情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2010-HPC-126 No.26
2010/8/4

分散共有メモリ環境におけるUCTの並列実行

吉本晴洋[†1,†2] 田浦健次朗[†1,†2]

モンテカルロ法を用いたゲーム木探索であるUCTは現在強い囲碁プレイヤを作るための最も一般的な手法である．本研究では分散共有メモリ処理系であるDMI上でUCTを利用した囲碁プログラムを並列化した．

Parallelization of UCT on Distributed Shared Memory System

Haruhiro Yoshimoto[†1,†2] and Kenjiro Taura[†1,†2]

UCT, a game tree search algorithm using Monte Carlo method, is currently a most popular algorithm to make strong Go game AI's. We parallelize UCT algorithm on DMI, one of Distributed Shared Memory system.

1. はじめに

将棋やチェスをはじめとする完全情報ゲームでは，先読みを行うことは，良い手をプレイする上で重要な要素である．α-β 法をはじめとするゲーム木探索では，局面の優劣を数値化する評価関数を利用して，先読みを行い，次の一手を決定している．ゲーム木探索は，完全情報ゲームをプレイするプログラムの多くに適用され，チェスでは人間の世界チャンピオンに勝つ[2]など目覚ましい成果を挙げてきた．将棋やチェスなどのプログラムの強さは，ゲーム木探索に改良を重ねることによって格段に進歩してきたが，囲碁ソフトウェアは人間のトッププレイヤーにまだまだ及ばない[7]．その最大の要因は，ゲーム木探索で利用できるような正確な評価関数の作成が，囲碁では難しいことにある．将棋やチェスなどでは，駒の価値等の基準を利用することによって，正確な評価関数を作りやすい．ところが，囲碁では，最終的には「地」と呼ばれる自石で囲まれた空点の多いプレイヤーの勝ちであるが，この地になりそうな部分を正確に判定できる評価関数の構築は，現状では困難である．

この評価関数の問題を解決するために，コンピュータ囲碁の分野では，局面の評価にモンテカルロ法を利用するモンテカルロ碁がの研究が盛んである．現在最も有効なモンテカルロ碁は，UCTアルゴリズム[6]に基づくものである．例えば，現在最強のプログラムであるZenやMogo[4]などの強豪プログラムは，UCTに様々なヒューリスティックを搭載している．

UCTはモンテカルロ法を利用したゲーム探索であり，シミュレーション数を増やすと強くなることが知られている．UCTを並列化すれば，それを複数のプロレッサ上で動かすことによって単位時間あたりのシミュレーション数を増やすことができる．本研究は分散メモリ上でUCTを並列化し，そのことにより強い囲碁プレイヤを作ることを目的とする．

2. UCTアルゴリズム

本章では，KoscisらのUCTアルゴリズム[6]を囲碁に利用した場合に限定して，説明を行う．UCTは，節点の評価にシミュレーションを利用した最良優先型の探索手法である．UCTにおける，ある節点のシミュレーションとは，その節点から終端節点まで，両プレイヤーがランダムにプレイし，そのランダムプレイの勝ち負けの結果 (勝ちは1で，負けは0) を返すことである．

UCTでは，各プレイヤーはUCB値に基づいて子節点の選択を行う．指し手 i のUCB値とは，指し手 i で到達する子節点 c_i と c_i の兄弟節点を利用して，以下のように定義される．

$$\mathbf{get\_ucb}(i) \equiv \frac{w_i}{n_i} + \sqrt{\frac{2\ln s}{n_i}} \quad (1)$$

ここで，w_i は c_i のシミュレーションの勝ち数，n_i は c_i のシミュレーション回数，s は c_i および c_i の兄弟節点のシミュレーション回数の総和である．UCB値の計算は，基本的に勝率に基づいているので，UCB値が大きければ，勝ちやすい指し手であると考えられる．ただし，訪問回数の節点が少ない節点の勝率は信頼できないので，探索によって，より正確な勝率を求める必要がある．UCB値では，訪問回数の反比例値も考慮に入れることで，勝率の高そうな指し手とそうでない指し手の探索の頻度をうまく制御することができる．UCTでは，有望そうな指し手の訪問頻度は，そうでない指し手よりも指数のオーダーで大きくなる．

†1 東京大学
University of Tokyo
†2 情報理工学系研究科
Information Science and Technology

UCT では，根節点よりシミュレーションが行われたことのない先端節点に至るまで，UCB 値の最も大きな指し手[⋆1]を各プレイヤーは選択する．次に，選択した先端節点において，シミュレーションを行い，その節点を評価し，探索木にその節点を追加する．この評価値を利用して，根節点よりその先端節点に至る経路を逆向きに手繰り，経路上の UCB 値を更新しながら，根節点まで戻る．UCT では，この木探索とシミュレーションの過程を何度も繰り返して，最も有効な次の一手を選ぶ．

3. 既存の UCT 並列化手法

本章では既存の UCT の並列化手法について述べる．

3.1 共有メモリと分散メモリ

対象となるアーキテクチャが共有メモリシステムなのか分散メモリシステムなのかということに応じて並列化の手法を変える必要がある．

アーチテクチャが共有メモリの場合はそのその実装は比較的簡単で，木探索部分は Mutex などのロックをかけて行ない，シミュレーション部分はそれぞれのプロセッサが別の乱数シードで行うだけでよい．シミュレーション部分は完全に独立に行なうことができるため Mutex などのロックを取る必要がなく，またシミュレーションにかかる時間は木探索にかかる時間より圧倒的に大きいために，この単純な実装で高い並列化効率を得られる．実際，Mogo では $\alpha \stackrel{\text{def}}{=} (\text{木探索の時間}) \div (\text{シミュレーションにかかる時間}) \simeq 0.05$ である[3]．現在の強い囲碁プログラムのほとんどで共有メモリでの並列化は行なわれている．ただし，プロセス数が上記の $\frac{1}{\alpha}$ を越えると，シミュレーションにかかる時間より全プロセスの木探索の時間の合計の方が大きくなるので上記の手法での並列化効率は落ちる．

一方本研究が対象としているのは分散メモリシステムである．分散メモリシステムにおいては通信遅延が大きいため上記のような単純な実装は使えない．したがって次節で述べるような探索木の共有を行なう．

3.2 探索木の共有方法

分散メモリ上の UCT における探索木の共有方法は以下の 3 種類に分類される．

一つ目は Root 並列化と呼ばれる方式で，各ノードが木の情報を共有せずに探索し，最後に集計するというものである．これは実装が簡単かつ探索途中での通信が一切ないので同期を取らずに済むため単位時間あたりのシミュレーションのプレイアウト数は多くなる．しかし，探索木を共有しないため無駄な探索も多く行ってしまい，結果として下記の探索木を共有する方式より弱い傾向がある[10]．

二つ目は Leaf 並列化と呼ばれる方式で，1 つのノード（クライアント）が木探索を行い，残りのノード（サーバ）がシミュレーションを行うというものである[8]．クライアントはサーバに対してどの節点を探索するかをいう情報を投げ，受け取ったサーバがその節点のシミュレーションを行なう．これは簡易に実装できるという利点がある一方，同期オーバーヘッドが大きいという欠点もある[10]．また，サーバ数が多くなると負荷がクライアントに集中してしまい性能が落ちる恐れがある．

最後は Tree 並列化と呼ばれる方式で，これは全てのノードが探索木を共有するというものである．もし探索木を完全に共有するのであれば一回のシミュレーションごとにその結果をブロードキャストする必要があるが，それではオーバッヘッドが大き過ぎるため，数秒ごとに探索木の主要な部分だけを同期するという手法が考えられている[3]．一般的には探索木を共有し，頻繁に同期すればするほど無駄な探索が減り，その結果強くなる．しかし，頻繁に同期させるとオーバーヘッドが大きくなる．この問題を如何に解決するかが Tree 並列化の性能を引き出す上でのポイントとなる．

本研究では Tree 並列化を行なう．その理由としては DMS 処理系である DMI[9] を使用すればノード上の各メモリを一つの仮想的な共有メモリとして扱えるため，Tree 並列化が自然に記述できるからである．

4. 本研究の実装

4.1 概　　要

本研究では DMI[9] を使って分散メモリ上での Tree 並列化を行う．

4.2 使用するライブラリ

4.2.1 Distributed Memory Interface

DMI(Distributed Memory Interface) は高性能並列計算を簡単に記述できるようにすることを目指した並列分散プログラミング処理系であり，以下のような特徴を持っている．

- プロセスの動的な参加／脱退が可能
- 共有分散メモリ型の処理系である
- データの所在を明示的かつ細粒度に指定することで並列計算の性能を最適化することが可能

本研究では計算途中での参加／脱退は考慮しないため最初の特徴は本研究にとって重要では

⋆1 シミュレーションの行われたことのない節点に至る指し手の UCB 値は，非常に大きい値となる．

ない．重要なのは二，三番目の特徴である．DMI は共有分散メモリ型の処理系でであるために Tree 並列化を自然に記述でき，かつ DMI を使用することで性能の最適化を施せることを期待できる．

以下に今後の説明で必要となる DMI の API を列挙する．

`DMI_mmap/DMI_munmap` グローバルメモリの確保/解放を行う

`DMI_read/DMI_write` グローバルメモリから読み込み/書き込み

`DMI_atomic/DMI_function` `DMI_function` にユーザ定義の read-modify-write を定義しておく．`DMI_atomic` を呼ぶことで `DMI_function` で定義された read-modify-write が実行される

`DMI_mutex_lock/DMI_mutex_unlock` Mutex のロック/アンロック

4.2.2 fuego

fuego[1] は GNU Lesser General Public License で提供されているオープンソースの C++ 用囲碁ライブラリである．本研究ではこのライブラリのうち碁盤クラス (GoBoard)，GTP エンジンクラス (GoGtpEngine)，シミュレーション用のランダムな合法手生成クラス (GoUctPlayoutPolicy) のみ使用し，UCT の木探索部分は独自に実装する．GoUctPlayoutPolicy はランダムな合法手を生成するためのクラスであるが，生成される合法手は完全にランダムなものではなく，自分の目を潰す手は生成しなかったり，パターンマッチに合致する手は高確率で生成したりすることで完全にランダムなシミュレーションより精度の良い結果が得られるようになっている．

4.3 速度向上のための工夫

UCT ではシミュレーションが終わるとその結果を，葉節点から根節点に向かってバックアプするのであるが，その際に `DMI_atomic` ではなく，Mutex しない `DMI_read` , `DMI_write` を使用する．

そのことにより，read/write の一貫性は保証されなくなるが，`DMI_atomic` によるロックがなくなるため高速化が期待できる．一貫性が破壊される操作順序の一例として (プロセス A の read) → (プロセス B の read) → (プロセス A の write) → (プロセス B の write) が考えられるが，これによって失われるのはプロセス A が行った 1 回分のシミュレーションの結果のみであるため，一貫性が破壊される頻度が少なければ全体の動作としては支障ない．以下，このとき失われるシミュレーションの情報をシミュレーション損失と呼ぶ．最悪の場合全てのシミュレーション情報が失われる可能性があるが，その可能性は非常に低い．実験の結果，実行速度は多少向上した．またシミュレーション損失は全体のわずか 3%程度であることが判明した．

5. 実　　験

5.1 実験環境

クラスタの各ノードの性能は以下の通り．

CPU	Core2Duo 2.13GHz
メモリ	4GB
NIC	GigE

使用するライブラリ，コンパイラなどのバージョンは以下の通り．

gcc/g++	4.3.2
boost	1.42.0
fuego	0.4.1
DMI	1.3.2.0

gcc/g++の最適化オプションは`-O3` を指定した．

5.2 実験結果および考察

高速化率を測定するために最も高速化率が高くなると思われる 19 路盤の初期局面から 30,000 シミュレーションの UCT を実行した．以下はその結果である．

ノード数	実行時間
1	139 秒
2	158 秒
4	165 秒
8	162 秒

このことから複数のノードを使用したにもかかわらず，逐次以上の性能は得られなかった．

性能が出なかった原因を推測するために，実行時の CPU の負荷を調べてみると 1 つの CPU のみが 100%使用され，残りの CPU は 15%〜25%程度しか稼働していなかった．つまり負荷分散がうまくできていない．このことから性能が出なかった理由として推測されるのは以下の要因である．

DMI 上で UCT を Tree 並列化した場合，共有メモリ上で UCT を並列化したものと同じアルゴリズムが使用できるため，プログラムの記述は容易である．しかし，それを実行しても良い性能を得られない．なぜなら全てのプロセスが同じ順番に節点を探索しようとするた

め，ロックにひっかかってしまうからである．したがって各プロセスがそれぞれ異なる節点を探索する必要がある．

それを実現する一つの方法としてどのプロセスがどの節点を探索しているかの情報を共有しておくというのがあるが，これは情報の同期のオーバーヘッドが高くなる，もう一つの方法はプロセスごとにどの節点を探索するか一定のルールにしたがって選ぶというものである．例えば盤面の右半分はプロセス A，左半分はプロセス B という具合である．適切なルールを設定すれば全てのプロセスに別の節点を探索させることができる．

6. 結論と今後の課題

DMI 上で UCT の Tree 並列化を実装した，共有メモリ上で UCT を並列化したものと同じアルゴリズムが使用したため性能は出なかった．今後はそれぞれのプロセスが異る節点を探索するように改良をほどこしたい．また，RAVE と呼ばれる高速に探索木を成長させる手法[5] を導入したい．

参 考 文 献

1) fuego. `http://fuego.sourceforge.net/`.
2) M.Campbell, A.JosephHoane Jr., and F.-h. Hsu. Deep Blue. *Artificial Intelligence*, Vol. 134, No. 1-2, pp. 57–83, 2002.
3) S.Gelly, J.B. Hoock, A.Rimmel, and O.Teytaud. The parallelization of monte-carlo planning. In *5th International Conference on Informatics in Control, Aotomation, and Robotics*, 2008.
4) S.Gelly, Y.Wang, R.Munos, and O.Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical report, INRIA, 2006. RR-6062.
5) Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pp. 273–280, New York, NY, USA, 2007. ACM.
6) L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, pp. 282–293, 2006.
7) M.Müller. Computer Go. *Artificial Intelligence*, Vol. 134, pp. 145–179, 2002.
8) 加藤英樹, 竹内郁雄. Parallel monte-carlo tree search with simulation servers. 第 13 回 ゲーム・プログラミングワークショップ, 2008.
9) 原健太朗, 田浦健次朗, 近山隆. Dmi：計算資源の動的な参加/脱退をサポートする大規模分散共有メモリインタフェース. 情報処理学会論文誌（プログラミング）, Vol.3, No.1, pp. 1–40, 2010.
10) 副島佑介, 岸本章宏, 渡辺治. モンテカルロ木探索の root 並列化とコンピュータ囲碁での有効性について. 第 14 回 ゲーム・プログラミングワークショップ, 2009.