Vol.2014-GI-31 No.6
2014/3/17

分散計算環境における並列パーセプトロンの将棋評価関数への適用

浦 晃[1,a)]　三輪 誠[2,b)]　鶴岡 慶雅[1,c)]　近山 隆[1,d)]

概要：将棋の評価関数の学習時間を短縮することができれば、学習パラメータの調整などに時間をかけられるようになるため有意義である。そこで、本研究では、将棋プログラムである激指の評価関数の学習を、分散計算環境で並列化した。激指の評価関数の学習にはパーセプトロンが用いられている。パーセプトロンの並列化手法として、ミニバッチを用いる手法が提案されているため、これを評価関数の学習に適用した。学習の評価には棋譜との一致率を用いた。64 台の計算機を用いた評価では、ある決められた一致率を実現するための学習時間を 1/11.6 に短縮できることを示した。

Application of Parallel Percepron to Shogi Evaluation Functions on Distributed Computing Environments

Akira Ura[1,a)]　Makoto Miwa[2,b)]　Yoshimasa Tsuruoka[1,c)]　Takashi Chikayama[1,d)]

Abstract: It is worthwhile to shorten the training time for shogi evaluation functions since that would enable us to use more time to adjust training parameters. In this work, we have parallelized the training for the evaluation function of Gekisashi, a shogi program, on a distributed computing environment. The perceptron is used in the training for Gekisashi's evaluation function. We have applied an existing method for parallel perceptron training using mini-batches to the training of Gekisashi's evaluation function. Agreement rates to moves in game records are used to evaluate the training. Experimental results with 64 computing nodes show that the training time to reach a certain agreement rate was shortened to 1/11.6.

1. はじめに

コンピュータ将棋では、強いプレイヤを作るためには、局面の有利不利を判定する評価関数の精度を高くしなければならない。近年では、評価関数のパラメータは機械学習の技術を用いて自動的に調整するのが主流になっている。その中でも、プロ棋士の棋譜を評価関数の学習に用いた比較学習 [11] が成功を収めており [16]、研究がなされている [6], [8]。

本研究では、比較学習による評価関数の学習にかかる時間に着目する。評価関数は探索中で用いられるため、学習の結果を用いるときにかかる時間は短くなければならない。これに比べて、評価関数の学習そのものにかかる時間は、実際に用いるときの性能に直接は影響しないため、学習が実行可能な範囲であれば長くても構わない。しかし、評価関数の学習の時間も短ければ短いほど有利である。

学習時間が短ければ、例えば、学習パラメータの調整に時間をかけられるようになる。実際に評価関数を学習させるときには、学習手法に伴う学習パラメータを適切に調整する必要がある。評価関数の学習が高速に実行できれば、一回の学習を行い、それに応じて学習パラメータを少し変更し、再び学習を行うというように、学習パラメータの調

[1] 東京大学大学院工学系研究科
Graduate School of Engineering, The University of Tokyo
[2] マンチェスター大学情報科学科
School of Computer Science, The University of Tokyo
[a)] ura@logos.t.u-tokyo.ac.jp
[b)] makoto.miwa@manchester.ac.uk
[c)] tsuruoka@logos.t.u-tokyo.ac.jp
[d)] chikayama@logos.t.u-tokyo.ac.jp

整が容易になる。さらに、より適切な学習パラメータを選ぶことができるようになるため、学習の精度そのものを高めることができる可能性もある。

学習が高速に実行できるようになると、使える棋譜の数を増やせるという利点も存在する。比較学習はプロ棋士の指し手を教師信号とする教師あり学習である。一般に教師あり学習は、用いる訓練データが多いほど、精度のよい学習が可能である。プロ棋士の棋譜の数には限りがあるが、コンピュータプレイヤ自身を用いて訓練データを生成することもできると考えられ、我々も試みている [12]。同時に、学習に有効な局面を自動的に選択する研究 [15] も行っている。もし、学習に有効な訓練データを自動生成できるようになると、大きな訓練データを用いるために学習に時間がかかるようになるため、学習が高速に実行できるようになることは重要である。

本研究では、将棋の評価関数の学習を高速に行うことを目的として、分散計算環境において並列化することを提案する。将棋プログラムとしては激指を用いた。激指の学習手法にはパーセプトロンが用いられている。パーセプトロンについては、ミニバッチを用いた並列化手法が提案されている [14]。そこで、この並列化手法を将棋の評価関数の学習に適用した。学習の精度の評価には、棋譜の指し手との一致率を用いた。64 計算ノードを用いた実験では、1 計算ノードでの学習と比較して、ある基準となる一致率を実現するための学習時間を短縮できることを示した。

本稿の構成は以下の通りである。まず、2 節において、激指における学習手法について述べる。続く 3 節において、パーセプトロンも含めたオンライン学習の並列化の関連研究を紹介する。4 節では、分散計算環境で学習を並列化する提案とその実装について述べる。5 節では、行った実験と結果について述べる。最後に 6 節で、まとめと今後の課題を述べる。

2. 激指の学習手法

本研究は、将棋プログラムである激指を実験に用いているので、激指が行っている学習手法について述べる。激指は次式で表されるような線形の評価関数を用いている。

$$f(\boldsymbol{w}, s) = \boldsymbol{w}^T \boldsymbol{\phi}(s) \tag{1}$$

ただし、$\boldsymbol{w}$ は重みベクトル（調整するパラメータ）、$\boldsymbol{\phi}(s)$ は局面 s に対する特徴ベクトルである。特徴ベクトルの要素数は全部で約 1,900,000 である。

激指の学習では、比較学習 [11] と平均化パーセプトロン (Averaged Perceptron) [2] を用いている。まず、重みベクトルの初期値として、各駒の重みなどいくつかの要素についてはヒューリスティックな値とし、その他の重みは 0 にする。パーセプトロンの更新式は次式で与えられる。

$$\boldsymbol{w}^{(t)} = \boldsymbol{w}^{(t-1)} + \frac{1}{|S^{(t)}|} \sum_{s_i \in S^{(t)}} (\boldsymbol{\phi}(s_1) - \boldsymbol{\phi}(s_i)) \tag{2}$$

ただし、t は更新回数、s_1 はプロの手を進めた後の局面から探索した最善応手手順の先のリーフ局面、s_i $(i \geq 2)$ はプロが選ばなかった手を進めた後の局面からの最善応手手順の先のリーフ局面である。$S^{(t)}$ は t 回目の更新において、プロの手より良いと判断されたか、あるいは、あるマージンの値以内しかプロの手との評価値の差がなかった手の後の最善応手手順の先のリーフ局面の集合であり、次式で表される。

$$S^{(t)} = \Big\{ s_i \big| (i \geq 2) \wedge \Big(f(\boldsymbol{w}^{(t-1)}, s_1) < f(\boldsymbol{w}^{(t-1)}, s_i) + m \Big) \Big\} \tag{3}$$

$S^{(t)}$ に含まれる局面の数を $|S^{(t)}|$ と表記した。m はマージンであり、激指では局面の進行度が大きく終盤に近いほどマージンも大きくとられている。

最終的に出力される重みベクトルは、学習の途中で得られた重みベクトルの平均ベクトルとして、次のように計算される。

$$\boldsymbol{w}^* = \frac{1}{T+1} \sum_{t=0}^{T} \boldsymbol{w}^{(t)} \tag{4}$$

ただし、T は学習における更新回数の合計である。

3. 関連研究

学習手法は大きくバッチ学習とオンライン学習の二つに分けられる。バッチ学習は、訓練データを全部処理してから重みベクトルを更新する方法である。これに対し、オンライン学習は、訓練例を一つ処理するごとに重みベクトルを更新していく方法であり、大規模データの学習に用いられる [13]。

バッチ学習は訓練データを分割することで容易に並列化が可能である。各プロセスに訓練データの一部分を分配し、プロセスはそれぞれ与えられた訓練データを処理し、最後にその結果を集約すればよい。これに対し、オンライン学習では、訓練例を一つずつ処理するという意味で、本質的に逐次的な処理であるため、バッチ学習より並列化が難しい。

激指では重みパラメータの調整にパーセプトロンを用いていることを 2 節で述べた。パーセプトロンはオンライン学習の一つである。ここでは、オンライン学習の並列化に関する先行研究について述べる。オンライン学習を並列化する場合に考慮しなければならないのは、「重みベクトルの更新をどの程度行わなくてもよいか」という点になる。

パーセプトロンの並列化の手法として、parameter mixing [9] がある。Parameter mixing の概要を図 **1** に示す。各プロセスは訓練データの一部分だけを持つ。そして、自

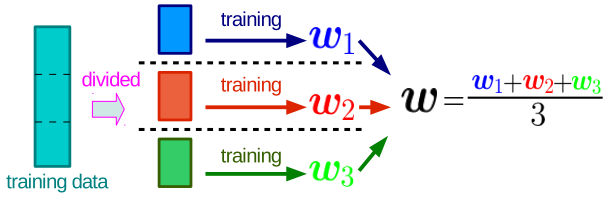

図 1　Parameter mixing

分が持っている訓練データだけを用いて重みベクトルの更新を行う。各プロセスが学習した重みベクトルの平均ベクトルが、最終的なシステム全体の重みベクトルとして出力される。必要ならば、出力される重みベクトルを初期値として、学習を繰り返す (iterative parameter mixing; IPM)。Parameter mixing は、[5] や [7] でも用いられている。

パーセプトロンの並列化を行った他の研究として、[14] がある。これは、訓練データをミニバッチと呼ばれる小さな訓練データに分割し、重みベクトルの更新をこのミニバッチが終わるごとに行う手法である。並列に実行する際には、各ミニバッチをさらに分割して、プロセスに訓練例を分配し、各プロセスが重みベクトルの更新ベクトルを求める。評価には、品詞タグ付けが用いられているが、各プロセスの処理量を均一にするために、各プロセスが処理する文章の長さがなるべく同じになるような工夫をしている。速度向上としては、12 プロセスで 5.6 倍が報告されている。IPM の速度向上は 4 倍だったことから、IPM に対する優位性も報告されている。

パーセプトロン以外にも、確率的勾配法 (stochastic gradient descent; SGD) を非同期に並列化する手法が研究されている [1], [3], [4]。これらは、ワーカは少数の訓練データから独立に勾配ベクトルを求め、マスタがワーカからそれを受け取り重みベクトルを更新していくという手法を採用している。このとき、ワーカは、マスタが持っている本当の重みベクトルではなく、いくらか情報が古い重みベクトルを使って勾配ベクトルを求めているため、この情報がどのくらい古くてもよいのか、といったことが議論されている。

4. 提案手法

4.1 並列パーセプトロンの将棋評価関数への適用

本研究では、コンピュータ将棋の評価関数の学習時間を短縮することを目的として、[14] で提案されているミニバッチを用いた並列パーセプトロンを用いて、学習を分散計算環境で並列化することを提案する。学習の大まかな手続きを図 **2** に示す。各プロセスは B 個の訓練例を用いて、他のプロセスと独立にパーセプトロンによって重みベクトルを更新した後、更新ベクトルをお互い通信し、そのベクトル和を求め、全体の重みベクトル $w^{(t)}$ を更新する。この全体の重みベクトル $w^{(t)}$ は全てのプロセスが保持する。

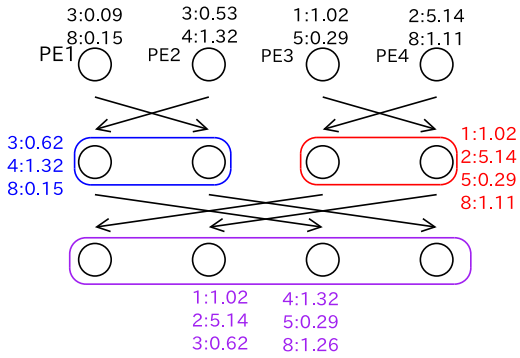

図 3　非零ベクトルの通信の例

各プロセスは更新されたベクトルを初期値として、次の B 個の訓練例を処理する。平均化パーセプトロンでは、全体の重みベクトル $w^{(t)}$ の各 t による平均ベクトルを求める。

更新ベクトルの通信の際には、通信量を減らすために非零要素だけを通信する必要がある。ベクトルの重みベクトルの要素数は約 1,900,000 であるが、一つの訓練例の処理で更新される重みベクトルの要素数はずっと少なく、数百から数千程度である。つまり、更新ベクトルはほとんどの要素が 0 のスパースなベクトルである。したがって、重みベクトルをそのまま通信するのではなく、更新ベクトルの非零要素だけを通信することが重要である。非零要素だけを通信するためには、そのインデックスも通信する必要がある。しかし、更新ベクトルが十分にスパースならば、非零要素だけを通信したほうが通信量が少なくて済む。

今回の実装では、非零要素だけを効率よく通信するために、バタフライを用いた集合通信 [10] を実装した。要は、非零要素だけを通信して、MPI_Allreduce() に相当することを行う。図 **3** に更新ベクトルの和を求める通信の例を示す。更新ベクトルはコロンの左側に重みベクトルのインデックスを、右側に更新量を書いている。二つのプロセスで通信したとき、同じインデックスの要素があれば、その時点で足し合わされる。バタフライにより、全てのプロセスが更新ベクトルの和を得るまでに、一つのプロセスが通信する回数を、プロセス数を P として、$\log P$ 回に抑えることができる。また、通信される要素数は、はじめは少なく、通信のたびに増えていくので、通信量が多いのは最後の数回の通信のみとなる。

各プロセスでの処理を詳細を述べる。各プロセスは訓練例を一つ処理するたびに、自身の重みベクトルを式 2 により、他のプロセスとは独立に更新する。次の訓練例を処理するときには、この更新された重みベクトルを用いる。なお、[14] ではこのローカルな重みベクトルの更新は行われていない代わりに、収束性が保証されている。本研究では、収束性が保証されていないとしても、使うことができる情報は使った方がいいと判断した。重みベクトルの更新の際には、その更新ベクトルも保存しておく。このとき、図 3 で示したように、更新を起こした要素のインデックスとそ

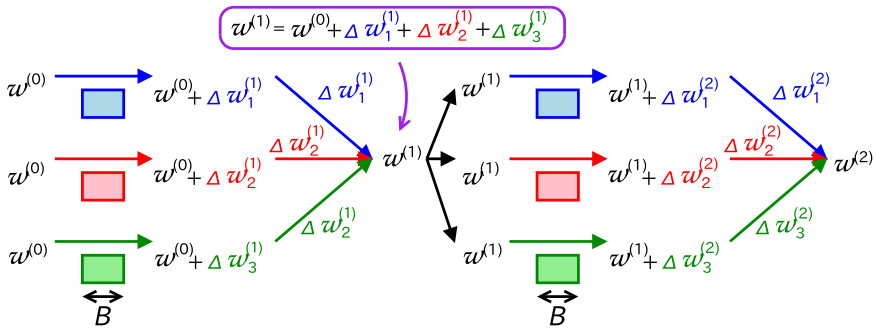

図 2　ミニバッチを用いた並列オンライン学習

の更新量を保持しておく。更新が起こった要素が以前の訓練例でも更新されていたら、その更新量に今の訓練例の分を加えていく。これを繰り返すと、B 個の訓練例を処理した後、更新された重みベクトルの要素のインデックスとその更新量が得られる。これを前述した図 3 の方法を用いて、お互いに通信しあう。通信後、すべてのプロセスは各プロセスが求めた重みベクトルの更新量の和を持っていることになるので、これを B 個の訓練例を処理する前のベクトルに加える。この操作により、次の B 個の訓練例を処理する前には、全プロセスは同じ重みベクトルを持っていることになる。

4.2　ミニバッチの大きさを時間で制限

比較学習による将棋の評価関数の学習では、探索が実行されるため、訓練例によって処理時間のばらつきが大きい。したがって、B 個の訓練例を処理する場合でも、あるプロセスはすぐに終わるのに、別のプロセスは時間がかかり、結果としてアイドル状態が長くなるという問題が起こる。品詞タグ付けで評価している [14] では、文の長さによって処理量が予測できるため、各プロセスの処理量を均一にするという工夫ができたが、ゲーム木探索では探索量の予測が難しいため、このような工夫はできない。

処理量のばらつきに起因するプロセスのアイドル時間を短くするため、ミニバッチの大きさを訓練例の個数にする方法とは別に、時間にする方法も試みた。ミニバッチの大きさを時間 T_B にする方法として次の二つが考えられる。

(A) 訓練例を処理するたびに時間をチェックし、時間 T_B が経過していたら更新ベクトルを通信する。

(B) 時間 T_B が経過したら、訓練例の処理を強制的に打ち切り、更新ベクトルを通信する。打ちきられた訓練例の処理は、更新ベクトルの通信後に最初から改めて実行される。

B の方法では、処理時間が T_B を越える訓練例の扱いが問題になる。これについても二つの方法が考えられる。

(a) そのような訓練例は捨てる。

(b) そのような訓練例には時間制限を適用しない。

本研究では、アイドル時間を極力抑えることを目的として、上記の B と a を組み合わせた方法を用いることにした。前回のミニバッチで、一つも訓練例を処理できなかったプロセスは、その訓練例を処理するのを諦め、次の訓練例を処理する。

5.　評価

プロ棋士の棋譜を用いて逐次の学習と並列の学習を行い比較した。学習精度の指標としては、コンピュータプレイヤが選んだ指し手が、プロ棋士の指し手と同じである一致率を用いた。

まず全体の実験に共通する設定を 5.1 節で述べた上で、ミニバッチのサイズを訓練例の個数と時間にした場合の評価をそれぞれ 5.2 節と 5.3 節で述べる。

5.1　実験設定

学習にはプロ棋士の 30,000 棋譜を用い、一致率を調べるためのテスト用の棋譜としては、学習に用いていない 1,000 棋譜を用いた。学習を始める前には 30,000 棋譜分の訓練例をランダムに並び替える。訓練例をすべて一回ずつ処理したら、またランダムに並び替えてから次のイテレーションを開始する。並列の学習では、訓練データを分割することはしなかった。つまり、全プロセスが全ての訓練データを持つ。ただし、各プロセスが訓練例をランダムに並び替えるため、訓練例を処理する順序は異なる。

比較学習で、1 手進めたあとの局面を探索する深さは 5 とした。つまり、全体の探索深さは 6 である。一致率を調べるときの探索深さも 6 とした。

用いた実験環境は以下の通りである。用いた計算ノード数は全部で 64 ノードであり、総コア数は 512 である。

- CPU : Xeon E5530 2.40GHz
  - 1 ノードあたり 4 コア ×2 ソケット
  - 8M cache, 2.40GHz, 5.86 GT/s QPI
- 1 ノードあたりのメモリ : 24GB
- ネットワーク : 10Gbps Ethernet
- gcc: version 4.4.5
- MPI: MPICH2 version 1.4.1p1

各計算ノードのコア数は 8 であるが、1 プロセスずつ実

行するようにした。これは、激指の学習はマルチスレッドを用いて並列化されているためである。激指の比較学習では、各スレッドが、訓練局面から一手進めた子ノードの探索を並列に実行する。以降、逐次の学習という用語は、このスレッド並列の学習を指し、並列の学習という用語は分散並列の学習を指すことにする。つまり、並列の学習では、一つの計算ノードが図 2 の 1 プロセスに相当する。提案手法では、プロセス数が多くなるほど通信量も多くなるため、通信量を抑えるためにも、スレッド並列が可能なところはスレッド並列にするべきだと判断した。

5.2 ミニバッチの大きさを訓練例の個数にした場合

まず、並列パーセプトロンの将棋評価関数に適用したときの評価を、ミニバッチのサイズを訓練例の個数として行った。ミニバッチのサイズとなる訓練例の数 B は 5、10、50 で実験した。B は大きいほど情報の更新が遅れてしまうが、小さすぎると計算に比較して通信時間が長くなってしまう。今回は、計算時間と通信時間のバランスがよいと考えられる $B = 10$ の場合を中心に実験を行った。

逐次の学習は訓練例をランダムに並び替えるための乱数の種を変更することにより、10 回の試行を行った。並列の学習の試行回数ついては、中心に調べる $B = 10$ の場合を 10 回とし、$B = 5$ の場合は 3 回、$B = 50$ の場合は 2 回とした。

逐次の学習では訓練例を 128,000 個処理するたびに、それまでの重みベクトルの平均ベクトルを求めてデータ点とした。並列の学習では、各プロセスが訓練例を 2,000 個処理するたびに、同様に平均ベクトルを求めた。つまり、一つのデータ点をプロットするための訓練例の総数を同じにした。データ点の間隔は、時間にすると、逐次の学習で 10 分程度、$B = 10$ のときの並列の学習で 25 秒程度である。

学習の進行度の指標としては、1 プロセスあたりの学習局面数と、学習時間の 2 つを用いる。ただし、学習時間には、学習開始時の棋譜の読み込み時間は含まない。また、それまでの重みベクトルの平均ベクトルを求める時間も含まない[*1]。

はじめに、逐次の学習の 10 回の試行結果を**図 4** に示す。また並列の学習の試行結果を $B = 5$、$B = 10$、$B = 50$ の場合についてそれぞれ**図 5**、**図 6**、**図 7** に示す。横軸は時間（単位は時間）、縦軸は棋譜の指し手との一致率である。図中の一つの線は一回の試行の学習の進行を示す。これらのグラフから以下のことが読み取れる。

- $B = 5$ と $B = 10$ の並列の学習では時間をかけすぎると、一致率が大きく下がっていく。逐次の学習でもその傾向は見られるが、並列の学習よりはその傾向は弱い。

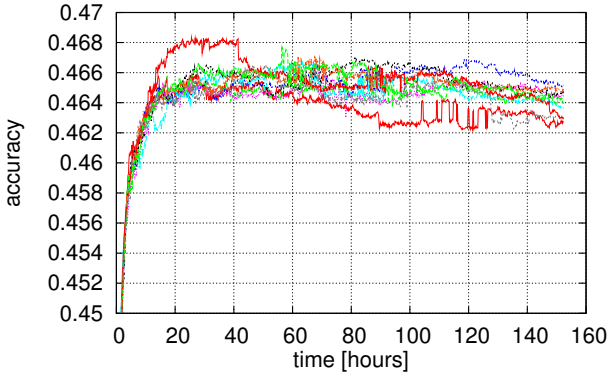

図 4 逐次の学習での学習時間と一致率

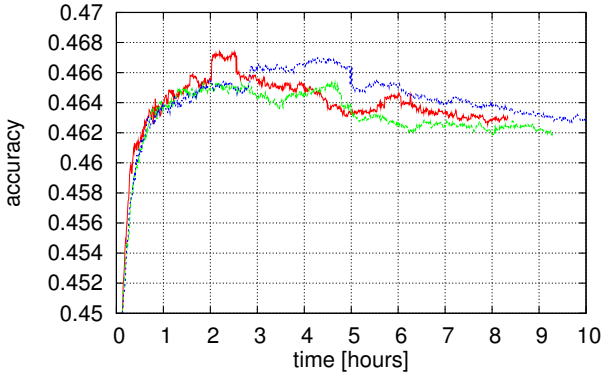

図 5 並列の学習での学習時間と一致率（64 プロセス、$B = 5$）

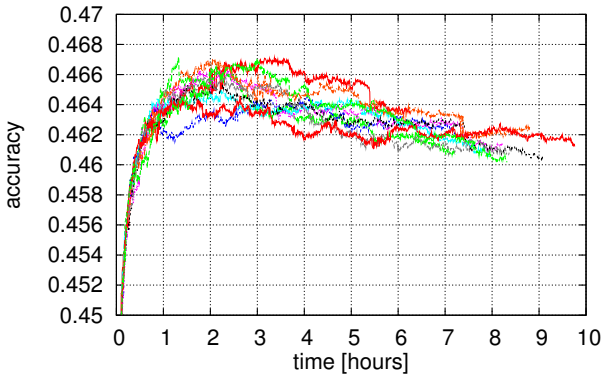

図 6 並列の学習での学習時間と一致率（64 プロセス、$B = 10$）

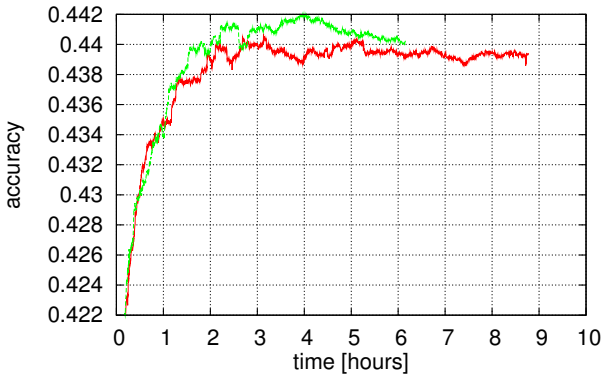

図 7 並列の学習での学習時間と一致率（64 プロセス、$B = 50$）

[*1] 平均ベクトルを求める操作は 1 プロセスしか実行しないため、他のプロセスは次の集合通信まで（つまり 1 バッチ）の処理を先に実行してしまうことになり、不当な評価になってしまっている。ただし、他のプロセスの処理も次のバッチまでしか進まず、平均ベクトルを求める操作も、例えば $B = 10$ の場合は 200 バッチごとに一回しか起こらないため、この影響は小さいと考えられる。

表 1 最大の一致率とそれを実現するための 1 プロセスあたりの学習局面数と学習時間

	逐次	$B = 5$	$B = 10$	$B = 50$
最大一致率 [%]	46.694	46.658	46.608	44.138
学習局面数 [$\times 10^6$]	51.302	0.813	0.733	2.244
学習時間 [hours]	63.376	3.692	2.461	3.542

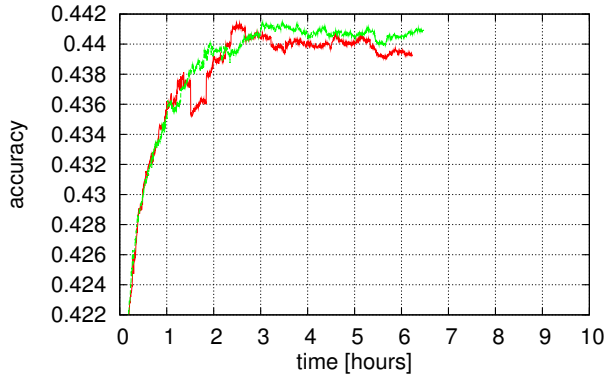

図 8 並列の学習での学習時間と一致率（64 プロセス、$B = 50$、ローカルな重みベクトルの更新なし）

- 逐次の学習の方が、並列の学習より、一致率が高い傾向がある。特に、$B = 50$ の並列の学習では明らかに一致率が低い。
- 一致率のピークに到達する時間は並列の学習の方が短い。

並列の学習では学習が進むと一致率が下がる傾向があった。逐次でもその傾向は見られたが、程度は小さい。一致率が下がるのは過学習のためであると考えられるが、並列の学習では更新ベクトルを足し合わせている影響が過学習を助長していた可能性がある。しかし、実際の学習では、一致率が下がりはじめた時点で学習を打ちきることができるため、あまり問題にはならない[*2]。

各学習において、どの程度の一致率を実現できたかを比較する。各学習において、一致率の最大値と、それを実現したときの 1 プロセスあたりの学習局面数と学習時間を調べた。逐次の学習と並列の学習のそれぞれで、これらの相加平均を求めたものを**表 1** に示す。並列の学習は逐次の学習より一致率が低く、B が大きいほどその程度も大きいことが分かる。一致率が下がるのは、重みベクトルの更新を 1 回ずつではなく、まとめて行っていることに起因していると推測される。

今回の実装では、ミニバッチの処理のときに、訓練例を処理するごとにローカルな重みベクトルも更新しており、これが一致率を下げている原因という可能性もある。なぜなら、ローカルな重みベクトルを更新することにより、学習の収束の保証がなくなるためである。この影響は B が大きいほど大きいと考えられる。そこで、$B = 50$ の場合については、ローカルな重みベクトルの更新を行わない場合についても実験を行った。試行回数は 2 回とした。結果を**図 8** に示す。一致率は図 7 と同様に低く、ローカルな重みベクトルの更新は学習性能に影響がないことが分かった。

[*2] 平均ベクトルを求めてファイルに書き出す操作にかかる時間は数秒である。今回の実験ではデータ点を多くとったが、図 6 などを見る限り、実用上はここまで多くのデータ点をとる必要はない。平均ベクトルがあれば、一致率を求める操作は、他の計算ノードで実行することができるため、この時間は問題にならない。

表 2 一致率 46.4496%を実現するための 1 プロセスあたりの学習局面数と学習時間

	逐次	$B = 5$	$B = 10$
学習局面数 [$\times 10^6$]	11.059	0.251	0.472
（台数効果）		44.1	23.4
学習時間 [hours]	14.284	1.235	1.607
（台数効果）		11.6	8.89

次に、並列化による学習の高速化の程度を見ていく。表 1 では、最大の一致率を実現するための学習局面数は少なくなり、学習時間は短縮されていることが分かる。しかし、そもそもの一致率が異なるため、このまま単純に比較することはできない。そこで、基準となる一致率を定め、この一致率を実現するための学習局面数や学習時間で比較を行う。基準となる一致率としては、$B = 10$ の学習の試行のうち、最小のピークの一致率を用いた。具体的には、$B = 10$ の並列の学習では、一致率が最大でも 46.4496%までしか到達しない試行があったため、これを基準となる一致率とした。

基準となる一致率を用いて並列効果を見る。各試行で一致率 46.4496%を最初に実現したときの、1 プロセスあたりの学習局面数と学習時間を求めた。逐次の学習と並列の学習それぞれで、これらの相加平均を求めたものを**表 2** に示す。$B = 50$ では基準の一致率を達成できなかったため、表 2 には含めていない。台数効果は、1 プロセスあたりの学習局面数と学習時間が、並列化により逐次の学習の何分の 1 になったかを示す。B が小さいほど、学習局面数や学習時間が少なくて済んでいることが分かる。特に学習局面数は B による差が顕著である。なお、$B = 5$ の場合は試行回数が少ないが、学習局面数については 3 回全ての試行において、$B = 10$ の全ての試行の中で最小の学習局面数以下だった。学習時間については、B が小さいほど以下で述べるアイドル時間や通信時間の影響が大きくなるため、学習局面に比べて台数効果の差は小さくなるが、$B = 5$ の場合で、11.6 倍の台数効果が得られた。

線形な台数効果が得られていない理由を述べる。まず、学習局面数が 1/64 になっていないのは、B が少ないほど基準となる一致率を実現する学習局面数が少なくて済んでいることから、重みベクトルの更新が訓練例 1 つを処理するたびに行っていないことに起因すると推測される。学習時間の台数効果が学習局面数の台数効果より小さいのは、計算時間以外に、各プロセスの重みベクトルの平均を求め

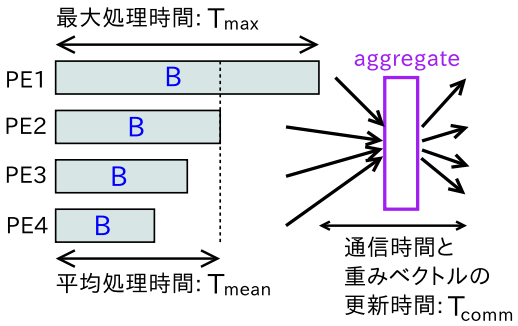

図 9　B 個の訓練例を処理する時間と重みベクトルの通信時間

表 3　64 プロセスが B 個の訓練例を処理する時間と、更新ベクトルの通信時間と非零要素数

	T_{mean} [ミリ秒]	$T_{\max}$ [ミリ秒]	T_{comm} [ミリ秒]	N_{comm}
$B = 5$	24.4	74.5	16.9	80,179
$B = 10$	47.0	108.9	25.2	118,068
$B = 50$	184.7	265.6	47.8	183,806

るために行う通信にかかる時間と、B 個の訓練例を処理し終えた後に、他のプロセスが終わるのを待っているアイドル時間が存在するからである。

重みベクトルの通信時間とアイドル時間の大きさを定量的に議論する。並列の学習では、**図 9** のように B 個の訓練例を処理するためにかかる時間がばらつくことによるアイドル時間と、通信時間が問題となる。これらの時間を調べるための実験を、これまでに述べた実験とは別に実行した。学習が進むほど重みベクトルが更新される回数が減るため、訓練例の処理時間や更新ベクトルの通信時間は短くなる。ここでは、一致率がまだ安定しない学習前半に着目し、1 プロセスあたり 128,000 局面を処理するまでの各ミニバッチにかかる平均処理時間を求めた。結果を**表 3** に示す。T_{mean} は B 個の訓練例を処理する時間、$T_{\max}$ は 64 プロセスが B 個の訓練例を処理したときの最大処理時間、T_{comm} は更新ベクトルの通信時間とそれを用いて重みベクトルを更新する時間を合わせたものである。また、通信後の更新ベクトルの非零要素数 N_{comm} も参考のために載せた。

B 個の訓練例そのものの平均処理時間に対する、通信時間も含めた実際の処理時間の割合に注目する。例えば、$B = 5$ の場合では、訓練例そのものの平均処理時間は 24.4 ミリ秒であるが、通信時間も含めた実際の処理時間は 91.4 ミリ秒であり、平均処理時間の 3.75 倍である。表 2 では学習局面数に対する台数効果が 44.1 倍だったので、学習時間に対する台数効果は $44.1/3.75 = 11.8$ 倍と予測される。これは実験結果の 11.6 倍とほぼ一致している。$B = 10$ の場合も、同様にして求めた値と実験結果がおおむね一致する。このことから、速度向上を抑制しているのは、アイドル時間と通信時間が主な要因だと結論づけることができる。また、通信時間より、アイドル時間の方がより大きな問題であることも分かる。

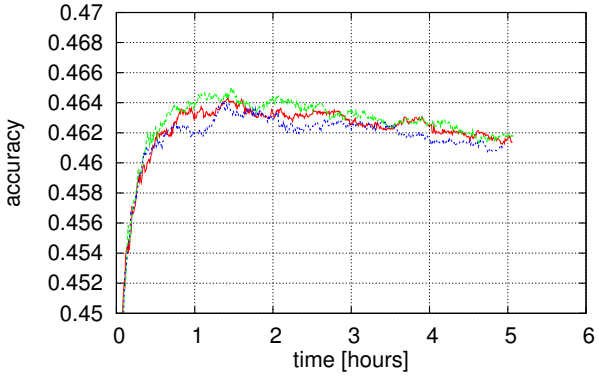

図 10　並列の学習での学習時間と一致率（64 プロセス、$T_B = 20$）

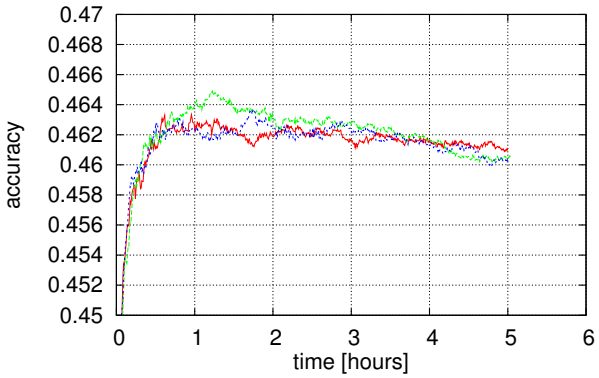

図 11　並列の学習での学習時間と一致率（64 プロセス、$T_B = 50$）

表 4　最大の一致率とそれを実現するための学習時間（ミニバッチの大きさが時間の場合）

	$T_B = 20$	$T_B = 50$
最大一致率 [%]	46.4459	46.3983
学習時間 [hours]	1.414	1.311

5.3　ミニバッチの大きさを時間にした場合

ミニバッチの大きさを時間にした場合の評価も行った。ミニバッチ一つの時間 T_B は 20 ミリ秒と 50 ミリ秒に設定し、学習の試行回数は 3 回ずつとした。プロセスが処理する訓練例の数は一様ではないため、データ点を得るために平均ベクトルを求めるまでの間隔を時間にし、30 秒とした。前節と同様に、学習時間には棋譜の読み込み時間と、データ点を得るための時間は含めない。

学習の進行の様子を $T_B = 20$ と $T_B = 50$ について、**図 10** と**図 11** にそれぞれ示す。また、各試行で最大の一致率とそれを実現した学習時間の相加平均を**表 4** に示す。ミニバッチの大きさを訓練例の個数にした場合より、一致率が低い傾向にあることが分かる。$T_B = 20$ と $T_B = 50$ のそれぞれの試行のうち、2 回は一致率が基準とした 46.4496%に到達していなかった。また、図 10 や図 11 を図 5 や図 6 と比較しても、ミニバッチの大きさを時間にすることによっ

て、学習が速くならなかったことも分かる。

このような実験結果になった理由の一つとしては、処理時間が T_B 以上かかる訓練例を捨てていることが挙げられる。調べたところ、1 プロセスあたり平均 128,000 局面を処理した時点で、学習局面のうち、$T_B = 20$ の場合には 2.41%、$T_B = 50$ の場合には 0.112%の局面が捨てられていた。この割合は大きくはないかもしれないが、捨てられた局面が学習には重要な局面だった可能性がある。

学習速度に関しては、同じ学習局面数を処理する時間は短くなっていることが分かった。具体的には、1 プロセスあたり平均 128,000 局面を学習するための時間をおおまかに比較したところ、ミニバッチの大きさを訓練例の個数にした場合では、$B = 5$ の場合で 2,300 秒、$B = 10$ の場合で 1,700 秒だったのに対し、ミニバッチの大きさを時間にした場合では、$T_B = 20$ の場合で 1,700 秒、$T_B = 50$ の場合で 1,100 秒であった[*3]。実際には打ち切られて無駄になった処理もあることから、この時間も理想的には短くなっていない。また、一致率が上昇する速度自体は改善されなかったことから、多くの学習局面を処理できるようになっても、訓練例を捨てている悪影響で相殺されたと考えられる。

6. おわりに

将棋プログラムの評価関数のパラメータ調整に、ミニバッチを用いたパーセプトロンの並列化手法を適用した。評価では、並列化により学習時間を 1/11.6 に短縮できることを示した。速度向上を妨げている要因としては、プロセスで処理時間がばらつくことに起因するアイドル時間と通信時間があり、これらの大きさを定量的に議論し、通信時間よりアイドル時間の方がより大きな問題であることを示した。

アイドル時間を減らすために、ミニバッチの大きさを時間にすることも提案した。具体的には、時間が来たら訓練例の処理を打ち切り、また、処理に時間がかかる訓練例は無視するようにした。しかし、一致率はミニバッチの大きさを訓練例の個数にしたときより下がる傾向にあることが分かった。これにより多くの学習局面を処理できるようにはなったが、一致率が上昇する速度は大きくならなかった。

今後の課題としては、通信時間やアイドル時間を小さくすることができるような適切な手法を考えることが挙げられる。例えば、重みベクトルの更新を非同期にすることが考えられる。しかし、この場合、通信効率の良い集合通信をそのまま使うことができず、分散環境では速度向上が得られないと考えられるため、よりよい方法を模索する必要がある。

[*3] これらの値は細かい時間を測定するための処理を行った場合であり、実際にはもう少し速い。

参考文献

[1] Agarwal, A. and Duchi, J. C.: Distributed Delayed Stochastic Optimization, *NIPS '11* (2011).
[2] Collins, M.: Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms, *EMNLP '02*, pp. 1–8 (2002).
[3] Dean, J., Corrado, G. S., Monga, R., Chen, K., Matthieu Devin, Q. V. L., Mao, M., Marc' AurelioRanzato, Senior, A. W., Tucker, P., Yang, K., Ng, A. Y. : Large Scale Distributed Deep Networks, *NIPS '12* (2012).
[4] Gimpel, K., Das, D. and Smith, N. A.: Distributed asynchronous online learning for natural language processing, *CoNLL '10*, pp. 213–222 (2010).
[5] Hall, K. B., Gilpin, S. and Mann, G.: MapReduce/Bigtable for Distributed Optimization, *NIPS LCCC Workshop* (2010).
[6] Hoki, K. and Kaneko, T.: The Global Landscape of Objective Functions for the Optimization of Shogi Piece Values with a Game-Tree Search, *ACG '11*, Lecture Notes in Computer Science, Vol. 7168, pp. 184–195 (2012).
[7] Jyothi, P., Johnson, L., Chelba, C. and Strope, B.: Large-scale discriminative language model reranking for voice-search, *NAACL Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, pp. 41–49 (2012).
[8] Kaneko, T. and Hoki, K.: Analysis of Evaluation-Function Learning by Comparison of Sibling Nodes, *ACG '11*, Lecture Notes in Computer Science, Vol. 7168, pp. 158–169 (2012).
[9] McDonald, R., Hall, K. and Mann, G.: Distributed training strategies for the structured perceptron, *NAACL '10*, pp. 456–464 (2010).
[10] Pacheco, P. S.: *Parallel programming with MPI*, Morgan Kaufmann (1996).
[11] Tesauro, G.: Comparison training of chess evaluation functions, *Machines that learn to play games*, Nova Science Publishers, Inc., pp. 117–130 (2001).
[12] Ura, A., Miwa, M., Tsuruoka, Y. and Chikayama, T.: Comparison Training of Shogi Evaluation Functions with Self-Generated Training Positions and Moves, *CG '13* (2013).
[13] Yuan, G.-X., Ho, C.-H. and Lin, C.-J.: Recent Advances of Large-Scale Linear Classification, *Proceedings of the IEEE*, Vol. 100, No. 9, pp. 2584–2603 (2012).
[14] Zhao, K. and Huang, L.: Minibatch and Parallelization for Online Large Margin Structured Learning, *NAACL '13* (2013).
[15] 川上裕生， 浦晃， 三輪誠，鶴岡慶雅， 近山隆：将棋の評価関数の学習に有用な局面の自動選択，第 18 回ゲームプログラミングワークショップ，pp. 66–72 (2013).
[16] 保木邦仁：局面評価の学習を目指した探索結果の最適制御，第 11 回ゲームプログラミングワークショップ，pp. 78–83 (2006).