並列MC/UCTアルゴリズムの実装

加　藤　英　樹[†1,†2]　竹　内　郁　雄[†1]

我々はコンピュータ囲碁を題材に，リカレントニューラルネットの応用を研究している．今回，テストベッドとして並列 UCT/MC アルゴリズムを用いた囲碁ソフトを作成し，実装法を検討した．探索木共有方式とクライアントサーバ方式を，タイプの異なる CPU を用いた 2 つのシステム，x86/自作 PC と Cell/Playstation 3 に実装し，実行速度を測定した．クライアントサーバ方式は，現在広く使われている探索木共有方式と比べて Cell では 3 倍高速に，x86 では 10%遅くなった．また，UCT アルゴリズムをモンテカルロシミュレーションの並列実行と組み合わせた時に起こる，アルゴリズムの挙動が変化するという問題の影響を GNU GO に対する勝率で評価した．実験した並列度 4 の場合，ELO レーティングに換算した勝率の低下は最大 35 ELO だったが，簡単な方法で最大 20 ELO に改善することができた．

A Study on Implementing Parallel MC/UCT Algorithm

HIDEKI KATO[†1,†2] and IKUO TAKEUCHI[†1]

We have developed a parallel MC/UCT computer Go program as a test bed for our research, applied recurrent neural networks. We measured the execution time of both commonly used shared-tree and client-server implementations on two different types of systems, Intel Core 2 Quad on a PC and Cell Broadband Engine on a SONY PLAYSTATION 3. The client-server implementation runs three times faster and 10% slower than shared-tree on the Playstation 3 and PC, respectively. Also, the effect of a well-known problem that parallelizing Monte Carlo simulations may make UCT algorithm behave differently was evaluated with the winning rates against GNU GO. Our experiments using four cores show that the winning rates decrease 35 ELO at most and can be improved to 20 ELO.

1. はじめに

モンテカルロ（Monte Carlo; MC）シミュレーションと UCT アルゴリズム[8)] を用いたコンピュータ囲碁対局プログラム（以下囲碁ソフト）は，これまで性能向上の最大の障壁だった静的評価関数が不要，かつ並列化に適しているという特徴を活かし，9 路ではこれまでの囲碁ソフトを超えてアマチュア有段の域に達している．

我々はコンピュータ囲碁を題材として，リカレントニューラルネット を用いた系列連想記憶の応用研究を進めており[7)]，今回テストベッドとして，並列 MC/UCT アルゴリズムを用いた囲碁ソフト（仮称 GGMC GO）を作成した．本報告ではこのソフトの並列実装に関して述べる．

GGMC GO の大きな特徴は，対称マルチコア（x86）による共有記憶マルチプロセッサシステムと，非対称マルチコア（Cell Broadband Engine[⋆1]; 以下 Cell）による分散記憶マルチプロセッサシステムの両プラットフォームに対応していることである．我々は，浮動小数点演算が高速な Cell がニューラルネットのシミュレーションに適していると考え，Playstation 3 Linux[⋆2]（以下 PS3）をプラットフォームに選んだ．しかし，PS3 の現在の開発環境は決して良いとは言えない．そこでプログラムを x86 でも動くようにして，論理的なデバッグは x86 の PC 上で行うことでこれをカバーすることにした．

本報告の構成は，本節が導入, 2 節で MC/UCT アルゴリズムの並列化に関連する研究を紹介し，3 節で本研究の目的と課題を説明し，4 節で探索木共有とクライアントサーバの両方式を比較・評価する．5 節で UCT アルゴリズムを並列化する時の問題について検討し，6 節でまとめと今後の課題を述べる．読者には MC/UCT アルゴリズムに関する知識を仮定する．

なお，探索木共有方式（4 節参照）による GGMC GO[⋆3] の PS3 版は Computer Olympiad 2007 の 9 路碁部門 8 位[⋆4]，x86 版は第 29 回 KGS コンピュータ碁トーナメント 3 位[⋆5] の成績を収めた．

2. 関連研究

S. Gelly らの MOGO に関する最初の報告[6)] に探索木共有方式による並列化に関する簡単な記述がある．また，T. Cazenave[2)] はネットワーク接続マルチ PC システム上に MPI を用いたマスタースレーブ型並列方式を実装してタスクの分割方法を検討している．

強い MC/UCT 囲碁ソフトは既にほとんど全て並列

†1 東京大学情報理工学系研究科創造情報学専攻
The Department of Creative Informatics, The Graduate School of Infomation Science and Technology, The University of Tokyo
†2 株式会社フィックスターズ
Fixstars Corporation

⋆1 http://cell.scei.co.jp/index_j.html

⋆2 http://www.playstation.com/ps3-openplatform/jp/, http://cell.fixstars.com/ps3linux/index.php/ 等

⋆3 http://www.gggo.jp でオープンソースとして公開

⋆4 http://www.grappa.univ-lille3.fr/icga/tournament.php?id=169

⋆5 http://www.weddslist.com/kgs/past/29/

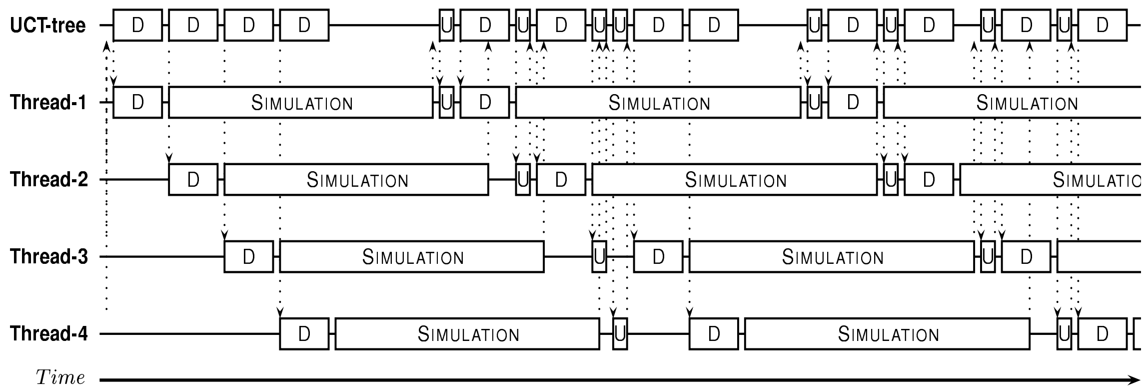

図 1 探索木共有方式のタイムチャートの例．4 スレッド時．左端: UCT-tree は共有されている探索木，Thread-1〜4 は各スレッド．箱の中: D は DESCENDTREE，Simulation は DOSIMULATION，U は UPDATETREE の各処理．探索木の時間軸上の箱は，いずれかのスレッドに占有されていることを示している．

Fig. 1 A time-chart for shared-tree implementation. 4 threads. “D”, “Simulation” and “U” mean DESCENDTREE, DOSIMULATION, UPDATETREE, respectively. The boxes on the line for UCT-tree show the tree is locked by a thread.

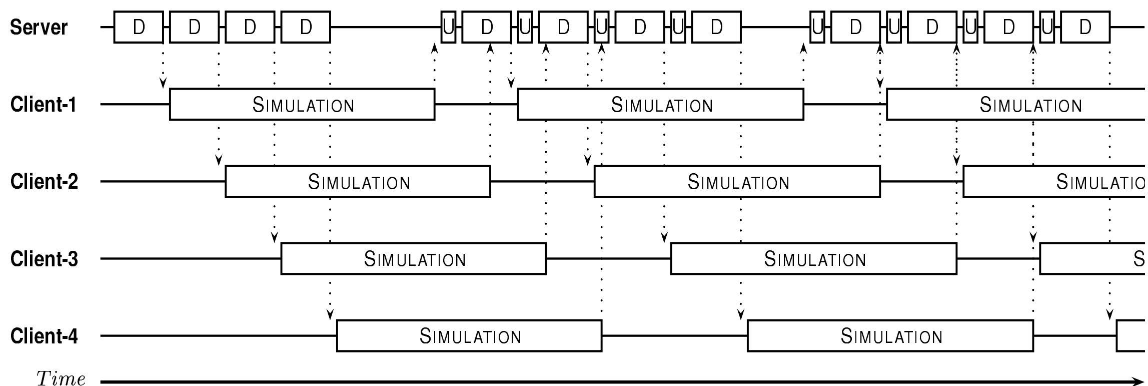

図 2 クライアントサーバ方式のタイムチャートの例．4 クライアント時．左端: Server はサーバ，Client-1〜4 は各クライアント．箱の中: D は DESCENDTREE，Simulation は DOSIMULATION，U は UPDATETREE の各処理．スレッドの切替えは省略．

Fig. 2 A time-chart for client-server implementation. 4 clients. “D”, “Simulation” and “U” mean DESCENDTREE, DOSIMULATION, UPDATETREE, respectively. Switching of threads are omitted.

化されている⋆1が，その実装に関する報告はこの 2 つの他にない．しかしこれは，並列 MC/UCT アルゴリズムの実装に報告する価値がないということではなく，囲碁ソフトの開発者がヒューリスティクスによる性能向上等の報告を優先しているためと思われる．

関連する報告として，D. Dailey[4] による，一手当たりのシミュレーション回数が 2 倍になると ELO⋆2 が 50〜100 増えるとの報告がある．我々の経験でも，CPU を 2 コアから 4 コアに変えただけで ELO が 70 ほど向上した（100 ELO がおよそ 1 子差に相当）．この事実からも，MC/UCT アルゴリズムの並列実装がモンテカルロ囲碁ソフトの性能向上のために重要なテーマであることは明らかだろう．

3. 目的と課題

本報告の目的は，MC/UCT アルゴリズムの並列化の実装法，および並列化に伴う課題の検討である．

モンテカルロ囲碁ソフトは，ある局面から終局まで多数の乱数を用いたシミュレーションを行い，その局面の最終的なスコア（勝敗）を推定する．各シミュレーションは独立に実行可能なので並列化は容易である．しかし UCT アルゴリズムは逐次実行を前提としており，文献 6) でも並列化することでその挙動が変化する可能性が指摘されている．この問題は 5 節で検討する．

並列化の方式に関しても，単一スレッド用のプログラムをそのままマルチスレッド化する方法が広く使われている．この時，探索木は全スレッドで共有し，あるスレッドが探索木にアクセスする時は，適当な排他機構で他のスレッドのアクセスを禁止する．しかしこの方式は，例えば Cell の様な，共有記憶でないアーキテクチャには適していない可能性が高く，またネットワーク接続マルチ PC システムには適用できないという問題がある．2 節で述べた文献 2) はネットワーク接続マルチ PC システム用の実装だが，事前に全体で一台の仮想マシンを構成するため，動的に PC を追加あるいは削除することができない．

クライアントサーバ方式にはこの様な問題はなく，また適用範囲も広いが，実装の報告はまだない．我々は x86 と Cell という 2 つの異なるタイプのプロセッサに探索木共有とクライアントサーバの両方式を実装

⋆1 例えば本年 6 月の Computer Olympiad Amsterdam では，並列化されてないのは GO INTELLECT だけだった．

⋆2 例えば http://en.wikipedia.org/wiki/Elo_rating

```
GenerateMove(position)
 1  root ← CreateNode(position)
 2  repeat
 3        Lock(tree)
 4        (leaf, path) ← DescendTree(root)
 5        position ← leaf.position
 6        move ← SelectMoveToX(position)
 7        move.flag ← true                    ※フラグ法
 8        move.fpu ← 0.5 × move.fpu           ※ fpu 法
 9        Unlock(tree)
10        position ← PlayMove(move, position)
11        node ← CreateNode(position)
12        move.node ← node
13        score ← DoSimulation(position)
14        Lock(tree)
15        UpdateTree(score, path)
16        move.flag ← false                   ※フラグ法
17        Unlock(tree)
18    until some condition
19  return SelectMoveToPlay(root)

DescendTree(node)
 1  path ← EMPTY
 2  while node.visits > 0
 3      do i ← argmax_i node.moves[i].fpu
 4         path ← path + (node, i)
 5         node ← node.moves[i].node
 6  return (node, path)

DoSimulation(position)
 1  while position.gameEnd = false
 2      do move ← SelectRandomMove(position)
 3         position ← PlayMove(move, position)
 4  return CountFinalScore(position)
```

図 3 探索木共有方式の擬似コード
Fig. 3 Pseudo code of shared-tree implementaion.

し，その実行速度を比較した．詳細は 4 節で述べる．

プロセッサ数のスケーラビリティの観点からは，ロックのオーバーヘッドや公平性の問題がある．例えば探索木共有方式の場合，探索木全体をまとめてロックする方法は簡単で必要な記憶量も少ないが，スレッド数が増えた時に待ち時間が長くなるという問題があり，各ノードを個別にロックすることで改善される可能性がある．

またロックの種類に関しても，spinlock は mutex より必要な記憶量やオーバーヘッドが小さいが，特に NUMA（non-uniform memory access）システムでプロセッサの利用率が不公平になる場合があり，fair-lock を使うことで改善されるとの報告[3] がある．しかし，クライアントサーバ方式では探索木の排他制御が不要なので，ロックに関してはこれ以上触れない．

4. 実　　装

UCT アルゴリズムを，モンテカルロシミュレーションの並列実行に対応させる最も簡明な方法は，単一スレッド用のプログラムをそのままマルチスレッド化する方法（探索木共有方式と呼ぶ）だろう．この場合探索木は全スレッドで共有し，アクセスする時は mutex や spinlock 等の排他機構を使って他のスレッドからのアクセスを禁止する（**図 1**）．この方式は現在広く使われており，我々も GGMC Go ver. 1 にはこの方

```
GenerateMove(position)
 1  root ← CreateNode(position)
 2  id ← −1
 3  repeat
 4        (leaf, path) ← DescendTree(root)
 5        temp ← FindNextFreeClient(id)
 6        if temp >= 0
 7          then id ← temp
 8               client ← clients[id]
 9               position ← leaf.position
10               move ← SelectMoveToX(position)
11               position ← PlayMove(move, position)
12               move.node ← CreateNode(position)
13               client.position ← position
14               client.path ← path
15               message.position ← position
16               SendToClient(message) ※
17               client.status ← busy
18          else message ← RecieveFromClient() ※
19               id ← message.id
20               client ← clients[id]
21               client.status ← free
22               score ← message.score
23               UpdateTree(score, client.path)
24    until some condition
25  return SelectMoveToPlay(root)
26
27  なお，
28      SendToClient(message) ※
29      message ← RecieveFromClient() ※
30  の 2 行は CPU に応じて
31  x86:
32      PushQueue(message, client.queue)
33      message ← PopQueue(globalQueue)
34  Cell:
35      WriteToInMbox(message, client)
36      message ← PollOutMbox(client)
37  となる．
```

図 4 クライアントサーバ方式の擬似コード（サーバ）
Fig. 4 Pseudo code of client-server implementaion (server side).

式を用いたが，PS3 での実行速度が 1.2 GHz の x86（1 コア）と同程度と，Cell のハードウェア性能から期待される速度と比べて非常に悪かった．そこでネットワーク接続マルチ PC システムへの拡張も考慮し，今回新たにクライアントサーバ方式（**図 2**）を実装した．

図 3 に探索木共有方式の，**図 4** と **図 5** にクライアントサーバ方式の擬似コードを，各々示す．コード中の “※フラグ法” と “※ fpu 法” に関しては 5 節で述べる．**図 6** に各関数の機能の簡単な説明があるが，実装方式の理解に必要な範囲に止めているので，アルゴリズムの詳細が必要な場合は，本実装の基になっている文献 6) を参照して貰いたい．ただし，fpu は一般的な名称ではないので，ここで説明する．

UCB1 アルゴリズム[1] は最初に全ての枝（手）の値を求めなければならない．これは（特に末端のノードでは）かなりの負担になる．S. Gelly らは，これを改善する目的で fpu（first-play urgency）を導入した．

fpu は，使い方などは通常の “値（value）” と全く同じだが，最初に初期値を与える点が異なる．この初期値を ∞ にすれば，UCB1 アルゴリズムは値（fpu）が最大の枝から順に評価するので，元のアルゴリズムと全く同じ動作になる．しかし，1.0 近辺の適当な値

```
MCClient(id, globalQueue, queue)
 1  repeat
 2      message ← RecieveFromServer() ※
 3      if message ≠ FINISH
 4        then
 5          position ← message.position
 6          score ← DoSimulation(position)
 7          message.score ← score
 8          message.id ← id
 9          SendToServer(message) ※
10    until message = FINISH
11  return
12
13  なお,
14      message ← RecieveFromServer() ※
15      SendToServer(message) ※
16  の 2 行は CPU に応じて
17  x86:
18      message ← PopQueue(queue)
19      PushQueue(message, globalQueue)
20  Cell:
21      message ← ReadFromInMbox()
22      WriteToOutMbox(message)
23  となる(MCClient の引数も変わるが省略).
```

図 5 クライアントサーバ方式の擬似コード(クライアント)
Fig. 5 Pseudo code of client-server implementaion (client side).

にした場合,ある枝の値がその値より大きくなった時点で,残っている未評価の枝の評価は行われなくなる.これは有望そうな枝が優先的に評価されることを意味し,探索の高速化に結びつく.また未評価の枝を特別扱いする必要がなく,コードが簡単になる.

なお,クライアントサーバ方式の x86 と Cell の通信関連の実装は異なる.サーバからクライアントへの通信チャネルは,どちらもクライアント毎に持つが,クライアントからサーバへの通信チャネルは,

Cell の場合 クライアントからサーバへの通信チャネルもクライアント毎に持ち,サーバは全チャネルをポーリングする.また,通信には MailBox という,queue と同様の機構を用いている.

x86 の場合 ポーリングではスレッドの切り替えが起こらず,クライアントがコアと同数の時にデッドロックを起こす可能性があるので,全クライアントに共通の queue を用いている.

4.1 実験

探索木共有方式とクライアントサーバ方式の,盤が空の状態(先番初手)からの playout(探索木の降下,シミュレーションおよび探索木の更新などの一連の処理をまとめて playout と呼んでいる)速度を,x86 と Cell 上で測定した.測定は複数回行い,最も外乱の影響が少ないと思われる,実時間が最も短かかったものを採用した.結果を**表 1** に示す.

続いて処理時間の内訳を調べるために,プログラムの各処理の前後に時間測定用のコードを挿入し,表 1 と同じ条件で処理毎の CPU 時間を測定した.結果を**表 2** に示す.

4.2 議論

4.2.1 アルゴリズム

クライアントサーバ方式の場合,サーバは通常クライアントの要求に従ってサービスを提供する.我々も

GenerateMove($position$) トップレベルの関数で,与えられた局面($position$)に対する最善の着手($move$)を返す.ここではコードを簡単にするために手番を省略している.
DescendTree($position$) fpu が最大の手を選びながら探索木を降り,まだ一度も訪れてない($node.visits = 0$)ノードを返す.この際,後で値を更新するために,降った経路($path$; $node$ と $move$ の番号)を覚えておき,ノードと共に返す.
DoSimulation($position$) 手をランダムに選びながら(SelectRandomMove)終局($gameEnd$)まで打つ.
SelectMoveToX($node$) 与えられた $node$ で次に展開する手を選び,それを返す.
PlayMove($move, position$) 与えられた $position$ に $move$ を打ち,新しい $position$ を返す.
CreateNode($position$) $position$ に対応するノードを新しく作り,それを返す.全合法手をリストアップし,訪問回数,値,子ノードへのリンク等を初期化する.
UpdateTree($score$, $path$) $path$ を逆順に辿りながら各ノードと手の訪問回数と値(fpu)を,UCB1-TUNED アルゴリズム[1] にしたがって更新する.
Lock($lock$),UnLock($lock$) 排他制御用の $lock$ を各々獲得/解放する.

図 6 関数の簡単な説明
Fig. 6 Description of the functions in Figures 3 to 5.

最初,サーバはクライアントから要求が到着してから探索木の降下を開始し,クライアントに(シミュレーションして貰う)局面を送出する形で実装した.しかし,これでは並列度があまり上がらなかったので,現在の形,すなわちサーバはクライアントからの要求を待たずに探索木を降り,展開する手を見つけてから暇なクライアントを探し,もしあればそれに対して局面を送出する形に変更した.もし暇なクライアントが見つからなければ,クライアントからシミュレーション結果(スコア)が返ってくるのを待つ.今回の x86 用の実装のようにスレッドがコアより多ければ,この時点でスレッドの切り替えが起こる.

この意味ではマスタースレーブ方式と呼ぶ方が適しているかも知れないが,今後ネットワーク接続マルチ PC システムに適用した時に,クライアントからのリクエストに応じて動的に参加/離脱ができるようにする予定なので,クライアントサーバ方式のままにしている.

なお,探索木共有方式の場合,探索木を操作する時の挙動を制御するにはロックを細工するしかなく,ほ

表 1 先番初手の playout 時間と速度.N はスレッド数あるいはクライアント数.Ratio は Cell の探索木共有方式の 1 クライアントを 1 とした速度比.ST は探索木共有,CS はクライアントサーバの各方式,x86 は Q6600/3 GHz,Cell は Cell /3.18 GHz,x86 は 10^6,Cell は 10^5playout の平均.
Table 1 Playout speed and time for an empty board. “N” is the number of threads or clients. “Ratio” is the ratio of playout speed compared to one client shared-tree implementation on Cell. ST and CS are shared-tree and client-server implementations, respectively. Average of 10^6 and 10^5 playouts on an x86 (4 core) at 3 GHz and a Cell (6 SPU) at 3.18 GHz, respectively.

CPU	Method	N	Playout time	Playout/s	Ratio
Cell	ST	1	830 μs	1.2 k	1
Cell	ST	6	400 μs	2.5 k	2.1
Cell	CS	1	852 μs	1.2 k	0.97
Cell	CS	6	140 μs	7.1 k	5.9
x86	ST	1	163 μs	6.1 k	5.1
x86	ST	4	38.5 μs	26 k	22
x86	CS	1	159 μs	6.3 k	5.2
x86	CS	4	43.0 μs	23 k	19

表 2 先番初手の処理毎の CPU 時間．DESCENDTREE などは図 3〜図 5 に対応．Others はこれら以外，Total は全 CPU 時間，F は全体に占める並列化できない部分の割合で，F = 1− (DOSIMULATION の CPU 時間) / Total．スレッド数あるいはクライアント数は，x86 は 4，Cell は 6．他の条件は表 1 と同じ．

Table 2 Detailed CPU time for an empty board. DESCENDTREE etc. correspond the ones in Figures 1 to 3. F is the ratio of CPU time of unparallelizable parts, i.e., F = 1− (CPU time of DOSIMULATION) / Total. The number of threads or clients are 4 and 6 for x86 and Cell, respectively. Other conditions are the same as in Table 1.

CPU	Method	DESCENDTREE	CREATENODE	DOSIMULATION	UPDATETREE	Others	Total	F
x86	ST	5.40μs (0.84%)	8.12μs (1.3%)	591μs (91.6%)	9.32μs (1.4%)	31.5μs (4.9%)	645μs	0.08
x86	CS	3.96μs (0.44%)	7.82μs (0.9%)	625μs (70.2%)	3.62μs (0.4%)	250μs (28.1%)	891μs	0.30
Cell	ST	0.90μs (0.03%)	8.20μs (0.3%)	778μs (27.5%)	39.1μs (1.4%)	2003μs (70.8%)	2830μs	0.72
Cell	CS	0.70μs (0.07%)	5.70μs (0.6%)	812μs (82.4%)	2.40μs (0.2%)	164μs (16.7%)	986μs	0.18

とんど自由度がない．しかし，クライアントサーバ方式ではサーバは単一スレッドなので，自由にアルゴリズムを弄ることができ，色々な実験を行うには都合がいい．

4.2.2 Playout 速度と CPU 時間

表 1 では，Cell 上の探索木共有方式を除き，いずれも並列度 N に比例して速度が向上しており，MC/UCT アルゴリズムの潜在的な並列度の高さがうかがえる．

Cell で並列度 6 の場合，クライアントサーバ方式が探索木共有方式の約 3 倍速く，また表 2 でも Cell 上の探索木共有方式の “Others” の値が突出しており，探索木共有方式はオーバヘッドが大きく，Cell に適してないことを裏付けている．

Cell の PPU は，クロック周波数は 3 GHz を超えているが，インオーダーかつ FGMT（fine-grain multithreading）によるユーザとシステムの 2 スレッド実行のため，かなり非力である．したがって，PPU 側の 6 スレッド（SPU と同数必要）を切り替えるオーバーヘッドは相当大きく，これが探索木共有方式の遅さの主原因と考えられる．

クライアントサーバ方式の場合，PPU で動いているスレッドは実質的に 1 つなのでこの問題はなく，Cell の様な非対称マルチコアにはクライアントサーバ方式の方が適していると言えるだろう．

x86 ではクライアントサーバ方式は逆に約 1 割遅くなっている．この実験ではクライアント数がコアと同じ，つまりサーバースレッドを加えるとスレッドがコアより 1 つ多い．そこで，クライアントからの受信時に mutex を使ってスレッドが切り替われるようにしており，このオーバヘッドが速度低下の原因の可能性が高い．

探索木共有方式にも探索木の排他制御に伴うオーバヘッドがあるが，これはクライアントサーバ方式の queue の排他制御と同程度だろうから無視できる．このオーバヘッドは，クライアントを 1 つ減らしてサーバに 1 コアを割り当てればなくすことができるが，総合的な性能はクライアントが減った分低下する．

この実験では，クライアントサーバ方式の実行速度の上限を調べるためにクライアントとコアを同数にしたが，外部（ネットワーク接続）のクライアントがある場合はサーバに 1 コアを割り当てるのが妥当だろうから，このオーバヘッドはなくなる．いずれにせよ，これは使用状況に応じて変更すればよい問題だろう．

また，実験結果は示してないが，終盤，ほとんどの局面が探索木の中に納まって処理がサーバ側で完結する場合，クライアントサーバ方式は探索木共有方式よりはっきり高速であった．

表 2 の F は，今回の各実装と CPU の組み合わせの「伸び代」（潜在的な並列度の高さ）を調べる目的で算出した．良く知られている様に，全処理時間の並列化できない部分の割合（F）が小さいほど，並列化の効果は高い．

表 2 の数値から，最も伸び代が大きいのは探索木共有方式で，また x86 には探索木共有方式，Cell にはクライアントサーバ方式が適していることは明らかだろう．しかしクライアントサーバ方式は，両 CPU ともに悪くない値を得ており，探索木共有方式が共有記憶型のシステムにしか適用できないことを考えると，良い方式であると言える．特に Cell の数値は，まだ高速化の余地が十分あることを示している．

5. モンテカルロシミュレーションの並列化と UCT アルゴリズム

5.1 問　　題

UCT アルゴリズムを，並列モンテカルロシミュレーションに適用した時の問題とは，先行するシミュレーションの結果による探索木の状態の更新を待たずに，探索木を降って手を選んでしまうという，アルゴリズムの振る舞いの変化，そしてその結果生じる性能低下の可能性である．ここでは，この問題の影響と提案する改善方法の効果を，結果として生じるであろう勝率の変化で評価する．

問題を探索木共有方式に沿って考察する．あるスレッドがロックを獲得して探索木を降り，ある手を選んでロックを解放すると直ちに，ロックの解放を待っていた他のスレッドがロックを獲得し，前のスレッドと全く同じ経路を辿って同じ手を選び，この結果，同じ手が複数のスレッドによってシミュレートされる．

これが無駄になるかどうかは一般的には分からないが，UCT アルゴリズムの基になっている UCB1 アルゴリズム[1)] は，最初にノードの全ての手を 1 回評価し，その後評価値に基づく選択的探索を開始するので，最初の評価を早く終えることが探索の効率を良くする．したがって，少なくともあるノードを展開した直後は，異なるスレッドが同じ手を選ばないようにした方が，探索の効率が良くなる．しかし常にこれを実行すると，

本来続けて選ばれるべき手が選ばれなくなるので，最終的に性能が改善されるかどうかは，実際に評価しなければ分からない．

5.2 改善法

ここでは，次の

フラグ法 ある手を展開する時にその手のフラグをセットし，展開する手を選ぶ時にはフラグがセットされている手を選ばないようにする．そのシミュレーション結果に基づいて探索木の状態を更新する時にフラグをリセットする．

Fpu **修正法** ある手を展開する時にその手の fpu を減らして，他のスレッドがその手を選ばないようにする．

2 つの改善方法を実装（図 3）し，勝率の変化を調べた．なお，図 4 ではコードを省略しているが，実装は自明だろう．

5.3 実験

各方式とそれに改善方法を適用した場合の，GNU GO 3.7.10 level 10 に対する勝率および ELO を**表 3** に示す．GNU GO の引数に

```
--level 10 --max-level 10 --min-level 10
```

を与え，レベルをできるだけ 10 に固定した．Playout 数は，GNU GO に対する勝率が 50%前後になるように，一手辺り 2,400 回とした．全ての対局は 4 コア（Intel Q6600）の自作 PC 2 台で行った．表中，“+ フラグ” はフラグ法，“+ fpu” は fpu 修正法を用いた場合を，各々示す．

5.4 議論

1 行目の探索木共有の 1 スレッド時の勝率を基準にして，4 スレッドにした時は勝率で 3.7%，ELO で 25 低下した．また，クライアントサーバ実装は勝率で 5.1%，ELO で 35 低下した．

探索木共有方式の場合，最初のスレッドがシミュレーションを終えれば，(ロックが取れ次第）探索木の状態は更新される．しかし，クライアントサーバ方式の今回の実装では，探索木の更新より局面の配布を優先しているので，勝率の低下も探索木共有方式より大きいものと思われる．

実験結果の勝率の変化からは，フラグ法の方が fpu 修正法より優れているように思えるが，確率的な誤差を考慮すると，両者の効果に明確な差があるとは言い難い．実装上は，fpu 修正法の方がフラグのリセットが要らない分簡単である．またフラグ法は，あるノード中の手全てにフラグがセットされている時の処理が必要になる点も不利である．

本実装では fpu の初期値が一定なので，fpu に掛ける値は勝率に影響しないが，文献 5) のように手の事前評価に基づいて fpu の初期値を変える場合は，検討が必要だろう．

結論として，これらの方法を使った時の勝率の低下は，我々の実験の 4 並列では最大 20 ELO と小さく，実用上問題にならないと思われるが，並列度がもっと高い場合はさらに検証が必要だろう．

表 3 各方式の GNU GO 3.7.10 level 10 に対する勝率．ST は探索木共有，CS はクライアントサーバ．Playout 数は 2,400/手，勝率は先後交替 2,000 対局の平均，± の後の数字は標準偏差．プラットフォームは 4 コアの x86 PC，1 行目以外は 4 スレッドあるいは 4 クライアント．フラグと fpu は本文参照．

Table 3 The winning rate of each implementaion against GNU GO 3.7.10 level 10 on a 4-core x86 PC. Average of 2,000 alternating B/W games. The numbers after ± are standard deviations. All except first column are of 4 threads or 4 clients.

方式	勝率	ELO
ST（1 スレッド）	50.4 ± 1.1%	+2
ST	46.7 ± 1.1%	−23
ST + fpu	47.4 ± 1.1%	−18
CS	45.3 ± 1.1%	−33
CS + フラグ	48.9 ± 1.1%	−8
CS + fpu	48.2 ± 1.1%	−13

6. まとめと今後の課題

リカレントニューラルネットの応用研究のテストベッドとして，今回開発した囲碁ソフトの，並列 MC/UCT アルゴリズムの実装について述べた．広く使われている探索木共有とクライアントサーバの両方式を異なるプラットフォーム上に実装して実行速度を比較・検討した．

クライアントサーバ方式の実行速度は，探索木共有方式と比べて PS3 で 3 倍高速になった．Cell の様な非対称マルチコアにはクライアントサーバの方が適していると言えるだろう．また x86 では逆に 10% 遅くなったが，1 コアをサーバに占用させることで改善できる．

UCT アルゴリズムを並列モンテカルロシミュレーションに適用した時，アルゴリズムの挙動が変わり性能が低下する可能性がある，という問題を検討し，改善法を提案した．この問題の影響と改善法の効果を GNU GO に対する勝率で評価した．4 コア CPU を使った実験では勝率の低下は最大 35 ELO，提案した改善法を用いた場合最大 20 ELO で，実用上無視できるだろう．

今後の課題は，クライアントサーバ方式のネットワーク接続マルチ PC システムへの適用だが，その前に遅延時間が性能に与える影響を定量的に評価する予定である．また，本来の目的であるリカレントニューラルネットのシミュレーションへの導入も早く行いたい．

参考文献

1) Auer, P., Fischer, P. and Cesa-Bianchi, N.: Finite-time Analysis of the Multi-armed Bandit Problem, *Machine Learning*, Vol. 47, pp. 235–256 (2002).
http://www2.imm.dtu.dk/pubdb/p.php?2088
2) Cazenave, T. and Jouandeau, N.: On the Parallelization of UCT, *Proceedings of the 6th International Conference on Computer and Games*, Amsterdam, Netherland, pp. 93–101 (2007).
3) Coulom, R.: Private communication (2007).
4) Dailey, D.: scalability studies with UCT, *computer-go mailing list* (2006).
5) Gelly, S. and Silver, D.: Combining online

and offline knowledge in UCT, *Proceedings of the 24th International Conference on Machine Learning* (Ghahramani, Z., ed.), USA, Omni Press, pp.273–280 (2007).
http://www.machinelearning.org/proceedings/icml2007/papers/387.pdf （加藤英樹訳: UCT でオンライン知識とオフライン知識を組み合わせる (2007).
http://www.gggo.jp/387.jp.pdf）

6) Gelly, S., Wang, Y., Munos, R. and Teytaud, O.: Modification of UCT with Patterns in Monte-Carlo Go, Technical Report RR-6062, INRIA (2006). http://hal.inria.fr/inria-00117266 （加藤英樹訳: モンテカルロ碁における UCT のパターンによる改良 (2007). http://www.gggo.jp/RR-6062-v3-jp.pdf）
7) 加藤英樹，竹内郁雄：系列連想記憶を用いた囲碁ソフト，第 11 回ゲーム・プログラミング ワークショップ予稿集，IPSJ SIG-GI, pp. 151–154 (2006).
8) Kocsis, L. and Szepesvári, C.: Bandit-Based Monte-Carlo Planning, *Proceedings of the 15th European Conference on Machine Learning* (Fürnkranz, J., Scheffer, T. and Spiliopoulou, M., eds.), Berlin, Germany (2006).