解　説

「激指」におけるゲーム木探索並列化手法

Parallelization Technique of Min-Max Tree Search on Gekisashi

横山 大作　東京大学
Daisaku YOKOYAMA　The University of Tokyo
yokoyama@tkl.iis.u-tokyo.ac.jp

keywords: game-tree search, parallel computing, shogi

1. は じ め に

ボードゲームに代表される二人零和完全情報ゲームにおいては、ゲーム木 (Min-Max 木) を探索することによって最善手を求めるというアプローチが広く用いられている。特に、オセロ、チェスや将棋などに関しては、様々な工夫を加えたゲーム木探索を行うことで、人間に匹敵する強さを持つコンピュータプレイヤを実現してきた。

ゲーム木探索でより良好な解、より強いプレイヤを得るためには、より遠いところまで先読みを行う、より大きいゲーム木を探索することが求められる。近年の計算機環境の進化は、多数の計算機を同時に利用して速度向上を図るという並列処理の方向に向かっており、ゲーム木探索においてもその並列化が望まれている。単純な木構造の探索であれば、そのアルゴリズムの並列化は容易であり、効率よく高速化を図ることができるが、多くの工夫を加えた実問題のゲーム木探索を対象にする場合は、並列化による高速化に際して大きな障害が存在する。逐次処理を前提にした効率化手法が多数用いられているため、処理の部分間に依存関係が生じたり、依存関係を無視して並列実行しても無駄な計算が増えるだけであったりという状況が起きがちであり、実効的な探索の速度向上は簡単には実現できない。特に、近年身近なものとなってきた多数の CPU を持つ高並列環境下では、その性能を十分に生かすことが難しく、さらなる研究が求められている。

本稿では、将棋のコンピュータプレイヤである「激指」[激指] を題材に、実問題において多数の工夫を施したゲーム木探索を並列化する際に用いられる技術とその性能を紹介する。激指は 1999 年より開発を続けており、毎年行われる「世界コンピュータ将棋選手権」において 4 度優勝するなど、トップレベルの性能を持つプレイヤである。将棋の問題領域で高度な逐次効率化手法を適用した探索木がどのような性質を持つかを検証し、その探索木の並列化ではどのような性能が得られるのかを、代表的なアルゴリズム間で比較検証していく。

人工知能分野でしばしば用いられる他の探索問題においても、同様に問題領域特有の効率化が施され、並列化による高速化の妨げになっていることも多いと思われる。効率化された逐次探索の性質保持と投機的な探索のバランスをどのようにとり、実効的な性能向上につなげるのか、ゲーム木の並列化における知見が生かされる場面は多々あるのではないかと考えている。

2. ゲーム木探索の並列化

枝刈りを考えない Min-Max 木の探索を並列化する方法は自明であり、高い並列性能が期待される。しかし、Min-Max 木の探索においては、計算効率を劇的に改善する $\alpha\beta$ 探索アルゴリズム [Knuth 75] が知られており、これを利用しなければ実用的な規模の問題は解けないと言ってよい。$\alpha\beta$ 探索は、先に探索したノードの値を用いて後の探索を枝刈りし、無駄な探索を省くことで効率化を図る手法であり、効率がノードの探索順序に大きく依存する。また、チェス、将棋などの実問題において強いプレイヤを作成するためには、先に読んだ手の情報を他の手の探索に利用する (killer move heuristic) など、探索順序に依存した様々な効率化手法が利用されている。探索を並列化する際には、これらの効率化手法を無視した手法では無駄な計算を増やすだけであり、探索の高速化は望めない。逐次探索の探索順序やその他の効率化技法をどの程度遵守し、どの部分でそれを無視して並列性を抽出するのか、という面での最適点を探ることが、ゲーム木探索の並列化では重要な要素となる。

2·1 並 列 化 手 法

ゲーム木探索の並列化に関する研究は古くから多数存在しており、実行環境として共有メモリ型並列計算機と PC クラスタに代表されるような分散計算環境との双方ともが想定されている。ここでは、共有メモリ型並列計算機を想定した既存研究を中心に紹介する。

i.　PV-split

ゲーム木の探索の並列化手法として、最初期に提案されたものに PV-split[Marsland 82] がある。ゲーム木において、根ノードから (Max 手番、Min 手番のそれぞれの立場で) 最善の子供を選びながら子ノードをたどって葉

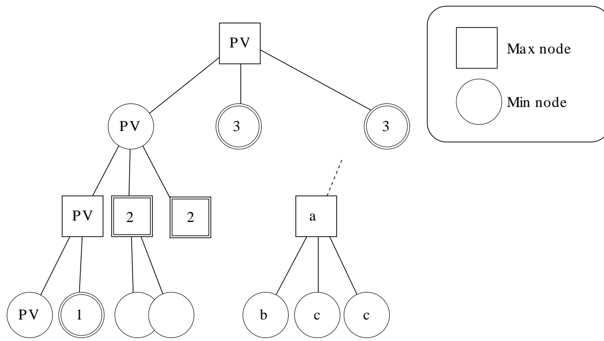


図 1 ゲーム探索木

まで至るとき、その経路上のノードを Principal Variation (PV) と呼ぶ。$\alpha\beta$ 探索で効率よく枝刈りを行うためには、全てのノードで、子ノードを最善のものから順に展開していく必要があるが、現在のチェスや将棋などの強いコンピュータプレイヤでは、反復深化などの様々な工夫を用いることで子ノードの良い探索順序制御を精度よく実現できている。順序制御が成功しているとき、PV ノードは図 1 に示したゲーム木のように、根ノードから最左の子供をたどった経路に存在することになる。PV-split はこのような状況を期待し、根ノードから最初の (最左の) 子ノードをたどった経路 (図 1 の PV と記されたノード) をまず展開する。その後、PV の子ノード (図 1 の二重線で囲まれた Max/Min ノード) を並列実行可能なタスクとして扱い、並列に探索する。これらの子タスクの間での順序関係は並列実行によって守られなくなるが、最左のノードが PV でありかつその値が求まっていれば、順序が変化することによる探索効率への影響は小さいことが知られている。このような特性を利用するため、それぞれの深さの PV ノードの値が確定してから兄弟のノードを探索するという制御を行う。よって、最初に最も葉ノードに近い PV ノードの兄弟ノード (ノード 1) を並列に探索し、それによって値が確定する親 PV ノード (一つ根ノードに近いノード) の兄弟 (ノード 2) を並列探索する、という過程を繰り返す実行順序をとることになる。

PV-split では、それぞれの PV ノードの子ノードの数だけしか並列度が存在せず、多数の CPU コアが利用可能な場合、リソースを埋めるだけの並列度がない可能性がある。また、一度分割された子タスクがそれ以上分割されないため、負荷の不均衡による性能劣化を生じやすくなる。あるノードの下に展開される探索木の大きさは局面によって大きく変化するので、PV-split で分割されたタスクは必要な計算量が大きく異なったものとなり、大きなタスクを担当した一部の CPU のみが働くことになってしまうのである。

ii. YBWC

PV-split を改良し、全てのノードで並列実行タスクを生成するようにしたのが Young Brothers Wait Concept (YBWC) [Feldmann 93] である。YBWC では、任意のノード (図 1 ノード a) で、最初の子ノード (b) を探索し、その値が決まったら残りの子ノード (c) を全て並列に探索する。逆に、YBWC の制御を最左辺ノードのみに制限したものが PV-split であるとも言える。

YBWC では全てのノードがタスク分割点となるため、並列実行可能なタスクが足りなくなる心配はほとんどない。細粒度の並列実行となるため、実際に利用できる限られた CPU コアをタスクに割り当てる、タスクスケジューリングを適切に行う必要がある。YBWC の場合、固定されたスケジューリングを行うのではなく、タスクがなくなった CPU に対してタスクを割り当てる動的なスケジューリングが少なくとも必須であり、その上で、探索効率を向上させるための種々の工夫が試みられている。

代表的な工夫に、helpful master concept [Feldmann 90] がある。あるスレッドが暇な状態になったとき、自分の子ノードの探索を行っている他スレッドの仕事に対してのみ、合流して並列探索の協力を行うという、スケジューリングの制約を加えるものである。逐次探索での探索順序をより遵守する方向でスケジューリングする手法であり、並列化による無駄な探索の増加を抑える狙いがある。この制御は、著名なチェスプログラムである Stockfish [Romstad] にも見られる。

iii. YBWC 改良

YBWC* [Feldmann 93] では、$\alpha\beta$ 探索木におけるノードの性質の違いを用いて、最初の子ノード数個を逐次展開するノードと最初から全ての子を並列展開してしまうノードとを使い分ける。ABDADA [Weill 96] や Dynamic Tree Splitting (DTS) [Hyatt] では、最左の子ノードについて、基本的には YBWC と同様に扱うが、暇な CPU が多いときには最左の値が完全に定まるのを待たずに残りの子ノードの並列展開を実行してしまう、という制御を行う。DTS では最左の値が変更されたときにその情報を右の子ノードに伝えるという処理を加えて、無駄な探索の増加を防いでいる。

なお、ABDADA や DTS はスレッド制御の実装方法やスケジューリング手法にも大きな変更が加えられているが、並列タスクを生成するノードの選び方と基本的なタスク順序制御ポリシーに関しては YBWC の派生であると言える。

iv. 分散探索

ここまでに紹介した研究は、ハードウェアによる共有メモリを持つ並列計算機を想定したものであった。一方、メモリを共有しない独立した計算機を多数ネットワークで接続して用いる分散計算環境も身近なものとなっており、そのような環境を想定した研究も進められている。マスタワーカ構造で非同期に局面情報をやりとりすることで探索木を分割する APHID[Brockington 97]、分散ハッシュテーブルを用いた Transposition Table Driven Scheduling [Kishimoto 02] などがその一例である。

分散計算と共有メモリを用いた並列計算の違いは、CPU

表 1 激指の指し手生成

(1) transposition table チェック、手損チェックなど
(2) null-move forward pruning
(3) ヒューリスティックな手
(4) 浅い読みでの最善手
(5) killer-move(存在すれば)
(6) パス
(7) その他の手 (実現確率に従った重みで生成)
(8) 詰みチェック (専用アルゴリズムによる)

間の通信のレイテンシと帯域幅のみであり、アルゴリズムに関係して本質的な意味での相違点は存在しない。そのため、共有メモリ環境での並列化において得られた知見は、分散環境でも役立つと考えられる。また、近年の CPU コア集積度の向上に伴い、共有メモリ計算環境においても通信性能の局所性、すなわち近いコアと遠いコアの違いを考慮した並列化を行う必要が増しており、分散計算環境での知見が大切になると考えられる。

2·2 激指の並列化手法

激指は YBWC に近い探索制御を行っている。探索木中の各ノードで、激指は表 1 に示したリストの順に手を生成し、子局面を展開する。このうち、「(6) パス」までの手を逐次的に展開し、その後「(7) その他の手」以降を並列に展開する。つまり、最初の一手のみでなく、ヒューリスティックで決めた数手の探索結果を待ってから並列タスクを実行する、というアルゴリズムである。

現在の実装におけるタスクのスケジューリングは、アイドルなスレッドが存在したらまだタスクがある任意の並列実行ポイントに合流する、というものであり、結果としてランダムなタスク選択となっている。

3. 実 験

3·1 評 価 用 局 面

将棋のような実ゲームにおける探索木は、局面の性質の違いによって大きく振る舞いが変わることが知られている。例えば、同程度に良い評価を返す手が多数存在する局面では枝刈りを効率よく行うことができず、探索に必要な計算量が増えがちであり、逆に「一手だけしか有効手がない」ような局面では、たとえ合法手が多数存在しても枝刈りによって計算量を抑えることができる。

また、激指は実現確率打ち切り [Tsuruoka 02] という探索制御手法を採用している。この手法は、指し手の特徴 (「取る手」などのカテゴリ、指し手周辺の駒の配置など) から、指し手ごとに末端評価までの残り深さを変化させ、ゲーム木の枝を選択的に展開していくものであり、特徴に応じてゲーム木の形や規模が大きく異なるものになる。このため、激指においては局面ごとに探索計算量が大きく変化しがちである。

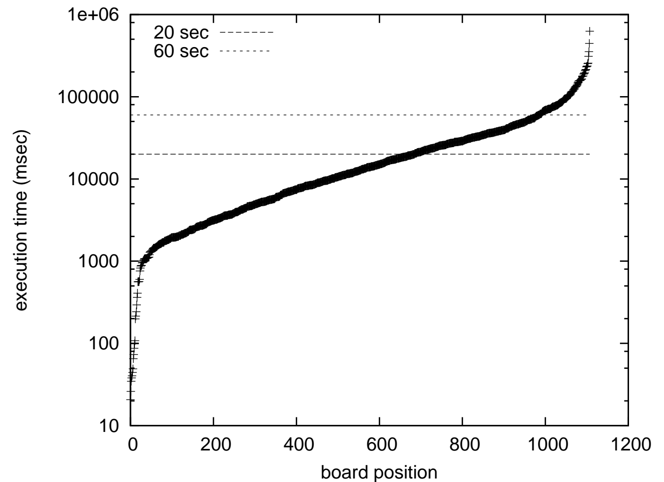


図 2 局面ごとの探索時間分布

図 2 は、人間の高段者が指した棋譜からランダムに 1100 個程度の局面を抽出し、激指を用いて同じ打ち切り条件で探索を行わせたときに各局面で必要だった計算時間をプロットしたものである。横軸が各局面、縦軸が計算時間を表し、横軸の局面の順番は探索時間が短いものが先になるように並べられている。実験環境の CPU は Xeon E5530 2.4GHz, 並列化は行わない逐次探索である。最も深くまで読む枝が 16 手になる木を作成するよう打ち切り条件を設定した。一部の例外を除いて、1 秒から 300 秒程度の範囲に探索時間が分布しており、局面に応じて数百倍程度の計算量の差が生じることがわかる。例外的に探索時間が極度に短くなっているものは、最終盤で即座に勝ち負けがわかるような局面である。また、極端に探索時間が長くなる局面もごく一部存在し、今回の実験では最長で 630 秒程度必要なものがあった。

以後の並列探索に関する評価実験においては、この探索時間分布を踏まえ、高速化の要求が大きい比較的大規模な探索木ができる問題、かつ実験が行いやすい規模にとどまるような問題を利用するため、図 2 の 20 秒から 60 秒の範囲に計算時間が収まる局面群からランダムに抽出した 50 個を用いることとした。

3·2 分 枝 数 の 分 布

i. 実効的な分枝数

並列探索時における振る舞いを理解するために、まずは探索木の分枝に関する情報を取得することにした。図 3 は、実験に用いた 50 個の問題において、探索木中の各ノードでの実効的な分枝数、すなわちカットが起きるなどしてそのノードでの探索が終了するまでに展開した子供の数を測定し、実効分枝数ごとにその出現回数をプロットしたものである。“all node” は探索木の全てのノードでの値、“split node” は激指が探索の並列化を試みるノードでの値である。ここでは、残りのゲーム木の最大深さが 7 以上になるような場合に並列化を試みるよう設定し

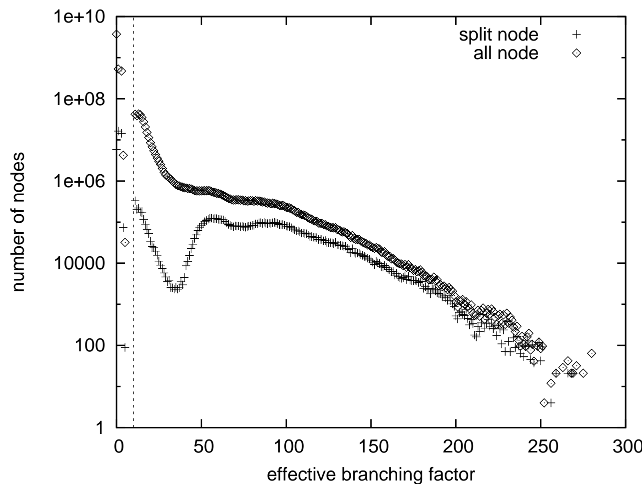


図 3 50 局面における実効分枝数の分布

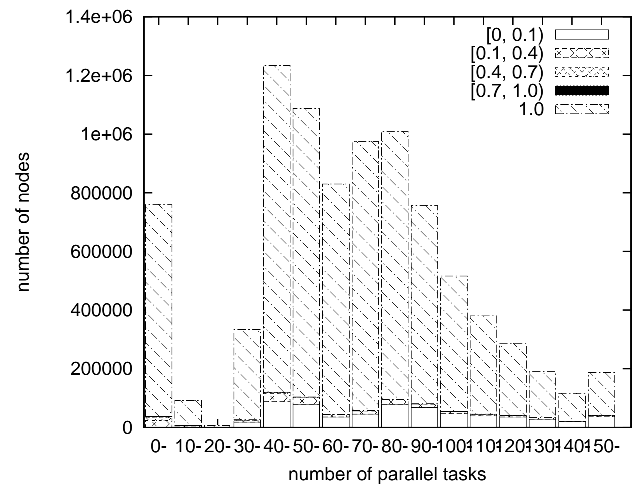


図 4 並列タスク数と実効分枝数割合 (並列探索条件ノード)

表 2 並列探索部分のタスク数割合

	並列部分割合
split node	0.36
all node	0.41

ている。2·2 節に示したように、激指は最初の数個の子ノードを逐次的に探索し、「一般的な手」の子ノードについてのみ並列化の対象とする。ここでは、そのノードの探索終了がどの時点で起きたかを明確にするために、並列化対象になる手を「10 個目以降の分枝」として表示する。つまり、逐次探索の 3 手目で結果を返した場合は分枝数 3、並列探索対象の 5 手目で結果を返した場合は分枝数 15 として数え、出現数をプロットしている。図 3 の横軸=10 の箇所に点線でその境界を示す。

図 3 より、ほとんどのノードでは逐次探索を行う部分の手で値が定まっていることが見て取れる。表 2 に、全体のタスク数に占める並列化対象部分 (図 3 の分枝数 10 以上の部分) のタスク数の割合をまとめた。タスク数は実際の計算量と直接的に結びつくものではないが、多くのタスクが逐次部分に含まれており、並列化した探索を開始するのに適したノードが割合的にはあまり多くない傾向にあることがわかる。なお、表 1 の (1) の部分はそれ以上子ノードを展開しないタスクであるため、ここで割合を求める際には (1) を除いたタスク数を用いることにした。

また、全てのノードを集計した場合は、実効分枝数は単調に減少しているが、現在の激指が並列探索のためのタスク分割対象としている、残り深さが一定以上大きいノードについては、実効分枝数は 100 程度のところまで大きな値を保っており、比較的多数の子ノードを展開する必要があったことが見て取れる。これは、並列に探索できる子ノードが多数存在する場合が多いことを示しており、並列化に希望が持てる結果となっている。しかし、今後の計算機環境のトレンドである CPU コア数増加に伴い、より細粒度の並列性を抽出するために木の末端の方までタスク分割点を広げていくと、本実験の全ノードでの分枝数分布に近づき、多くのノードで少数の子ノードしか生まれない状況に近づいていくと考えられる。これは並列化を難しくする要因の一つとなり、何らかの対策が必要とされる可能性がある。

ii. 並列タスク生成数と必須タスクの割合

並列探索を行うノードで、どれだけの数の子ノードがタスクとして生成され、そのうちどれだけの子ノードが探索された時点で解が確定しそのノードの値が定まったか、という割合も、並列探索の性能を制約する大きな要素となる。並列に実行できる状態で生成されたタスク数に対し、解を求めるために本当に必要だったタスクが少ない場合は、いくら並列数を増加させても無駄な仕事を行うばかりであり、実効的な探索性能が向上しないことが予想される。対象問題におけるこの割合を測定した結果を図 4 に示す。

図 4 の横軸は、並列実行できる状態で生成された子ノードの数を表す。具体的には、表 1 の「(7) その他の手」以降に生成された数である。最初の柱が子ノード数 0 以上 10 未満、次の柱が 10 以上 20 未満、のように集計を行い、該当するグループの出現回数を縦軸にとっている。各々の柱は、解を定めるために本当に展開しなければならなかった子ノードの割合、すなわち実効分枝割合で色分けされている。$[0, 0.1)$ は実効的な分枝が 0 以上 0.1 未満、1.0 は全ての子ノードを探索する必要があった場合を示している。また、集計対象ノードは激指が並列探索を行うノードである。

並列タスク数は 40 から 90 程度に出現回数のピークを持ち、並列実行時には多数の子タスクを生成できていると考えられる。また、並列タスク数に関わらず、全ての場合で実効分枝数が 1.0 の場合が大きな割合を占めている。生成されたタスクは全て必要なものである場合が非常に多いことを示している。これは、並列タスクを生成

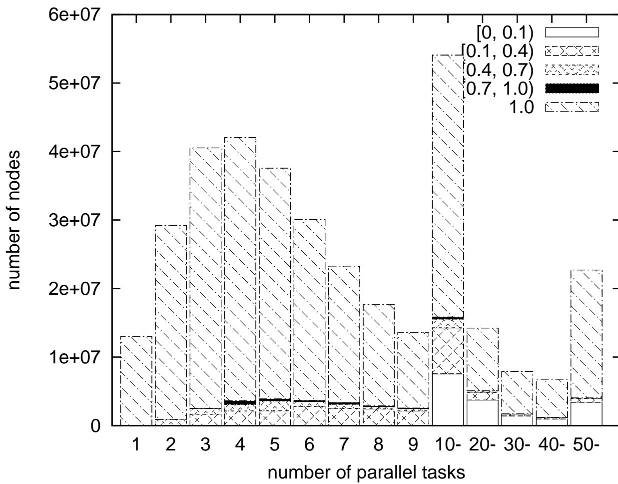


図 5 並列タスク数と実効分枝数割合 (全ノード)

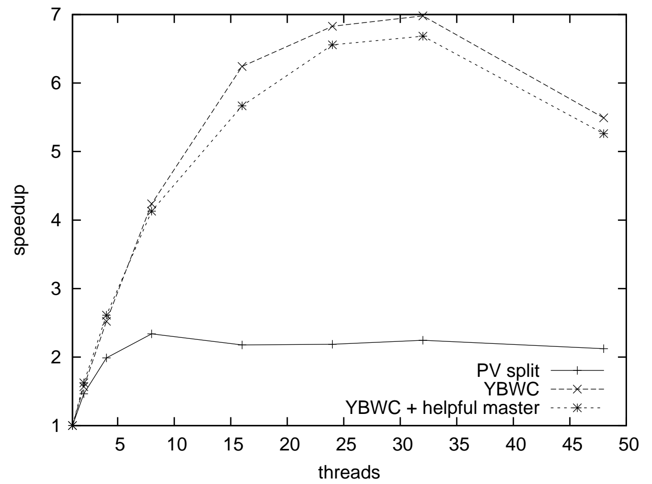


図 6 速度向上率

する前に存在する逐次部分が十分な精度で枝刈りを行っていることに起因すると考えられる。ゲーム木のノードは、根ノードからの探索解となる PV ノード、枝刈りが行える cut ノード、全ての子ノードを展開する必要がある all ノード、の 3 種類に分けられる [Knuth 75] が、このうち cut ノードにおいては表 1 の (6) までの逐次探索によって大部分が枝刈りされてしまい、並列タスクが生成される時点まで到達したノードはほとんどが all ノードになっていると推察される。このことは、結果として並列探索のために良い性質を保っている箇所で探索の分割が起きていることを示している。

同様の集計を全てのノードで行った結果を図 5 に示す。図 4 と比較して並列タスク数が少ないノードの割合が大きいため、並列タスク数 10 未満のグループを分割し、それぞれのタスク数ごとに集計している。また、タスク数 50 以上のグループの割合が小さくなったため、集計ではまとめている。図 4 と同様、実効分枝数割合が 1.0 であるノードがほとんどを占めていることが見て取れる。細粒度並列を指向した際、並列タスク数そのものが少ないことは問題ではあるが、並列探索されたタスクが無駄になりにくい良い性質を保っていることは、性能向上に期待が持てる結果である。

3·3 実験概要

3 通りの並列探索アルゴリズムを用いて、その速度向上率ならびに探索ノード数の変化を測定する。用いるアルゴリズムは以下のとおり。

- PV split
  PV ノードにおいてのみ子ノードの探索を並列に実行するもの。激指を改造したため、各ノードにおいて最初の数手を逐次に展開した後に残りの手を並列探索するという構造が残っている。
- YBWC
  オリジナルの激指が利用しているアルゴリズム。最初の数手を逐次探索するため、純粋な YBWC (最初の一手のみを探索した後で残りの手を並列探索する) とは若干異なる。
- YBWC + helpful master
  激指の YBWC において、タスクスケジューリングに helpful master の制約を加えたアルゴリズム。アイドル状態になっているスレッドを別のスレッドが自分の探索に参加させるとき、もし自分のノードの祖先ノードの結果を待っているアイドルスレッドが存在すれば、それを優先して参加させるように修正したものである。厳密に helpful master のスレッドのみを参加させるものではなく、アイドル状態のスレッドの子孫により適切なタスクが存在していても、運悪く別のスレッドがアイドルスレッドを奪うことがありえる。

実験環境は以下のとおり。

- CPU: AMD Opteron 6174 (1 socket あたり 12 core , 2.2GHz), 4 socket, total 48 core
- OS: linux (カーネル 2.6.32)
- transposition table: エントリ数 64M。これは、使用した個々の問題を解くためには十分大きな量である。

並列探索を行うと、ノードへの訪問タイミングの変化によって探索木の形が変化するために、偶然大きく枝刈りが可能になったり逆に大量に無駄な探索が発生するなどの影響がしばしば起こり、探索時間は大きくバラつくことになる。そのため、本実験では各問題局面ごとに 5 回ずつ探索を行い、その平均時間を用いて各問題局面の速度向上を測定した。

3·4 実験結果

アルゴリズムを変化させたときの速度向上率の変化を図 6 に示す。1 スレッドでの探索実行時間を基準とし、実験計算環境の 48CPU コアを全て利用した場合までスレッド数を変化させて測定を行った。なお、アルゴリズムを

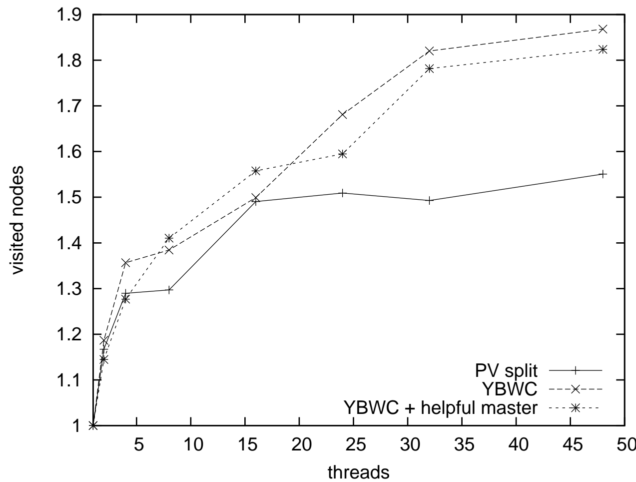


図 7 探索ノード数の増加率

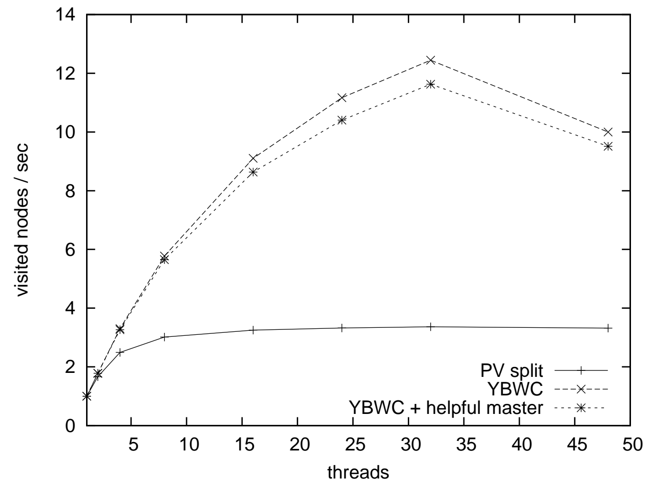


図 8 単位時間あたり探索ノード数の増加率

変えても 1 スレッドでの探索実行時間にはほとんど影響がなかった。50 問の実験対象局面それぞれに速度向上率を求め、その幾何平均を用いて全体の速度向上率とした。

また、それぞれの探索過程で展開したノード数の変化割合を図 7 に示す。全ての問題での探索ノード数を合計し、逐次探索時のノード数との比を求めた。並列探索を行うと、逐次探索ならば展開しないですんだノードを展開してしまい、無駄な仕事が行われることがしばしば観察される。

PV split は 8 並列で約 2.3 倍の速度向上が得られたが、それ以上並列度を増加させても性能向上は得られなかった。ノード数の推移を見ると、16 並列以上の並列度では 1.5 倍程度のノード数でほぼ一定しており、並列実行を試みるタスク数がこのあたりで足りなくなったため速度向上が得られなくなったと考えられる。

YBWC にすると、速度向上は 32 並列時に 7 倍程度まで達したものの、それ以上の並列度では向上率が低下した。PV split より多くのノードで並列実行を試みるため、期待通り性能は向上したが、探索ノード数は並列度が高い領域で増加しており、無駄なタスクの増加が速度向上を妨げる一つの原因になっていることが推察される。

YBWC に helpful master concept によるスケジューリングの制約を加えると、探索ノード数は若干抑えられた。helpful master concept は、自分のタスクを早く終わらせられる子タスクのみを助けるという手法であり、無駄なタスクの手助けを行わないという狙いがあるが、その効果がある程度確かめられたといえる。しかし、速度向上率は若干落ち込み、最大で 6.7 倍程度にとどまった。スレッドのスケジューリングに制約を加えたため、暇なスレッドを十分に働かせられなかった可能性がある。

図 8 に単位時間あたり探索ノード数の増加率を示す。PV split に関してはある程度の並列度で一定値に収束しており、その時点で同時実行可能なタスク数やスレッド数による制約に達して性能向上が頭打ちになったのではないか、と考えられる。一方、YBWC、YBWC + helpful master に関しては、32 スレッドから 48 スレッドに並列度を上げたときにこの値が減少しており、何らかの実装上のボトルネックによってアイドルなスレッドが増えたときに悪影響を受けている可能性が疑われる。

3·5 考 察

現在の激指のタスクスケジューリングはランダムなものであり、本稿では helpful master concept による制約を加えて性能を比較したが、性能は若干劣化する結果となった。

激指は、並列実行可能なタスクを持つスレッドが暇なスレッドを発見して計算に参加させるという実装を用いている。そのため、暇なスレッドが必ずしも自分の子孫ノードの探索スレッドに発見されず、偶然不適切なスレッドに補足されることが起きうる。制御構造を変え、暇なスレッドが全てのワーカスレッドの状況を見て最適なノードに参加するという実装を用いると、今回の実験とは異なった性能が得られる可能性がある。

また、32 並列を越えた並列度で単位時間あたり探索ノード数が低下したことは実装上のボトルネックの存在を疑わせる。transposition table (探索部分木の計算結果をメモ化しておく機構) の衝突、種々のロック部分の衝突など、いくつか想像される実装上の要素を検証していく必要がある。

単位時間あたり探索ノード数が 32 並列で 12 倍強しか得られないのは、前述の実装ボトルネックの可能性とともに、逐次部分の多さによる並列タスクの不足も原因となっていることが考えられる。さらなる性能向上のためには、DTS に用いられている、逐次部分が少し進んだ時点で後ろの並列部分を開始してしまうという投機的な並列化手法も考慮する価値があると考える。

4. ま と め

本稿では、ゲーム木探索の並列化において用いられる手法を共有メモリ環境を中心に紹介し、将棋のプレイヤである「激指」を題材にその実装と探索されるゲーム木の性質を紹介した。激指は効率の良い逐次探索アルゴリズムを採用しており、並列化のために現在採用しているYBWCの変種アルゴリズムは、並列タスクの多くの部分が無駄にならないことが期待されるという意味で良い性質を持っていることが示された。

また、代表的な探索アルゴリズムを激指に適用し、その性能比較を行った。PVSに対しYBWCが高い性能を示し、実効性能として32並列で7倍程度の速度向上を実現することができた。一方、YBWCのスケジューリングにおいて逐次探索順序を遵守する方向で制約を加えても探索性能は若干劣化する結果となった。激指のスケジューリング時には一部のスレッドに関する情報だけを利用しており、全てのスレッドの情報を利用できるような実装を行った時にはまた異なった性能が得られる可能性がある。今後の課題としたい。

探索問題は本質的に計算量爆発を起こすものであり、並列計算はそれを解決する手段ではない。しかも、ゲーム木のように複雑な制約が加えられた実問題においては、線形の速度向上を得ることすら難しく、本稿のように32並列を用いても7倍程度しか速度向上が得られない結果となることがしばしばである。しかし、この速度向上によって実問題にもたらされる影響は決して小さくはない。将棋の場合で言えば、7倍速度向上した激指を元の激指と対局させると、勝率は7〜8割程度に達する。これは、人間の感覚では1段ほど棋力が向上したように見える勝率である。並列化による探索の高速化は、今後も重要な研究要素の一つであり続けると考えている。

◇ 参 考 文 献 ◇

[Brockington 97] Brockington, M. G.: *Asynchronous Parallel Game-Tree Search*, PhD thesis, Department of Computing Science, University of Alberta (1997)

[Feldmann 90] Feldmann, R., Monien, B., Mysliwietz, P., and Vornberger, O.: Distributed Game Tree Search, in Kumar, V., Gopalakrishnan, P. S., and Kanal, L. N. eds., *Parallel Algorithms for Machine Intelligence and Vision*, pp. 66–101, Springer-Verlag New York, Inc., New York, NY, USA (1990)

[Feldmann 93] Feldmann, R.: *Game Tree Search on Massively Parallel Systems*, PhD thesis, University of Paderborn (1993)

[Hyatt] Hyatt, R.: The DTS high-performance parallel tree search algorithm: http://www.cis.uab.edu/hyatt/search.html

[Kishimoto 02] Kishimoto, A. and Schaeffer, J.: Transposition Table Driven Work Scheduling in Distributed Game-Tree Search, in *Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, AI ’02, pp. 56–68, London, UK, UK (2002), Springer-Verlag

[Knuth 75] Knuth, D. E. and Moore, R. W.: An analysis of alpha-beta pruning, *Artificial Intelligence*, Vol. 6, No. 4, pp. 293 – 326 (1975)

[Marsland 82] Marsland, T. A. and Campbell, M.: Parallel Search of Strongly Ordered Game Trees, *ACM Computing Surveys*, Vol. 14, pp. 533–551 (1982)

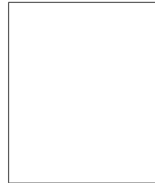
[Romstad] Romstad, T.: Stockfish – Open Source Chess Engine: http://www.stockfishchess.com/

[Tsuruoka 02] Tsuruoka, Y., Yokoyama, D., and Chikayama, T.: Game-tree Search Algorithm based on Realization Probability, *ICGA Journal*, Vol. 25, No. 3, pp. 145–152 (2002)

[Weill 96] Weill, J.-C.: The ABDADA Distributed Minimax Search Algorithm, in *Proceedings of the 1996 ACM 24th annual conference on Computer science*, CSC ’96, pp. 131–138, New York, NY, USA (1996), ACM

[激指] 激指開発チーム：将棋プログラム「激指」 : http://www.logos.ic.i.u-tokyo.ac.jp/˜gekisashi/

著 者 紹 介



横山 大作

2006年東京大学より博士号取得。博士(科学)。2002年より同大学新領域創成科学研究科助手などを経て、2009年より同大学生産技術研究所助教、現在に至る。並列プログラミング環境、ゲームプログラミングに関する研究に従事。