投機を用いた並列ゲーム木探索の効率化

浦　晃[†1]　横山大作[†2]　近山隆[†1]

従来の並列ゲーム木探索手法の多くは、プロセッサが数十という環境を想定しているため、計算量を抑制することに配慮するあまり並列度が低下し、多数のプロセッサを有効利用できない。本稿では、数百以上のプロセッサからなる環境において大きな速度向上を実現するために、必要なタスクの実行を妨げないようにスケジューリングしつつ、必要性が明らかでないタスクも投機実行することを提案する。提案手法を実装して評価したところ、タスクの粒度と優先度の設定が適切であれば、大きな速度向上が得られることがわかった。また、逐次探索プログラムとの対戦でも提案手法の優位性を示すことができた。

Improving Parallel Game Tree Search with Speculation

Akira URA,[†1] Daisaku YOKOYAMA[†2]
and Takashi CHIKAYAMA[†1]

Most of the conventional parallel game tree search methods try to prevent increased total computation in environments with dezens of processors, resulting in low parallelism, which cannot utilize a large number of processors effectively. In this paper, we propose a method to realize large speed-up in environments with hundreds of processors by executing speculative tasks, which may revealed to be unnecessary afterwords. These speculative tasks are controlled so as not to disturb mandatory tasks. Evaluation through implementing the proposed method shows high speed-up with appropriate granularity and priority settings. It also shows better performance than program with conventional method.

1. はじめに

コンピュータゲームプレイヤにおいてはゲーム木を深くまで探索すればするほど強くなると言われており、ゲーム木探索の速度は非常に重要である。複数のプロセッサを用いたゲーム木の並列探索は、ゲーム木探索の速度を向上させるための有力な手段として挙げられる。ところが、ゲーム木探索で広く用いられている $\alpha\beta$ 探索は、すでに行った探索の結果を用いて枝刈りを行うため、探索中の部分木に依存関係が存在することになり、並列化が難しい。逐次探索と同程度の枝刈りを行おうとすれば並列度が小さくなり、逆に並列度を大きくしようとすると、逐次探索では行えた枝刈りが行えず、無駄に探索するノード数が増えてしまう。並列探索を行うときは、使用する環境に応じて、このトレードオフの最適点を探すことになる。

†1 東京大学大学院工学系研究科
Graduate School of Engineering, The University of Tokyo
†2 東京大学生産技術研究所
Institute of Industrial Science, The University of Tokyo

これまで行われてきた並列ゲーム木探索の研究は、プロセッサが多くても数十程度の規模の環境で行われているものが大半であり、逐次探索に近い探索ノード数を保ったまま並列探索が行える YBWC (Young Brothers Wait Concept)[1] などの同期を用いた手法が一定の成功を収めている。しかし、数百から数千以上からなる多くのプロセッサからなる環境を利用するとなると、従来の手法では並列度が小さく、プロセッサのアイドル時間が長くなる。このような環境では、アイドル状態のプロセッサが存在するときに、探索ノード数を逐次探索より増やしてでも、将来探索される可能性のあるノードを事前に探索することで、全体の探索時間を短縮することができる。

本稿では、数百以上のプロセッサを用いてゲーム木探索を並列化する際に、全体の探索時間を短縮することを目的として、タスクの投機実行を考慮した優先度を用いるスケジューリング手法を提案する。この手法を実装し評価したところ、最終的に探索しなければならない探索を妨げないように、アイドル状態のプロセッサでタスクを投機的に実行することで、全体としての探索時間を短縮できた。

本稿の構成は以下の通りである。まず 2 章で関連研究について述べる。続く 3 章で提案手法について述べた後で、4 章で実際に行った実装について説明する。次に 5 章で行った評価実験の内容とその結果を示す。最後に 6 章でまとめと今後の課題を述べる。

2. 関連研究

2.1 YBWC

同期を用いる並列ゲーム木探索のアルゴリズムとして代表的なものは、YBWC (Young Brothers Wait Concept)[1] であり、現在広く用いられている。YBWC では、最初に探索する子ノード (最左の子ノード) の探索の結果が得られるまで待った後で、残りの子ノードの探索を並列に行う。常に最善手を最初に探索する理想的な場合には、探索ノード数が逐次探索と同じになる。ただし、すべてのノードで同期を行うと同期を行う箇所が増えてしまうため、実際には ALL ノードでは同期を行わないことも多い。

Feldmann[1] は、チェスプログラム ZugZwang を work stealing を用いて実装を行った。30MHz、20Mbps の T805 トランスピュータ 1024 台を用いて、逐次探索の時間が 10 万秒程度の規模の問題で、344 倍の速度向上比を実現している。

Campbell ら[2] はチェスプログラム DeepBlue を、200 万ノード/秒で探索できる専用のチェスチップをワーカに用いたマスタ・ワーカ方式で実装した。NegaScout アルゴリズム[3] の並列化で 480 台で 38 倍から 58 倍程度の速度向上比を達成している。

Kishimoto ら[4] は TDS (Transposition Table Driven Work Scheduling)[5] を用いて、MTD(f) アルゴリズム[6] を YBWC で並列化している。ゲームとしては Awari と Amazon を対象としている。CPU が 933MHz の Pentium III、ネットワークが 100Mbps の環境で実験を行い、逐次探索時間が 2100 秒程度のアワリで 21.8 倍、逐次探索時間が 1600 秒程度のアマゾンで 23.5 倍の速度向上比を 64 台で達成している。

Steenhuisen[7] も YBWC に TDS を用いてチェスプログラム DarkSight の実装を行っている。PVS アルゴリズム[8] を並列化しており、CPU が 1GHz の Pentium III、ネットワークが 1.2Gbps の環境で、32 台を用いて 5.7 倍の速度向上比が得られている。

YBWC では同期を行う場所を減らす必要性が主張されている。同期点において計算時間のかかる子ノードが存在すると、プロセッサがアイドル状態になるためである。また、YBWC では探索開始直後の並列度が小さいのも問題である。ALL ノードで子ノードを全て並列探索する改良も、同期を減らすためであるし、プロセッサのアイドル時間を減らすために、ルートノードでは子ノードはすべて並列探索するという改良も存在する[4]。理論的な計算では、プロセッサ数が多いときには、YBWC よりも非同期アルゴリズムの方が性能が良くなるという主張もある[9]。

同時に使用可能なプロセッサ数は現在でもますます多くなっている。そのため今後の並列ゲーム木探索は、無駄な探索を増加させてでも、同期を減らす方向に向かうと考えられる。

2.2 非同期型のアルゴリズム

基本的な非同期アルゴリズムとして UIDPABS (Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search)[10] がある。UIDPABS はルート局面の M 個の手を均等に K プロセッサに分配して、反復深化を用いて個別に探索させるアルゴリズムである。

非同期アルゴリズムとしてよく紹介されるものとして、APHID (Asynchronous Parallel Hierarchical Iterative Deepening)[9),11] がある。APHID はマスタ・ワーカ構成をとる。全体で深さ d の探索を実行するとき、マスタは深さ $d'(< d)$ の $\alpha\beta$ 探索を繰り返し行う。その際、訪問したリーフノードをタスクとしてワーカに割り当てていく。ワーカはマスタから割り当てられたタスクの中から優先度の高い順に選び、反復深化の 1 反復を行っていく。マスタはワーカが十分な深さ $(d - d')$ だけ探索した結果はそのまま用いるが、まだ深さが十分でない場合は推定値を用いる。マスタはルートノードを評価するのに推定値を一つも用いなかったら探索を終了する。APHID は 4 つの既存のプログラムに適用して性能評価が行われており、64 台での速度向上比は、チェスプログラムの Crafty と The Turk でそれぞれ 18 倍、16 倍、チェッカープログラムの Chinook で 14 倍、オセロプログラムの Keyano で 37 倍が報告されている。問題規模はチェス、チェッカー、オセロでそれぞれ並列化して、180 秒、120 秒、60 秒に収まる程度である。

田中ら[12] は将棋プログラム GPS 将棋の並列化をマスタ・ワーカ方式で行っている。ルートノードから左側の子ノードほど多くのワーカを割り当てていきながら、ワーカが 1 つになるまでゲーム木を展開していき、各ワーカが探索を担当することになるノードを指定する。ワーカは時間いっぱいに探索を実行し、最後にマスタが結果をまとめる。性能評価としては、634 コアで 8 コアマシンでのスレッド並列化に、バグを除いて 13 戦全勝という報告がされている。

3. 提 案 手 法

本研究では無駄な探索が少ない YBWC を基本として、YBWC では逐次化していたところを待たずに並列実行を行う。本稿ではこのことを投機実行と呼ぶ。このとき投機実行で増加したタスクの実行順序を、YBWC で探索を行う順序に近づけるように優先度をつけ、YBWC で実行するタスクを妨げないようにスケジューリングすることを提案する。

本研究では、すべてのタスクを「必須」と「投機」の二種類に分類する。そのために、探索中のゲーム木に含まれる全てのノードについて、「必須」か「投機」かのラベルを付ける。

「必須」 ある時点において、YBWC では探索中のゲーム木に含まれているノード

「投機」 投機実行を行っているときに、探索中のゲーム木には含まれているが、「必須」ではないノード

すべてのノードを「必須」と「投機」に分類すると、それに応じてタスクも「必須」と「投機」に分類される。各タスクには優先度がつけられており、「必須」のタスクも「投機」のタスクもその優先度の順に実行される。

このとき、「必須」のタスクはどの「投機」のタスクよりも優先度が高い。そのため、提案手法では、「必須」のタスクが生じたら「投機」のタスクを中断することになってでも、「必須」のタスクを優先して実行する。タスクの中断と再開は、ゲームにおいてはトランスポジションテーブルを用いることで実現できるため、小さなオーバーヘッドで行える。なお、「投機」のタスクは探索が進むことで、必要ならば「必須」となる。つまり提案手法では、探索が進むとともに変化するような、動的なタスクの優先度を用いていることになる。

提案手法を用いることで、YBWC に加えてタスクを先立って投機的に実行しつつも、YBWC において実行しなければならないタスクの実行を妨げることはない。

3.1 提案手法の詳細

実装に多重反復深化を用いたときの、提案手法の詳細について述べる。

3.1.1 「必須」と「投機」のラベルづけ

「必須」と「投機」の各ノードへのラベルは以下のようにつけられる。

- ルートノードは「必須」
- 「投機」のノードの子ノードは全て「投機」
- 「必須」のノードの子ノードについて、
  - 浅い探索が終わっていないならば、浅い探索だけが「必須」、残りは「投機」
  - 浅い探索が終わり、最左の子ノードの探索が終わっていないならば、最左が「必須」、残りは「投機」
  - 最左の子ノードの探索が終わっているならば、残りの子ノードは全て「必須」

親ノードのラベルが変更されたら、子孫ノードのラベルを変更する必要があることに注意する。

3.1.2 投機実行を行う箇所

投機実行する箇所として、本研究では PV ノード、ALL ノード、CUT ノードの 3 つから選んで行った。これらのノードの分類は次のように決定される。

- ルートノードは PV ノード
- PV ノードの最左の子ノードは PV ノード、残りの子ノードは CUT ノード
- CUT ノードの最左の子ノードは ALL ノード、残りの子ノードは CUT ノード
- ALL ノードの子ノードは全て CUT ノード

投機実行時は浅い探索が終わるまでは最左が分からないので、浅い探索以外の子ノードは CUT ノードとして展開する。そして浅い探索が終わったときに、最左だけ展開し直す。なお、CUT ノードで投機実行するときは、全ての CUT ノードで投機実行すると、ゲーム木が大きくなりすぎてしまうので、今回はルート局面の直下の子ノードの CUT ノードのみに投機実行を適用することにした。

3.2 タスクの優先度

タスクの優先度には signature を用いた[7]。signature はあるノードに到達するまでに何番目の子ノードを通ってきたかを表現するものである。例えばルートノードから 2 番目、3 番目、1 番目の順に手を選んで到達できるノードの signature は 2,3,1 と表現できる。これにより、ゲーム木の左側ほど高くなる優先度を用いることができる。なお、浅い探索は他の子ノードよりも左側にある子ノードと判断される。

4. 実　　装

今回は実装としてマスタ・ワーカ方式を用いた。概要を図 1 に示す。 探索全体の流れは次のようになる。探索する深さを d とする。マスタはルートノードを含むゲーム木をある深さ $d'(< d)$ まで展開し、メモリ上に保持する。本稿では、マスタのゲーム木のリーフノード以下の部分木を (深さ $d - d'$ を粒度とした) タスクと呼ぶ。マスタはタスクを優先度の順にワーカに送信する。そして、ワーカからの結果を受信して、

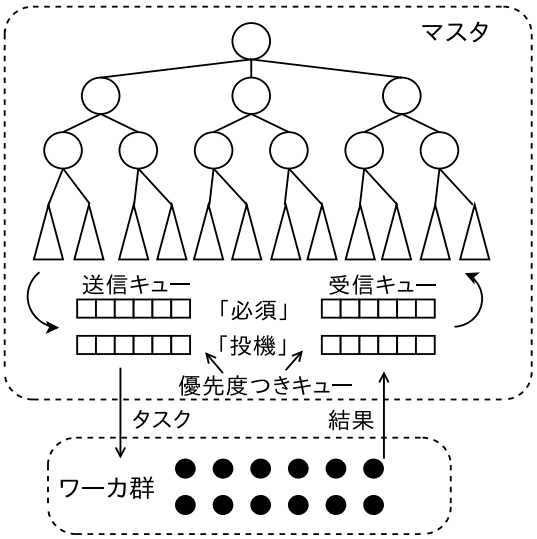

図 1 実装の概要

それらの結果を優先度の順に処理し、ゲーム木を更新する。結果を送ってきたワーカには次のタスクを送信する。

対象としたゲームは将棋である。将棋プログラムである激指を用いて C++言語で実装した。ワーカは激指の探索関数を呼んで探索を行う。マスタには激指の関数を用いて、多重反復深化と実現確率[13] を用いた $\alpha\beta$ 探索を実装したものを用いた。

4.1 マスタの設定

4.1.1 多重反復深化

マスタでは多重反復深化を行う。つまり全てのノードにおいてまず浅い探索を行い、その結果よって最左の子ノードを決定する。そのため YBWC は次のようになる。

(i) まず浅い探索 (子ノードとみなす) を行う
(ii) 浅い探索の結果が得られたら、最左の子ノードの探索を行う
(iii) その後、残りの子ノードを並列探索する

反復深化中のゲーム木の様子を図 2 に示す。本研究では浅い探索も子ノードとみなすので、投機実行をした場合、普通の子ノードの探索と同時に、反復深化の各反復もすべて並列に実行されることになる。

4.1.2 実現確率における再探索

本研究ではマスタは実現確率を用いている。実現確率は対数をとって絶対値をとることにより深さとみなすことができるので、本稿では深さという用語で統一する。

子ノードを全て並列探索するのであれば、浅い探索は本来は必要ない。今回の実装では、最左は実現確率を用いた場合よりも深く探索するようにしているため、最左を求めるために浅い探索を行う。投機実行をしているならば、浅い探索が終わったときに最左を再探索することになる。本来ならば、残りの子ノードについても、α 更新を起こしたときには再探索が必要である。しかし、今回の実装ではそのときの再探索は未実装であり、今後の課題である。

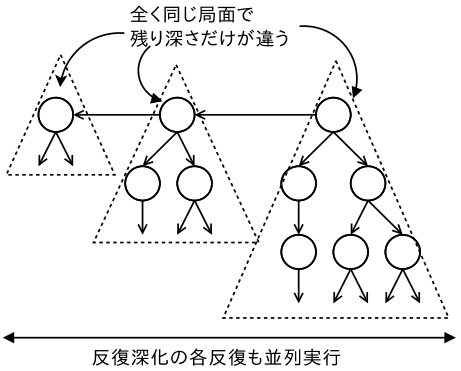

図 2 反復深化の扱い

4.2 マスタの詳細

マスタは送信スレッド、受信スレッド、処理スレッドの 3 スレッドからなる。探索開始時に、ゲーム木を投機実行も含めて展開できるだけ展開し、リーフノードをタスクとして送信キューに入れた後に、各スレッドの処理が始まる。

送信スレッドは送信キューに入っているタスクを取り出し、優先度の順にワーカに送信することを繰り返す。タスクは文字列に変換してから送信する。

受信スレッドはワーカから受信した文字列を結果としてマスタで処理するデータ構造に変換し、受信キューに入れることを繰り返す。

処理スレッドはゲーム木を管理しており、受信キューから結果を取り出し、処理することを繰り返す。新しくノードを展開する必要があれば展開できるだけ展開する。もし枝刈りが発生したならばゲーム木からノードを削除する。

探索窓の更新や枝刈りはマスタ内で全て行う。これらをすでに送ってしまったタスクについても行う必要があるときは、そのことをワーカに通知する。

マスタのトランスポジションテーブルは新しくノードを作るときに参照する。無駄な探索を減らすために、すでに探索が始まっている局面のエントリを参照した場合、探索窓が自分の方が広ければ、その探索を待つということを行う必要があるが、まだ実装を行っていない。ただし、これでは投機実行時に、テーブルを参照する機会がほとんどなくなってしまうため、ノードに結果を伝えるときには毎回テーブルを参照する実装

とした。なお、ワーカとのトランスポジションテーブルに関する情報のやりとりは行わないこととした。

4.3 ワーカの詳細

ワーカは基本的にマスタからタスクを受信して探索を行い、結果を送信することを繰り返す。

トランスポジションテーブルは各ワーカが独自のものを用いる。今回はタスク毎にテーブルをクリアするようにした。テーブルの内容に関してマスタや他のワーカとのやりとりは行わない。

現在の実装ではワーカは最大でも同時に二つのタスクしか保持しない。なぜならマスタがアイドル状態でないワーカにタスクを送ってくるのは、送るタスクが「必須」で、ワーカが「投機」のタスクを実行中のときに限るからである。このとき、ワーカはいったん「投機」の探索を中止し、「必須」のタスクを行ってから、元の「投機」のタスクを再実行する。この一連の流れの間にトランスポジションテーブルをクリアしなければ、再実行は途中から探索を再開することとほぼ同じである。

ワーカはタスク以外にも、タスクの「必須」への変更通知、タスクの探索窓の更新通知、タスクの枝刈り通知を受け取る。ワーカはこれらの通知を受け取ったら、それが現在ワーカが所有しているものかを確認し、通知された各処理を行う。このうち、窓更新はいったん探索を中止し、テーブルをクリアせずに再探索することで行った。

5. 評　　価

5.1 評 価 方 法

まず、6 つの局面を実際に並列探索させることで、提案手法の評価を行った。速度向上比を計算するための 1 台の計算時間としては、指定された粒度より残り深さが小さくなると、激指の探索関数を呼び出す、深さ優先探索を用いた逐次の $\alpha\beta$ アルゴリズムを実装して用いた。本稿では以後このプログラムのことを、1 並列プログラムと呼ぶことにする。このプログラムで今回用いた 6 局面を探索させたときの時間を図 3 に示す。粒度 (granularity) は激指の探索関数 (並列プログラムではワーカ) が探索する深さである。

並列探索はクラスタ環境の 1 ノードにマスタを 1 つ、別のノード 1 つにつきワーカを 8 つ起動して行った。並列探索は最大で 512 ワーカまでで、各粒度、各局面について 5 回ずつ行った。用いたクラスタ環境は以下の通りである。

- 1 ノードあたり 8 コア ×2 スレッド
- CPU : Xeon E5530 2.40GHz

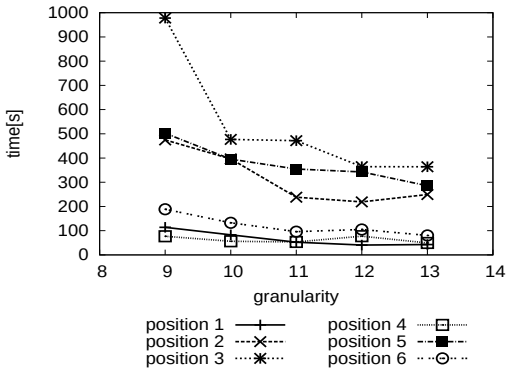

図 3 1 並列プログラムでの実行時間

- 1 ノードあたりのメモリ : 24GB
- ネットワーク : 10Gbps Ethernet
- OS : Linux 2.6.26-2
- コンパイラ : gcc 4.3.2

次に提案手法を用いて並列化したプログラムが、どの程度強さが向上したのかを、逐次プログラムと対戦させることで調べた。実験には InTrigger[14] の二つのクラスタ (CPU が 2.33GHz と 2.40GHz) を用いた。ワーカ数は 120 とした。対戦には乱数で選ばれた定跡を先手後手あわせて 30 手までは使えるだけ使った。なお、250 手で決着がつかない場合は引き分けとし、勝率の計算からは除いた。

5.2 投機実行の評価

投機実行によって並列探索の性能がどのように変化するのかを示す。図 4 に投機実行しない場合の速度向上比を示す。次に投機実行をしない場合と比較して、投機実行をすることで速度が何倍になったのかを粒度 12 と粒度 9 について、図 5、6 にそれぞれ示す。図 5 と図 6 では、ALL ノードで投機実行した場合 (ALL)、PV ノードで投機実行した場合 (PV)、PV ノードと ALL ノードで投機実行した場合 (PV&ALL) のそれぞれについて示してある。これらの投機実行によって生じたタスクは「投機」としてスケジューリングを行っている。さらに、PV ノードと ALL ノードで投機実行してできたタスクを「必須」とし、CUT ノードでも投機実行 (ただしルート局面の直下の CUT ノードのみ) し、それによって生じたタスクは「投機」として、スケジューリングを行った場合 (PV&ALL&CUT) も同時に示してある。このようにしたのは、PV ノードと ALL ノードでの投機実行と CUT ノードでの投機実行を比べると、明らかに後者が無駄になりやすく、この二つの間での投機実行には明確な優先度をつける必要があると判断したためである。

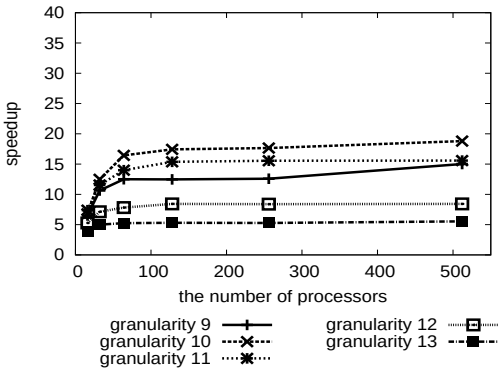

図 4 YBWC の速度向上比

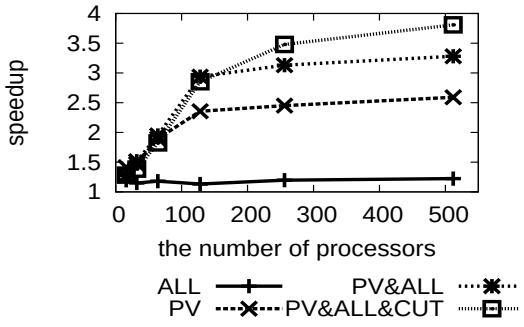

図 5 YBWC を基準とした投機実行時の速度向上比 (粒度 12)

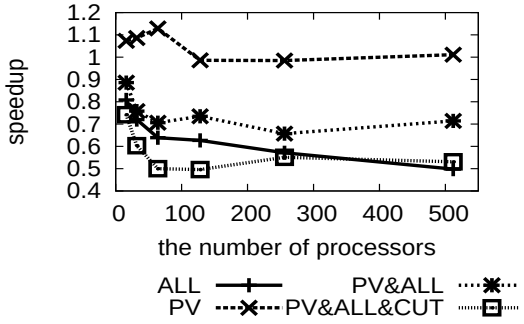

図 6 YBWC を基準とした投機実行時の速度向上比 (粒度 9)

ALL ノードだけの投機実行は影響が小さいことがわかる。また、粒度が大きいときは投機実行をするほど性能が改善するのに対し、粒度が小さいときは性能が悪化していることがわかる。粒度が小さいときに性能が悪化するのは、タスクが不足していないのにも関わらず、投機実行によって必要以上に大きなゲーム木を展開することになり、ゲーム木の管理コストが大きくなったためである。

今回行った実験の中で、性能が良かったのは、粒度 11 で PV ノードと ALL ノードで投機実行した場合か、粒度 12 で CUT ノードまで投機実行した場合で、速度向上比は 30 倍程度であった。

5.3 「必須」と「投機」の 2 レベルスケジューリングの評価

タスクを「必須」と「投機」に分けて 2 レベルスケジューリングを行ったことでの性能変化を見る。PV ノードと ALL ノードで投機実行したとき、タスクをすべて「必須」として扱う 1 レベルスケジューリングを行ったときのワーカ数 512 のときの速度向上比を、投機実行のタスクを「投機」とする 2 レベルスケジューリングの場合と併せて図 7 に示す。少しだけだが、1 レベルスケジューリングのほうが性能が良かった。PV ノードと ALL ノードは、いつかは実行する必要があるので、2 レベルスケジューリングの効果が弱いのに対し、「投機」のタスクを「必須」にするのに余計な処理が入ってしまうからである

次に、PV ノードと ALL ノードで投機実行して生じたタスクは「必須」とした上で、CUT ノードで投機実行して生じたタスクを「必須」とした 1 レベルスケジューリングの場合と、「投機」とした 2 レベルスケジューリング場合の、ワーカ数 512 のときの速度向上比の違いを図 8 に示す。この場合は、2 レベルスケジューリングを行った方が性能が良いことが分かった。CUT ノードは PV ノードや ALL ノードと違って、子ノードを先立って探索しても無駄になってしまうことが多いので、優先度を低くしたほうがよいためである。

5.4 ワーカの負荷

各ワーカにどれだけの負荷がかかっていたかを検討する。粒度が 11 のとき、全実行時間に対する、ワーカが実際に探索を行っていた時間の割合 (worker load) を図 9 に示す。投機実行をしない場合と、各ノードで投機実行した場合について示してある。さらに、ワーカ数が 512 のとき、実際に探索を行っていたワーカの数を、ある局面の粒度 11 の場合について、投機実行しないとき、PV ノードと ALL ノードで投機実行したとき、加えて CUT ノードでも投機実行したときの 3 つの場合について図 10、11、12 に示す。粒度が 11 のときは、投機実行をするほどワーカの負荷が上がっていることがわかる。

なお、粒度を小さくしてタスクを増やしても、ワーカの負荷は大きくならなかった。現在の実装では、ワーカが結果を送信して、新しいタスクを受信するまでに 1ms のオーバーヘッドがあり、各粒度におけるタスクの大きさは表 1 のようになっているため、粒度を小さくするとオーバーヘッドの大きさが無視できなくなるからである。

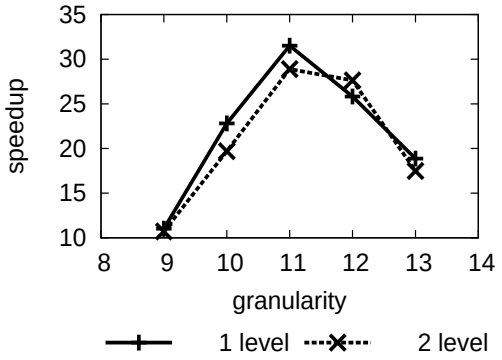

図 7 PV ノードと ALL ノードで投機実行したときの 2 レベルスケジューリングの効果

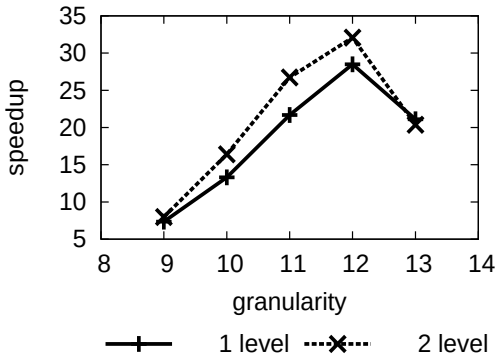

図 8 CUT ノードで投機実行したときの 2 レベルスケジューリングの効果

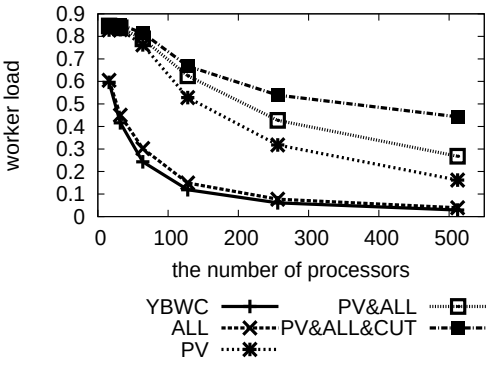

図 9 粒度が 11 のときのワーカの負荷

5.5 対戦結果

投機実行なしの YBWC を用いた並列プログラム、PV ノードと ALL ノードで投機実行した並列プログラムを各粒度について、100 局ずつ逐次の激指と 1 手 3 秒で対戦させたときの勝率を表 2 に示す。最も勝率が高くなったのは、粒度 11 の投機実行ありの場合で 83.8%だった。

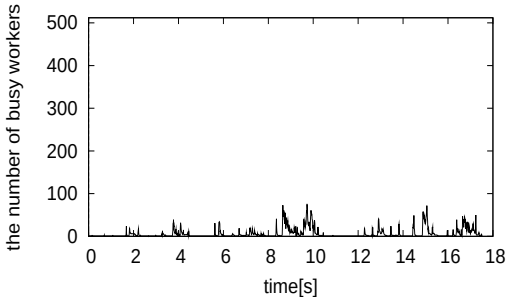

図 10 投機実行しないときの各時刻における探索実行中のワーカ数

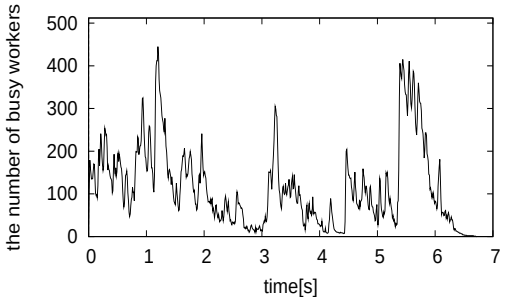

図 11 PV ノードと ALL ノードで投機実行をした場合の、各時刻における探索実行中のワーカ数

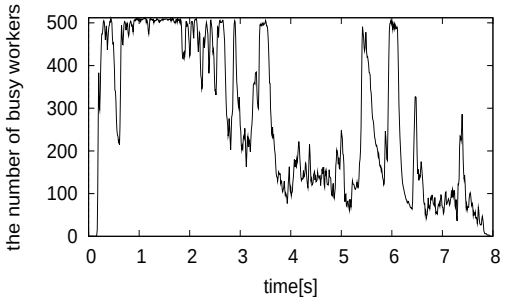

図 12 CUT ノードでも投機実行をした場合の、各時刻における探索実行中のワーカ数

実際にこれが逐次プログラムで何倍時間をかけたときと同程度になるのかを調べるために、1 並列プログラムと逐次の激指を 1 手あたりの時間を変えて、それぞれ 1 手 3 秒の逐次の激指と対局させた。粒度 11 の 1 並列プログラムと 1 手 3 秒の逐次の激指との対戦結果を表 3 に、逐次の激指どうしの対戦結果を表 4 に示す。それぞれ 200 局ずつ行っている。 表 2、3、4 を比較すると、今回の並列プログラムは 120 並列で、1 並列プログラムに対しては 10 倍以上の速度で、激指に対しては 2.5 倍程度の速度だったと言える。実際に 1 手 3 秒の 120 台並列プログラムを 1 手 8 秒の逐次

表 1 各粒度のタスクの平均実行時間 [ms]

粒度 9	粒度 10	粒度 11	粒度 12	粒度 13
1.0	2.6	6.8	17	43

の激指と 100 局対戦させたところ、並列プログラムが 51 勝 49 敗と互角であった。

6. おわりに

本稿ではタスクを先立って投機実行させ、タスクのスケジューリングを動的に変化する「必須」と「投機」の 2 種類の優先度で管理し、優先度の高いタスクの邪魔にならないように、実行中のタスクを中断させてでも優先度の高いタスクを実行する手法を提案した。評価実験では、PV ノードと ALL ノードで投機実行することで強くなることを確認した。また、CUT ノードで投機実行したときは、「必須」と「投機」の 2 つの優先度で管理する提案手法の優位性を示すことができた。しかし、PV ノードと ALL ノードの投機実行では、この優位性はなかった。また、タスクが十分足りているのに、必要以上に投機実行すると、ゲーム木が大きくなりすぎてしまい、その管理コストが問題となることがわかった。

今後の課題としてはまず、マスタ・ワーカ間でのタスクのやりとりのオーバーヘッドを小さくする必要がある。そのためには実装の最適化はもちろん必要であるが、手法としてはワーカに複数のタスクを常に持たせておけばよい。これにより粒度が小さいときの性能向上が期待できる。次に、必要以上に大きなゲーム木を管理する必要がないように、投機実行はタスクが枯渇する危険性があるときのみに行うように変更するべきである。また、強いプレイヤにするためには、実現確率を用いたときの再探索を実装しなければならない。ただし、再探索を行うとそこがボトルネックとなるので、適切な対応策を講じる必要がある。

表 2 ワーカ数 120 の並列プログラムの、逐次の激指に対する勝率 (1 手 3 秒)

	粒度 8	粒度 9	粒度 10	粒度 11
投機実行なし	37.8	63.3	48.0	56.0
投機実行あり	45.8	65.3	71.0	83.8

表 3 1 手 n 秒の 1 並列プログラムの、1 手 3 秒の逐次の激指に対する勝率

1 手 3 秒	1 手 15 秒	1 手 21 秒	1 手 30 秒
33.2	72.5	77.3	77.0

表 4 1 手 n 秒の逐次の激指の、1 手 3 秒の逐次の激指に対する勝率

1 手 1 秒	1 手 6 秒	1 手 10 秒	1 手 30 秒
19.8	75.9	87.2	94.7

謝辞 本研究の一部は文部科学省科学研究費補助金特定領域研究「情報爆発に対応する高度にスケーラブルなソフトウェア構成基盤」の助成を得て行われた。

参考文献

1) R.Feldmann. *Game Tree Search on Massively Parallel Systems*. PhD thesis, University of Paderborn, 1993.
2) M.Campbell, A.J. Hoane, etal. Deep blue. *Artificial Intelligence*, Vol. 134, No. 1-2, pp. 57–83, 2002.
3) A. Reinefeld. An improvement to the scout tree-search algorithm. *ICCA Journal*, 6(4):4-14, 1983.
4) A. Kishimoto and J. Schaeffer. Distributed game-tree search using transposition table driven work scheduling. In *ICPP'02*, pp. 323–330. IEEE, 2002.
5) J.W Romein, A. Plaat, H.E. Bal, and J. Schaeffer. Transposition table driven work scheduling in distributed search. In *AAAI'99*, pp. 725–731, 1999.
6) A. Plaat. *Research Re:Search & Re-search*. PhD thesis, Erasmus University, 1996.
7) J.R. Steenhuisen. Transposition-driven scheduling in parallel two-player state-space search. Master's thesis, Delft University of Technology, 2005.
8) T.A. Marsland. Relative efficiency of alpha-beta implementations. In *IJCAI-83*, Vol.2, pp. 763–766, 1983.
9) M.G. Brockington. *Asynchronous Parallel Game-Tree Search*. PhD thesis, University of Alberta, 1998.
10) M.Newborn. Unsynchronized iteratively deepening parallel alpha-beta search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol.10, No.5, pp. 687–694, 1988.
11) M.G. Brockington and J.Schaeffer. APHID: Asynchronous parallel game-tree search. *Journal of Parallel and Distributed Computing*, Vol.60, No.2, pp. 247–273, 2000.
12) 田中哲朗, 金子知適. 将棋プログラムの大規模並列実行. 情報処理学会研究報告, Vol. 2010-GI-24, No.2, 2010.
13) Y.Tsuruoka, D.Yokoyama, and T.Chikayama. Game-tree search algorithm based on realization probability. *ICGA Journal*, Vol.25, No.3, pp. 146–153, 2002.
14) InTrigger プラットフォーム, `http://www.intrigger.jp/`.