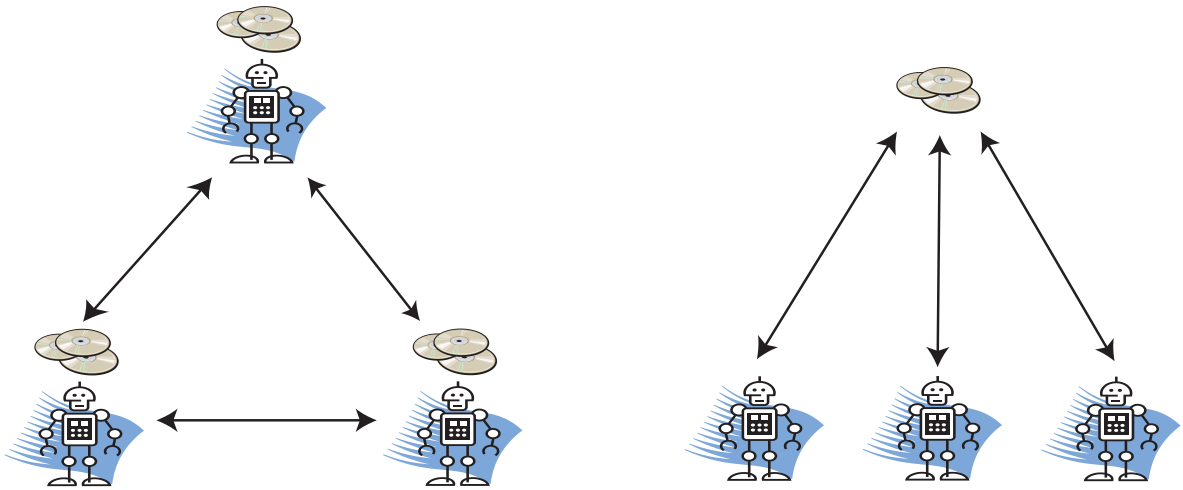

図 20: ロボットにおける価値関数を分散する手法と価値関数を共有する手法

クトなパラメータで表すことが可能である。パラメータのサイズはテーブルに比べてはるかに小さいため通信料を大幅に減らすことができると考えられる。

6.4 実ロボットへの適用可能性

提案手法の 2 つ目の欠点はシミュレーションのみを対象にしている点である。提案手法では環境との相互作用を行うエージェントが複数存在し、それらが並列に動作することを仮定している。この仮定はシミュレーション上の学習では容易に実現ができるが実ロボットの学習では実現が難しい。なぜならまったく同じロボットを複数用意するのはコストと学習時間のトレードオフを考えると非現実的だからである。並列化のためだけに同じロボットを複数製作するよりは、1 体のみで学習させた方がかかる時間は短いと考えられる。

ただし、実ロボットでも複数のロボットを用意できれば並列化による学習の高速化は可能であると考える。AIBO[22] や Khepera[23] のような価格の安い小型ロボットでは複数用意することも可能だろう。そこで、ロボットを用いた強化学習の並列化を考えてみたい。価値関数を分散する手法と価値関数を共有する手法をロボットに当てはめてみると図 20 のようになるだろう。

価値関数を分散する手法 (a) では、互いに独立したエージェントがそれぞれ環境との相互作用を行う。そして、強化学習の学習成果であり、エージェントの記憶（経験）とも言える価値関数をメッセージパッシングというコストがかかる手段を用いて交換・結合するイメージである。それに対し、価値関数を共有する提案手法 (b) では、エージェントの記憶（経験）である価値関数を全エージェントが即座に共有できるイメージである。学習効率という点では共有していた方が優れているのは明らかである。メッセージパッシングは必要ないからである。

人間は記憶の共有がまだ不可能であるため前者のアプローチを取らざるを得ない。記憶（経験）のメッセージパッシングは人間ではコミュニケーションにあたると考えられる。メッセージパッシングを行って価値関数（経験）の結合を行ってもさして効果が出ないという結論（5.3.8 節）は人間同士のコミュニケーションが完全にうまくいかず情報伝達の効率が悪いことの比喩になっているようで興味深い。しかし、ロボットは人間と同じ方法を取る必要はない。記憶を共有する後者のアプローチを取ることができる。複数のロボットが並列に学習を行いその経験を即座に全ロボットに及ぼすことが可能となる。このように経験の共有は分散に比べて学習効率を向上させることにつながると考える。

7 おわりに

強化学習は環境との相互作用から得られる報酬をもとに適切な行動戦略を学習するアルゴリズムとして近年注目を集めている。しかし、強化学習は知識がない状態からの試行錯誤学習であるため学習が遅いという大きな欠点がある。この問題を解決するために強化学習高速化に関する研究は盛んに行われている。

本論文では、強化学習高速化に関する研究を

- 学習の収束に要する経験数の減少を目的とした研究。
- 学習の収束に要する計算時間の短縮を目的とした研究。

に分類しまとめた。特に後者の一部である並列化に関する研究については詳細をまとめた。従来の強化学習並列化手法は分散メモリ型並列計算機を対象とした手法だけであり、オーバーヘッドによる速度向上の低下が問題となっていた。

そこで本論文では共有メモリ型並列計算機を対象に共有した価値関数を非同期並列的に更新する手法を提案し、学習の高速化に有効であることを示した。また従来の並列化はテーブル形式の価値関数 ×Q-Learning の環境でしか行われていなかったが、本論文ではそれに加えて線形関数形式の価値関数 ×Q(λ) の環境でも並列化の効果が出せることを示した。共有メモリ型並列計算機は高価であるが、近年普及が目覚しいデュアルプロセッサ環境で本手法を使えば少なくとも約 2 倍の速度向上は出せるであろう。デバッグを繰り返すたびに長い学習時間を待たなくてはならない開発者にとっては 2 倍の速度向上でも十分利用価値があると考える。

本論文は並列化による学習時間の向上に焦点を当てた。しかし、サーベイを進めても強化学習の並列化に関する文献は少ないというのが現状であった。これは同じく学習の高速化が問題となっているニューラルネットや遺伝的アルゴリズムの並列化が 1 つの分野を形成しているのとは対照的である。そこで、サーベイの範囲を並列化に限らずマルチエージェント強化学習にまで広げてみたところ並列化手法として使えそうな考え方をいくつか発見することができた。たとえば、価値関数の結合手法としてあげた [5] やマルチエー

ジェントによる経験の共有を扱った [16] などがあげられる。一般的なマルチエージェント強化学習の研究は、ジョイントタスク[*17]の学習に焦点が当てられている。しかし、並列化とマルチエージェント強化学習の関係を追究していくのも興味深いと考えている。

[*17] 複数のエージェントが協力しないと目標を達成できないようなタスク

謝辞

本論文の作成にあたり数々のご指導をいただいた山名早人助教授に感謝します。

参考文献

[1] Antonova, D.: Parallel Reinforcement Learning - Extending the Concept to Continuous Multi-State Tasks, thesis, Denison University, 2003.

[2] Cichosz, P.: Truncating Temporal Differences: On the Efficient Implementation of TD(λ) for Reinforcement Learning, Journal of Artificial Intelligence Research, Vol.2, pp.287-318, 1995.

[3] 片山晋, 小林重信: TD(λ) の対数時間更新算法, 人工知能学会誌, Vol.14, No.5, pp.879-890, 1999.

[4] 北野宏明: 遺伝的アルゴリズム 4, 産業図書, 2000.

[5] Kretchmar, R. M.:Parallel Reinforcement Learning, Proc. of the 6th World Conf. on Systemics, Cybernetics and Informatics, Vol.6, pp.114-118, 2002.

[6] Maclin, R. and Shavlik, J. W.: Creating Advice-Taking Reinforcement Learning, Machine Learning, Vol.22, pp.251-281, 1996.

[7] 宮崎和光, 山村雅幸, 小林重信: k-確実探査法: 強化学習における環境同定のための行動選択戦略, 人工知能学会誌, Vol.10, No.3, pp.454-463, 1995.

[8] Peng, J., Williams, R. J.: Incremental Multi-Step Q-Learning, Machine Learning, Vol.22, pp.283-290, 1996.

[9] Printista, A. M., Errecalde, M. L. and Montoya, C. I.: A Parallel Implementation of Q-Learning Based on Communication with Cache, Journal of Computer Science and Technology, Vol.1, No.6, 2002.

[10] Rosenstein, M. T. and Barto, A. G.: Supervised Actor-Critic Reinforcement Learning, In Si, J., Barto, A. G., Powell, W. and Wunsch, D. eds. Learning and Approximate Dynamic Programming: Scaling Up to the Real World, John Wiley & Sons, Inc., pp.359-380, 2004.

[11] Russell, S. and Norvig, P.: Artificial Intelligence A Modern Approach 2nd Edition, Prentice Hall, 2003.

[12] Singh, S. P. and Sutton, R. S.: Reinforcement Learning with Replacing Eligibility Traces, Machine Learning, Vol.22, pp.123-158, 1996.

[13] Sutton, R. S.: Integrated Architectures for Learning, Planning and Reacting Based on Approximating Dynamic Programming, Proc. of the 7th International Conf. of Machine Learning, pp.216-224, 1990.

[14] Sutton, R. S.: Dyna, an Integrated Architecture for Learning, Planning and Reacting, Working Notes of the AAAI Spring Symposium, pp.151-155, 1991.

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

[16] Tan, M.: Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents, Proc. of the 10th International Conf. on Machine Learning, pp.330-337, 1993.

[17] Tesauro, G. J.: Practical Issues in Temporal Difference Learning, Machine Learning, Vol.8, pp.257-277, 1992.

[18] Tesauro, G. J.: Temporal Difference Learning and TD-Gammon, Communications of the ACM, Vol.38, pp.58-68, 1995.

[19] Watkins, C. J. C. H. and Dayan, P.: Technical Note: Q-Learning, Machine Learning, Vol.8, pp.55-68, 1992.

[20] 山口智浩, 増渕元臣, 田中康祐, 谷内田正彦: 経験型強化学習における仮想個体から実ロボットへの学習行動の伝播, 人工知能学会誌, Vol.12, No.4, pp.570-581, 1997.

[21] 山森一人, 堀口進: 並列計算機上での誤差逆伝播法の並列学習モデル, 信学論 (D-II), Vol.J81-DII, No.2, pp.370-377, 1998.

[22] AIBO Official Site
http://www.jp.aibo.com/

[23] K-Team Corporation
http://www.k-team.com/

研究業績

研究会

森紘一郎, 山名早人: 強化学習並列化による学習の高速化, 情処研報 (ICS), Vol.2004, No.29, pp.89-94, 2004.

全国大会

森紘一郎, 山名早人: 共有メモリ型並列計算機上での強化学習の並列学習法, 情報科学技術フォーラム (FIT), F-33, 2004.