連載講座: 高生産並列言語を使いこなす(4)
ゲーム木探索の並列化

田浦健次朗
東京大学大学院情報理工学系研究科, 情報基盤センター

目次

1 準備

1.1 問題の定義

前号に続いて, オセロゲームを題材とした詰み探索を取り上げる. 問題は, 与えられた盤面 g に対して以下の再帰式で定義される「評価値」$E(g)$ を求めることであった.

$$E(g) \quad = \quad \begin{cases} \text{手番側のコマ数 - 相手側のコマ数} & (g \text{ が最終局面のとき}) \\ \max\limits_{h:g\to h}(-E(h)) & (g \text{ が最終局面でないとき}) \end{cases}$$

$g \to h$ は g から一手で h へ遷移できるという関係を表したものである. $g \to h$ を親子関係とした木構造をゲーム木と呼ぶ. 素朴なアルゴリズムでは, g を根としたゲーム木のノードをすべて訪問して, 上記の評価値を得る.

オセロ以外のゲームでも, g が最終局面の時の評価値を適切に変更すれば, 上記の定義は共通である. また g が最終局面でなくても, ある深さで適切な評価関数を用いた評価を行うことにより, 詰み探索以外にも適用できる.

1.2 $\alpha\beta$ 法

この評価を高速化する枝刈り手法である $\alpha\beta$ 法を C 言語で書いたものが以下である.

```
 1: int eval(game_state_t g, int alpha, int beta) {
 2:   int moves[MAXEMPTIES];
 3:   int n_moves = gen_moves(g, moves);
 4:   if (n_moves == 0) {
 5:     return g->disc_diff;
 6:   } else {
 7:     int i;
 8:     for (i = 0; i < n_moves; i++) {
 9:       int e = -move_and_eval(*g, i, moves[i], -beta, -alpha);
10:       if (e > alpha) {
11:         alpha = e;
12:         if (e >= beta) break;
13:       }
14:     }
15:     return alpha;
16:   }
17: }
```

引数として盤面 g の他に, **探索窓**と呼ばれる区間 $[\alpha, \beta]$ を受け取る. この関数は, $\alpha < E(g) < \beta$ であれば $E(g)$ を返すが, $E(g) \leq \alpha$ であれば, ある α 以下の値を, $E(g) \geq \beta$ であれば, ある β 以上の値を返す. 後者を β **カット**と呼び, 実際にそれが起きているのは 12 行目である.

今回はこの $\alpha\beta$ 法の並列化を, タスク並列をサポートする各種処理系を用いて行う.

2 $\alpha\beta$ 法の並列化

2.1 概要

$\alpha\beta$ 法の並列化の基本戦略は, 複数の再帰呼び出しを並列に行うことである. つまり, 上のプログラム 8-13 行目で行われているループの並列化に相当する. しかし $\alpha\beta$ 法を, 訪問ノード数を保ったまま並列化することは簡単ではない.

まず表面的には,

- ループの途中までの結果により, β カットがおこり, ループが途中で終了する可能性がある (12 行目),
- ある繰り返しで用いられる探索窓 (のうちの, $-\alpha$) がそれ以前の繰り返しの結果に依存している (9 行目),,

という 2 点にある. それらは両者とも $\alpha\beta$ 法の強力な枝刈り手法を実現するための, 本質的な依存関係である. 一方で, 並列化を可能にする以下の性質もある.

- β カットを行わなくても, 計算量 ($\approx$ 訪問ノード数) は大きく増大するものの, 計算結果には影響しない,
- 引数として渡す探索窓 $[-\beta, -\alpha]$ についても, 逐次的に行った場合より広い範囲を渡しても—すなわち, より小さい α を用いて計算しても—結果には影響しない.

これは, $\alpha\beta$ 法がそもそも素朴な探索法 (g を頂点としたゲーム木のノードをすべて訪問する) を, 結果を保ったまま高速化したものであることを考えれば, 当然の性質である. $\alpha\beta$ 法の並列化では, この自由度を用いて訪問ノード数の増大を抑えつつ並列化を行う. 以降で具体的な方法について, 特定の言語には依存しない言葉で述べる. ここで述べた方法は概ね, Cilk によるチェスの実装である [1, 6, 5] などで述べられている方法を元にしている.

2.2 Young Brothers Wait Concept

まず基本的な方法として, Young Brothers Wait Concept (YBWC) と呼ばれる方式がある [2]. それは, あるノード g の子ノード $h_0, h_1, h_2, \ldots$ の評価値を求める際, 最初の子ノード h_0 の評価が終わってから, 以降のノード $h_1, h_2, \ldots$ を (並列に) 評価するというものである. つまり, 以降のノードに渡される探索窓は, h_0 の結果だけを反映したものになる. その根拠は以下のとおりである.

- もし指し手の順序付け (move ordering) により, h_0 が常に最善—つまり, $\forall i(-E(-h_0) \geq -E(-h_i))$—であったとすると, 11 行目での α 値の更新は行われない. 従って, $h_1, h_2, \ldots$ に渡される探索窓は逐次計算におけるそれと等しい.
- そしてもちろん, 逐次計算で h_0 終了時に β カットが行われないならば, 以降 $h_1, h_2, \ldots$ 終了時にも β カットは発生しない.

つまり, 最善の指し手を正確に選べたとするならば, この並列化によって訪問ノード数は増加しない.

もちろん実際には最善の指し手を常に正確に選べるわけではない. 簡単な亜種として, 最初のいくつかの子ノードを逐次的に評価するという方式も考えられる.

YBWC によって, 並列度への影響はどのようになるだろうか? 見積りのための仮定として, ゲーム木が深さ l の完全 d 分木であり, 最善の指し手を常に選べたとする. 前号で述べたとおり総計算量 (≈ 訪問ノード数)

$$\Theta(d^{l/2})$$

であった. 無限個のプロセッサを仮定したときの実行時間に相当する, **クリティカルパス長**は,

- すべてのノードを並列に評価した場合は $\Theta(dl)$ である.

  再帰呼び出しをすべて並列に行うから, 木の段数 l に比例する. d は, 再帰呼び出しの生成やその中の最大値の取り出しを逐次的に行っているからで, わずかの変更で $\Theta(l \log d)$ に減らすことができるが実践的にはほとんど違いはない.

- 一方 YBWC を用いた場合, 各ノードで h_0 とそれ以外の再帰呼び出しが逐次的に行われるので, $\Theta(d2^{l/2})$ となる.

- 一般に最初の w 個の子ノードを逐次的に評価すれば, $\Theta(d(w+1)^{l/2})$ となる. この場合, その w の中に最善手が含まれていれば, 訪問ノード数は増加しない.

すべてのノードを並列に評価した場合と比べれば, $w \geq 1$ とするとクリティカルパス長が指数関数的に増えており, 並列実行時の台数効果が制限される. しかし, 総計算量 $\Theta(d^{l/2})$ との比は, $d > w+1$ である限りなお指数関数的である. 従って十分手の数が多い—≈ 空きマス数が多い—状態からの探索では, YBWC ($w = 1$ に相当) や, $w = 2$ 程度では大した妨げにはならないであろうと予想される.

2.3 段数による逐次化

あるノードの子ノードの評価を並列に行って, それぞれの子ノードの中でもさらに並列に子ノードの評価を並列に行って, ... という並列再帰呼び出しを繰り返していくと, たちまち多数の並列タスクが発生する. Cilk, TBB, OpenMP などのタスク並列機能は, まさにこのような計算を少ないオーバーヘッドで行うことを目標に設計されているが, それでもゲーム木の末端までタスクを作り続ける意味はほとんどない. さらに, 前節で議論したとおり, 並列化によって β カットの機会を失う, 探索窓が逐次の時よりも広くなる, など, 訪問ノード数の増大が避けられない. $\alpha\beta$ 法においてはタスク生成のオーバーヘッドよりもこちらの方が深刻である.

そのため, 再帰呼び出しの並列化はある段数までで打ち切り, 以降は逐次的に行うという, 簡単な最適化が考えられる.

2.4 適応的な待機

YBWC やその亜種として, ある w を固定して, 最初の w 個の子ノードの評価は逐次的に行う, という方法を述べた. 言い換えれば最初の w 個の子ノードの評価結果は以降の探索窓に反映するし, 最初の w 個の子ノードを評価した結果, β カットが起きれば, そこで探索は終了する.

それ以降の子ノードについては,

- 逐次よりも広い探索窓での実行

- β カットを行わない実行

をすることになり, 訪問ノード数の増大と引換えに並列実行を行う.

しかし「並列実行」と言ってもプログラムの字面上, 並列化されている—別々のタスクで実行するよう記述されている—ことと, 実際に別々のプロセッサで実行されていることとは別の事象である. つまり, 子ノード $h_1, h_2, \ldots$ の評価が, プログラムの字面上別々のタスクで実行されていたとしても, 実行時のスケジューリング, プロセッサ数などとの兼ね合いによって, どこが実際に同時に評価されるかはわからない. もともとプロセッサ数に比べて膨大な数のタスクを作り, それを負荷分散するのがタスク並列処理系であった. そこでは並列実行可能なタスクであっても, 多くのタスクが同一プロセッサ上で実行される. 後に述べるとおり, 多くのタスク並列処理系では, 並列再帰呼び出しであってもある程度, 逐次の実行順序に沿ったスケジューリングを行うので, この傾向はますます強くなる (詳細は処理系毎に異なり, 次節以降で述べる).

そこでそのような実行時の情報に基づいて, 実際に逐次実行された部分については, それを用いて β カットやこれから作られるタスクの探索窓への反映を行いたい. このような適応的な待機が実際にはどのように記述できるのかは, 処理系毎に異なる. 具体的な説明は 3-5 節で行う.

2.5 強制終了

訪問ノード数の増大をさらに抑えるための方法として, ある盤面 g の子ノードとして実行されているノード h が一つ終了するごとに, β カットが起きるかどうかを判定し, もし起きるならば, **すでに実行中の** g の子ノードの探索も強制的に打ち切ってしまう, という方法も考えられる.

以降の節で, 共有メモリ上で動的負荷分散を行うタスク並列の処理系である, Cilk, OpenMP, TBB を用いた並列化について具体的な記述方法を説明する.

3 Cilk による並列化

3.1 基本

Cilk では, spawn 構文を用いて関数呼び出しを非同期に行うことが出来, sync 構文を用いて, その関数内で行われた spawn すべての終了を待つことができる. まず説明のために, 訪問ノード数の増大を度外視してすべての子ノードを並列に評価するプログラムは以下のように簡単に書ける.

```
int eval(game_state_t g, int alpha, int beta) {
  int moves[MAXEMPTIES];
  int child_results[MAXEMPTIES];
  int n_moves = gen_moves(g, moves);
  if (n_moves == 0) {
    return g->disc_diff;
  } else {
    int i;
    for (i = 0; i < n_moves; i++) {
      child_results[i] = spawn move_and_eval(*g, i, moves[i], -beta, -alpha);
    }
    sync;
    for (i = 0; i < n_moves; i++) {
      int e = -child_results[i];
```

```
      if (e > alpha) {
        alpha = e;
        if (e >= beta) break;
      }
    }
    return alpha;
  }
}
```

3.2 YBWC

これに YBWC を加えるには, ループの最初の繰り返しだけ, sync およびそれ以降の処理を加えれば良い. 以下では一般化して, パラメータ w を受け取り最初の w 個の子ノードを逐次的に評価する.

```
 1: int eval(game_state_t g, int alpha, int beta, int w) {
 2:   int moves[MAXEMPTIES];
 3:   int child_results[MAXEMPTIES];
 4:   int n_moves = gen_moves(g, moves);
 5:   if (n_moves == 0) {
 6:     return g->disc_diff;
 7:   } else {
 8:     int i;
 9:     for (i = 0; i < n_moves; i++) child_results[i] = INF;
10:     for (i = 0; i < n_moves; i++) {
11:       child_results[i] = spawn move_and_eval(*g, i, moves[i], -beta, -alpha);
12:       if (i < w) { /* YBWC : 最初の w 個までは逐次的に */
13:         sync;
14:         int e = -child_results[i];
15:         if (e > alpha) {
16:           alpha = e;
17:           if (e >= beta) break;
18:         }
19:       }
20:     }
21:     sync;
22:     for (i = 0; i < n_moves; i++) {
23:       int e = -child_results[i];
24:       if (e > alpha) {
25:         alpha = e;
26:         if (e >= beta) break;
27:       }
28:     }
29:     return alpha;
30:   }
```

```
31: }
```

3.3 深さによる逐次化

コードの重複を厭わなければ実際の記述は非常に簡単である. eval 関数の先頭で, 再帰呼び出しの深さを検査し, ある程度深かったら, 別途書かれた逐次コードを呼び出すだけの処理に変更すれば良い. これは他の処理系の場合でも同様であるので, 以下省略する.

3.4 適応的な待機

2.4 節で述べた適応的な待機を Cilk の言葉で述べれば以下のようになる. まず基本的な観測として, プログラムの表面上 spawn された関数呼び出しであっても, 実際には逐次的に実行されている部分も多い. つまり,

```
10:     for (i = 0; i < n_moves; i++) {
11:       child_results[i] = spawn move_and_eval(*g, i, moves[i], -beta, -alpha);
12:       ...
```

という並列ループにおいても, i $= (x+1)$ の繰り返しが行われる前に, 実際には i $\leq x$ の繰り返しが終了しているかもしれない. このような場合に, i $\leq x$ の結果を用いて i $\geq (x+1)$ に対する探索窓を狭めたり, i $\leq x$ の結果で β カットを起こして探索を終了させたい.

Cilk では, この適応的な待機を行うための簡便な方法が存在する. それは, SYNCHED という組み込みの変数で, その関数が spawn したタスクが**実は既にすべて終了している**場合に真を返す. 言い換えれば, sync による子タスクの待機が直ちに成立するかどうかを事前に検査できる. Cilk において, 適応的な待機を行う方法は非常に簡単で, 11 行目からの

```
12:       if (i < w) { /* YBWC : 最初の w 個までは逐次的に */
13:         sync;
14:         ...
```

を,

```
12':      if (i < w || SYNCHED) {
13:         sync;
14:         ...
```

と書き換えるだけで良い!

ところで, 実際に SYNCHED がしばしば真になるのかは, スケジューラがどのような順番でタスクを実行するのかに依存する. Cilk のスケジューラの実行方式 [3] は連載第一回目で述べたとおりであるが, ここでのポイントは以下である. Cilk においては, spawn 時はあたかも逐次的な関数呼び出しが実行されたかのごとく, **spawn されたタスクがまず実行を開始**する. この方式を, **work-first** 実行と呼ぶ. spawn されたタスクが実行している間, 暇なワーカが現れたら, それは spawn されたタスクではなくその親 (タスクの継続) を盗んで, 実行する (work stealing). 従って, ある程度全ワーカにタスクが行き渡り, work stealing が起きていない間は各ワーカは逐次と同じ順序で計算を行う. そのような状態では SYNCHED 変数はことごとく 1 を返す.

SYNCHED 変数が 0 になるのは, あるノード g の評価中,

- その子ノード h の評価が spawn される
- h の評価が実行されている間に, g (h の継続) が他のワーカに盗まれる
- その盗んだワーカが SYNCHED をチェックするコードを実行する

というケースだけである. つまり, 子ノードを評価するループが実際に複数のワーカによって並列実行された時以外, 訪問ノード数は増大しないことになる. 特に 1 ワーカでの実行では訪問ノード数は増大しない. これは非常に有用な性質である.

3.5 強制終了

目標は, ある状態 g の子ノードが並列に実行されている際, ある子ノードの結果が判明し次第, その結果で β カットが起きるならば, 実行中の子ノードがあっても, g の評価をそこで終了させることである. Cilk の言葉で言えば, spawn されているタスクのうちの「ひとつ」が終了し次第, spawn されている残りのタスクを強制的に終了させることである.

Cilk で spawn されているタスクの終了を検出するのは sync 構文であった. これは spawn されている「すべての」タスクの終了を待つ構文であるため, この目的—どれか「ひとつ」のタスクの終了を待つ—には使えない. 3.4 節で述べた SYNCHED も, spawn されている「すべての」タスクが終了しているかどうかを示す変数で, やはり使えない.

Cilk でそのようなことを行う手段として, inlet という特別な構文が用意されている. それは, spawn したタスクが終了したときに, その返り値を伴って呼び出されるハンドラである. inlet を定義する構文は C の関数定義の構文とほぼ同じである. ただしプログラムのトップレベルではなく, それを用いる Cilk 関数の中で定義される. そして spawn 時に終了時ハンドラである inlet を指定することができるよう, spawn の構文が拡張される. 以下は, `spawn f(...)` を実行しつつ, それが終了したら, handler を呼び出す.

```
inlet handler(int val, int x, int y) {
   ...
}
handler(spawn f(..), x, y);
```

また, inlet 中では特別な文 abort を実行することが出来, タスク g の inlet 中でこれを呼び出すと, g のすべての子タスクが強制終了する.

以上をまとめて, 強制終了を Cilk でコーディングしたものが以下である. 雰囲気としては, sync 後に行われた処理が inlet の中に移動し, 「非同期に」行われる, という感じになる.

```
/* 局面 g の評価値を返す */
int eval(game_state_t g, int alpha, int beta, int w) {
  int moves[MAXEMPTIES];
  int n_moves = gen_moves(g, moves);
  if (n_moves == 0) {
    return g->disc_diff;
  } else {
    int i;
    inlet handler(int child_result, int j) {
      int e = -child_result;
```

```
      if (e > alpha) {
        alpha = e;
        if (e >= beta) abort;
      }
    }
    for (i = 0; i < n_moves; i++) {
      handler(spawn move_and_eval(*g, i, moves[i], -beta, -alpha, w), i);
      if (i < w || SYNCHED) sync;
      if (alpha >= beta) break;
    }
    return alpha;
  }
}
```

4 OpenMP による並列化

4.1 基本

OpenMP も 3.0 以降の仕様でタスク並列をサポートする．Cilk とほぼ同様の概念で, 並列化を記述することができる．タスクを生成する構文—Cilk の spawn に相当する—は,

```
#pragma omp task shared(...) if(...)
  文
```

である．文には任意の C の文を指定でき, これで「文」を実行するタスクが生成される．Cilk のように, 関数呼び出ししかタスクとして実行できないというような構文的な制限はない．shared 節を使って, 親タスクと子タスク間で共有する局所変数を指定する．さもなければ局所変数はコピーされ, 代入文による更新も伝搬しない．if 節で, タスクを生成するための条件を指定でき, 成り立たない場合は「文」は通常通り逐次実行される．

タスクの終了待ち—Cilk の sync に相当する—は,

```
#pragma omp taskwait
```

というプラグマである．

以上で Cilk の spawn/sync を用いたのと同様の並列化が記述できる．ただし OpenMP では, task プラグマに先立って, parallel プラグマを使って並列セクションに入ってワーカを生成する必要がある．さらに, parallel プラグマのセマンティクスは, 全ワーカが同じコードを重複して実行するというもので, ここでの目的と合致しない．そのため, parallel プラグマの直後に, master プラグマを使って, 一ワーカでの実行を始める．つまり

```
#pragma omp parallel
#pragma omp master
  文
```

としておいて, 上記の「文」中で task/taskwait を用いる．

4.2 YBWC

基本的には以上のテンプレートに従って, Cilk の spawn を task, sync を taskwait に置き換えれば良い.

```
 1: int eval(game_state_t g, int alpha, int beta, int w) {
 2:   int moves[MAXEMPTIES];
 3:   int child_results[MAXEMPTIES];
 4:   int n_moves = gen_moves(g, moves);
 5:   if (n_moves == 0) {
 6:     return g->disc_diff;
 7:   } else {
 8:     int i;
 9:     for (i = 0; i < n_moves; i++) child_results[i] = INF;
10:     for (i = 0; i < n_moves; i++) {
11: #pragma omp task shared(child_results)
12:       child_results[i] = move_and_eval(*g, i, moves[i], -beta, -alpha, w);
13:       if (i < w) { /* YBWC : 最初の w 個までは逐次的に */
14: #pragma omp taskwait
15:         int e = -child_results[i];
16:         if (e > alpha) {
17:           alpha = e;
18:           if (e >= beta) break;
19:         }
20:       }
21:     }
22: #pragma omp taskwait
23:     for (i = 0; i < n_moves; i++) {
24:       int e = -child_results[i];
25:       if (e > alpha) {
26:         alpha = e;
27:         if (e >= beta) break;
28:       }
29:     }
30:     return alpha;
31:   }
32: }
33:
34: int main() {
35: #pragma omp parallel
36: #pragma omp master
37:   {
38:      ...
39:      eval(...);
40:      ...
```

```
41:   }
42: }
```

4.3 適応的な待機

OpenMP には, Cilk の SYNCHED 変数に対応するものはない. 従って上記のタスクを生成しているループ

```
10:     for (i = 0; i < n_moves; i++) {
11: #pragma omp task shared(child_results)
12:       child_results[i] = move_and_eval(*g, i, moves[i], -beta, -alpha, w);
13:       if (i < w) { /* YBWC : 最初の w 個までは逐次的に */
14: #pragma omp taskwait
15:         ...
```

において, `if (i < w)` を書き換える簡便な方法が存在しない.

今の目標は, あるタスクがすでに終了しているか否かを判定するということだが, そのためのポータブルな方法として, 適当な共有変数への書き込みでタスクの終了を通知するという方法も考えられる. 今の場合それは簡単で, 関数呼び出しの結果の書き込みでそれを通知するのが自然である. 例えば,

```
10:      for (i = 0; i < n_moves; i++) {
10':       child_results[i] = INVALID;
11:  #pragma omp task shared(child_results)
12:        child_results[i] = move_and_eval(*g, i, moves[i], -beta, -alpha, w);
13':       if (i < w || child_results[i] != INVALID) {
14:  #pragma omp taskwait
15:          ...
```

のようにすることが考えられる. `INVALID` は, `move_and_eval` の返り値 (盤面の評価値) に成り得ない適当な値である.

しかし実際にこれが功を奏するかどうかはスケジューラの実装 (タスクの実行順序) に依存し, 我々が評価対象としている GCC の OpenMP 実装 (GOMP[4]) では, 功を奏さない. その理由は, GOMP のスケジューラが Cilk のような work-first 実行ではない点にある. GOMP スケジューラでは,

- task プラグマが実行された際, 生成された子タスクを, 個々のワーカが管理しているタスクスタックにプッシュする.
- その上で, もともと実行していた**親が実行を継続する (parent-first 実行)**.
- taskwait に到達した時点で, すでに子タスクが終了していなければ, タスクスタックの上から子タスクを取り出して順次実行する. この時点で, 他のワーカによってそれらの子タスクの一部がすでに盗まれて, 実行されている可能性もある.

性質として, 一旦タスクが全ワーカに行き渡ったあとは, work stealing があまり起きなくなるという点は Cilk と共通しているが, その際の実行順序は異なる. 特に, 子タスクは, 他のワーカに盗まれない限り, 親タスクが taskwait に到達するまで実行を**開始しない**. これは, 複数の子タスク h_1, h_2 を (taskwait を挟まずに) 生成した際, h_2 の実行開始時点で, h_1 が実行を終了している可能性はほとん

どないことを意味している. あるとすれば, h_1 生成直後にそれが他のワーカに盗まれて, h_2 生成以前に終了した場合だが, これはほとんど期待できない.

改めて適応的な待機の目的を思い返すと, それは兄弟ノード間の情報の伝搬であった. つまり, あるノード g の子ノード h の評価結果に応じて, g の他の子ノード (つまり h の兄弟) の評価を不要にしたり, 探索窓を狭めたりすることであった.

そのための, むしろ素直な方法は, 親ノードの局所変数である alpha をノード間で共有してしまうことである. タスクは, 子ノードを評価するだけでなく, 子ノードの結果により alpha を更新する処理までを行う.

以下はこの方針に基づいて書きなおしたものである. 子ノードを生成するループ部分だけを抜粋している. alpha が共有変数になっている点がポイントである.

```
 1: for (i = 0; i < n_moves; i++) {
 2: #pragma omp task shared(alpha) if(i>=w)
 3:   if (alpha < beta) {
 4:     int e = -move_and_eval(g, i, moves[i], -beta, -alpha, w);
 5:     omp_set_lock(&lock);
 6:     if (e > alpha) alpha = e;
 7:     omp_unset_lock(&lock);
 8:   }
 9:   if (alpha >= beta) break;
10:  }
11: #pragma omp taskwait
12: return alpha;
```

3 行目から始まる if 文全体がタスクとして実行されることに注意. そして, 3, 4 行目で alpha を読んでいるが, これは他の兄弟タスクによって更新されているかも知れない値である. そして, 5-7 行目で alpha を更新しており, それにより, **これから始まる**兄弟タスクは, 更新された探索窓を使って探索ができる.

このコードが, 他のワーカからの work stealing を受けずに, parent-first 実行を行うスケジューラで実行されたときの挙動を想像してみる.

1. まず 3 行目から 8 行目までの if 文は, `i < w` の時を除き, 実行されずにスタックに置かれたまま親の実行が継続する.
2. 11 行目の taskwait に達したところでそれらの実行が順次, 行われていく.
3. そこで最初に実行された子タスクが終了する時には, alpha の値が更新されているかもしれない.
4. その場合, 次のタスクが 3 行目や 4 行目で alpha を読む際に, 更新した値が読まれる. これは望みどおりの結果である.

なお, 上記のスケジューリング方式のもうひとつの注意点として, 同一の親から作られた複数の子タスクは, 他のワーカに盗まれない限り, **作られたのと逆順に**実行される, という点があげられる. つまり,

```
for (i = 0; i < n; i++) {
#pragma omp task
    E(h_i);
```

```
}
#pragma omp taskwait
```

というようなシーケンスでの実際の実行順序は, 直感に反する, $h_{n-1}, h_{n-2}, \ldots, h_0$ である. そしてなお悪いことにこれはスケジューラの実装依存である.

これは, 有望な手を先に探索するという, 指し手の順序付けに効率が大きく依存しているプログラムにとっては, かなり重大な問題である. 今回の実装では, GOMP スケジューラの挙動を仮定して, 並列に実行される子タスクの生成を逆順に行っている. for 文の $\mathtt{i} = x$ の繰り返しで, 配列 moves から適切な手を選択するよう, 上記のプログラムを書き換える必要があるがコード省略する.

4.4 強制終了

OpenMP の task プラグマには, 実行中のタスクを強制終了する機能—Cilk の abort に相当する—は存在しない. 言語処理系の特別な機能に頼らずに実装するには, あるタスクから兄弟タスクへ, 終了を命令し, それを検出する機構が必要である.

そのために,

1. 評価中の各ノードに, 「強制終了」フラグを設ける
2. ノード g の評価中, そのどれかの子ノード h の評価結果がわかったことによって, g の評価を終了 (β カット) できる場合, h は, g の強制終了フラグを 1 にセットする
3. 各タスクは実行開始に先立ち, 自分の先祖のどれかですでに強制終了フラグが 1 になっているものがあるかを検査し, あったら直ちに終了する.

先祖の強制終了フラグを見られるようにするため, 以下のようなリンクリスト風の構造体

```
typedef struct abort_signal {
  int aborting;
  struct abort_signal * parent;
} abort_signal, * abort_signal_t;
```

を用意する. 子タスクを生成する際, 各タスクに新しい abort_signal 構造体を用意し, 自分が受け取った構造体へのリンクを作る. コードの抜粋としては以下のようになる.

```
 1: /* 局面 g の評価値を返す */
 2: int eval(game_state_t g, int alpha, int beta, abort_signal_t as, int w) {
 3:   if (check_abort(as)) return -INF;
 4:   int moves[MAXEMPTIES];
 5:   int n_moves = gen_moves(g, moves);
 6:   if (n_moves == 0) {
 7:     return g->disc_diff;
 8:   } else {
 9:     int i;
10:     abort_signal child_as[MAXEMPTIES];
11:     for (i = 0; i < n_moves; i++) {
12:       child_as[i].parent = as;
13:       child_as[i].aborting = 0;
```

```
14: #pragma omp task shared(alpha, as, child_as)
15:       if (alpha < beta) {
16:         int e = -move_and_eval(g, i, moves[i], -beta, -alpha_, &child_as[i], w);
17:         omp_set_lock(&lock);
18:         if (e > alpha) alpha = e;
19:         if (e >= beta) as[i].aborting = 1;
20:         omp_unset_lock(&lock);
21:       }
22:       if (alpha >= beta) break;
23:     }
24: #pragma omp taskwait
25:     return alpha;
26:   }
27: }
```

check_abort はリンクリストをたどるだけの, 以下のような単純な関数である.

```
int check_abort(abort_signal_t as) {
  abort_signal_t p;
  for (p = as; p; p = p->parent)
    if (p->aborting) return 1;
  return 0;
}
```

なお, あるノードで自分の先祖の強制終了フラグを検査するには, そのノードの深さに比例したコストがかかる. そのためゲーム木の末端に至るまですべてのノードで検査を行うと, 相対的なオーバーヘッドが大きくなるし, またそうする必要もない. そこで, ノードの深さがある程度以上大きい場所では検査をしないことにする. ここでは, 3.3 節で述べた逐次化が行われるまでこの検査をし, 逐次化された部分では検査をしないことにしている.

5 Intel TBB による並列化

5.1 基本

Intel TBB のタスク並列機能はいくつか異なった水準のものが提供されており, 低水準なインタフェースでは, task クラスのサブクラスを定義し, それに execute 演算子を定義する. そして各タスクをそのようなクラスのインスタンスとして, spawn() というメソッドで起動する.

もう少し高水準なインタフェースとして, Cilk の spawn/sync や, OpenMP の task/taskwait に相当するものも用意されており, それが task_group というクラスである. task_group クラスのオブジェクトを作り,

- run() メソッドを呼び出してタスクを生成する
- wait() メソッドを呼び出して, 生成されたタスクの終了を待つ

というのが基本である. この際, run() メソッドには無引数の関数, operator() を持つオブジェクト, task_handle というクラスのオブジェクトを渡せるが, 特に C++の時期標準とされている C++0x の

クロージャという機能を使うと, 付加的なクラスを定義することなく, タスクを生成できる. GCC であれば, -std=c++0x というオプションを付けることで利用可能である. これで構文的にも, OpenMP の task とほぼ同じことができる.

以下は再帰的にタスクを生成する最も簡単な使用例である.

```
 1: #include <tbb/task_group.h>
 2: using namespace tbb;
 3:
 4: int fib(int n) {
 5:   if (n < 2) return 1;
 6:   else {
 7:     task_group tg;
 8:     int x, y;
 9:     tg.run([=,&x] { x = fib(n - 1); });
10:     y = fib(n - 2);
11:     tg.wait();
12:     return x + y;
13:   }
14: }
15:
16: int main(int argc, char ** argv) {
17:   int n = atoi(argv[1]);
18:   int x= fib(n);
19:   printf("fib(%d) = %d\n", n, x);
20: }
```

以下が GCC でコンパイルするための具体的なコマンドになる (<tbb_dir>は TBB のトップディレクトリ).

```
g++ -std=c++0x -I<tbb_dir>/include fib.cc -L<tbb_dir>/lib -ltbb
```

ここで,

```
 9:     tg.run([=,&x] { x = fib(n - 1); });
```

に現れる, [=,&x] { x = fib(n - 1); } という構文が C++0x のクロージャ(ラムダ式) である. [=,&x] は,

- &x により, 変数 x はクロージャの内外間で共有する
- =によりそれ以外はクロージャ生成時にコピーする

ことを示している. OpenMP の task プラグマでいえば, shared(x) と書いたことに相当する.

5.2 YBWC

基本的な YBWC による並列化は以下のとおり, ほぼ OpenMP の task/taskwait を, task_group クラスの run/wait の呼び出しに置き換えるだけでよい.

```
 1: int eval(game_state_t g, int alpha, int beta, int w) {
 2:   int moves[MAXEMPTIES];
 3:   int child_results[MAXEMPTIES];
 4:   int n_moves = gen_moves(g, moves);
 5:   if (n_moves == 0) {
 6:     return g->disc_diff;
 7:   } else {
 8:     int ii;
 9:     task_group tg;
10:     for (ii = 0; ii < n_moves; ii++) child_results[ii] = INF;
11:     for (ii = 0; ii < n_moves; ii++) {
12:       int i = (ii < w ? ii : n_moves - ii + w - 1);
13:       tg.run([=,&child_results]
14:          { child_results[i] = move_and_eval(*g, i, moves[i], -beta, -alpha, w); });
15:       if (i < w) { /* YBWC : 最初の w 個までは逐次的に */
16:         tg.wait();
17:         int e = -child_results[i];
18:         if (e > alpha) {
19:           alpha = e;
20:           if (e >= beta) break;
21:         }
22:       }
23:     }
24:     /* 改めてすべてを待つ */
25:     tg.wait();
26:     for (ii = 0; ii < n_moves; ii++) {
27:       int e = -child_results[ii];
28:       if (e > alpha) {
29:         alpha = e;
30:         if (e >= beta) break;
31:       }
32:     }
33:     return alpha;
34:   }
35: }
```

なお TBB のスケジューラでもタスクの実行順序は, `i >= w` 以降, 生成と逆順になる. それを加味して, タスクの生成順序を変えているのが,

```
11:       int i = (ii < w ? ii : n_moves - ii + w - 1);
```

の部分である.

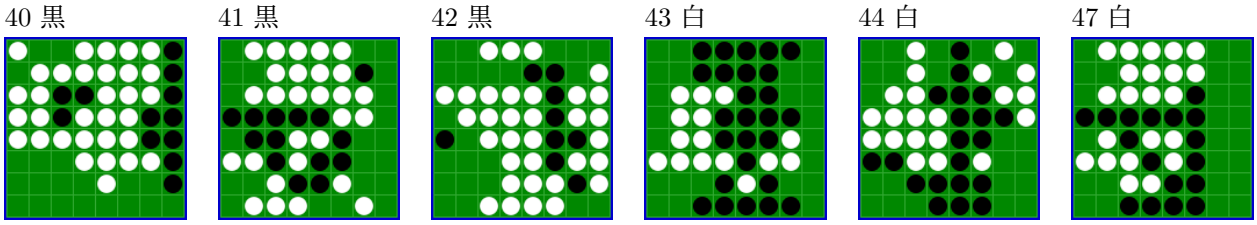

図 1: 評価に用いた盤面と手番 (http://radagast.se/othello/ffotest.html より). いずれも直前の手はパスではない.

5.3 適応的な待機

TBB における適応的な待機を実装するための考慮事項は, OpenMP の場合とほとんど同じである. Cilk の SYNCHED 変数に相当するものは存在しないし, スケジューラの挙動も GOMP スケジューラと同じ, parent-first 方式である. OpenMP と同様, 局所変数を親子タスク間で共有することができるので, 解決方法も同様のものが採用できる.

OpenMP の時と同様, 子ノードを生成するループ部分だけを抜粋する.

```
task_group tg;
spin_mutex lock_;
for (ii = 0; ii < n_moves; ii++) {
  int i = (ii < w ? ii : n_moves - ii + w - 1);
  tg.run([=,&alpha,&lock_] {
      if (alpha < beta) {
        int e = -move_and_eval(g, i, moves[i], -beta, -alpha, w);
        spin_mutex::scoped_lock lock(lock_);
        if (e > alpha) alpha = e;
      }
    });
  if (i < w) tg.wait();
  if (alpha >= beta) break;
}
tg.wait();
return alpha;
```

なお, 強制終了も OpenMP におけるものと同様に実装できる. 詳細は省略する.

6 実験

6.1 設定

評価対象の盤面としては, 前号の記事でも述べた, Andersson の提供する The FFO endgame test suite (http://radagast.se/othello/ffotest.html) にある局面を用いた. これは, 逐次での実行時間が 500 秒以内のもの 6 つで, 主に各種プログラムを様々なパラメータで実行する際の, 実験にかかる時間の制約から, 選んだものである.

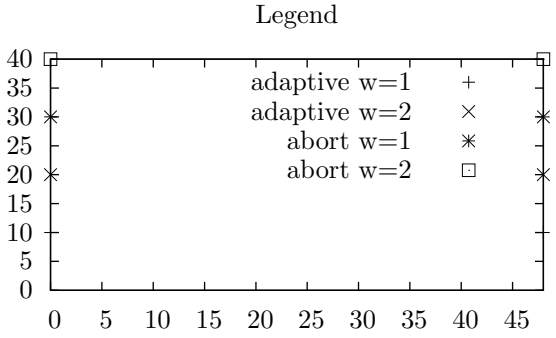

図 2: 図 3-5 の凡例. adaptive は 2.4 節で述べた適応的待機方式, abort は 2.5 節で述べた強制終了方式. w は, 2.2 節で説明した, 何個目の子ノードまでを逐次的に評価するかのパラメータ.

また, 紙面の関係からこのうち, 問題 40, 47 に対する結果 (これはこの 6 つのうちで, 逐次での訪問ノード数が多い 2 つである) を掲載する.

評価環境も前号で用いた 24 コア (48 ハードウェアスレッド) のマシンである.

- CPU: Intel Nehalem-EX (E7540) 2.0GHz (6 core/12 スレッド × 4 ソケット)
- L3 cache: 18MB
- memory: 24GB DDR3

6.2 訪問ノード数の増大

まず, 並列化によってどれだけ訪問ノード数が増大するかを調査する. 下表は, 逐次での訪問ノード数と空きマス数 (≈ ゲーム木の深さ) を示している.

問題番号	空きマス数	訪問ノード数 ($\times 10^6$)
40	20	53
41	22	379
42	22	328
43	23	341
44	23	347
47	25	1266

図 3 は, Cilk, OpenMP, TBB それぞれで実装した, 適応的な待機, 強制終了それぞれの方法について, ワーカ数とともに, 訪問ノード数が逐次の何倍に増加したかを示している.

パラメータとしては,

- 並列にタスクを生成する深さは 12 までに固定
- 各ノードの評価で何番目までの子ノードを逐次的に評価するかのパラメータ (w) を 1 または 2
- アルゴリズムとして適応的な待機および強制終了

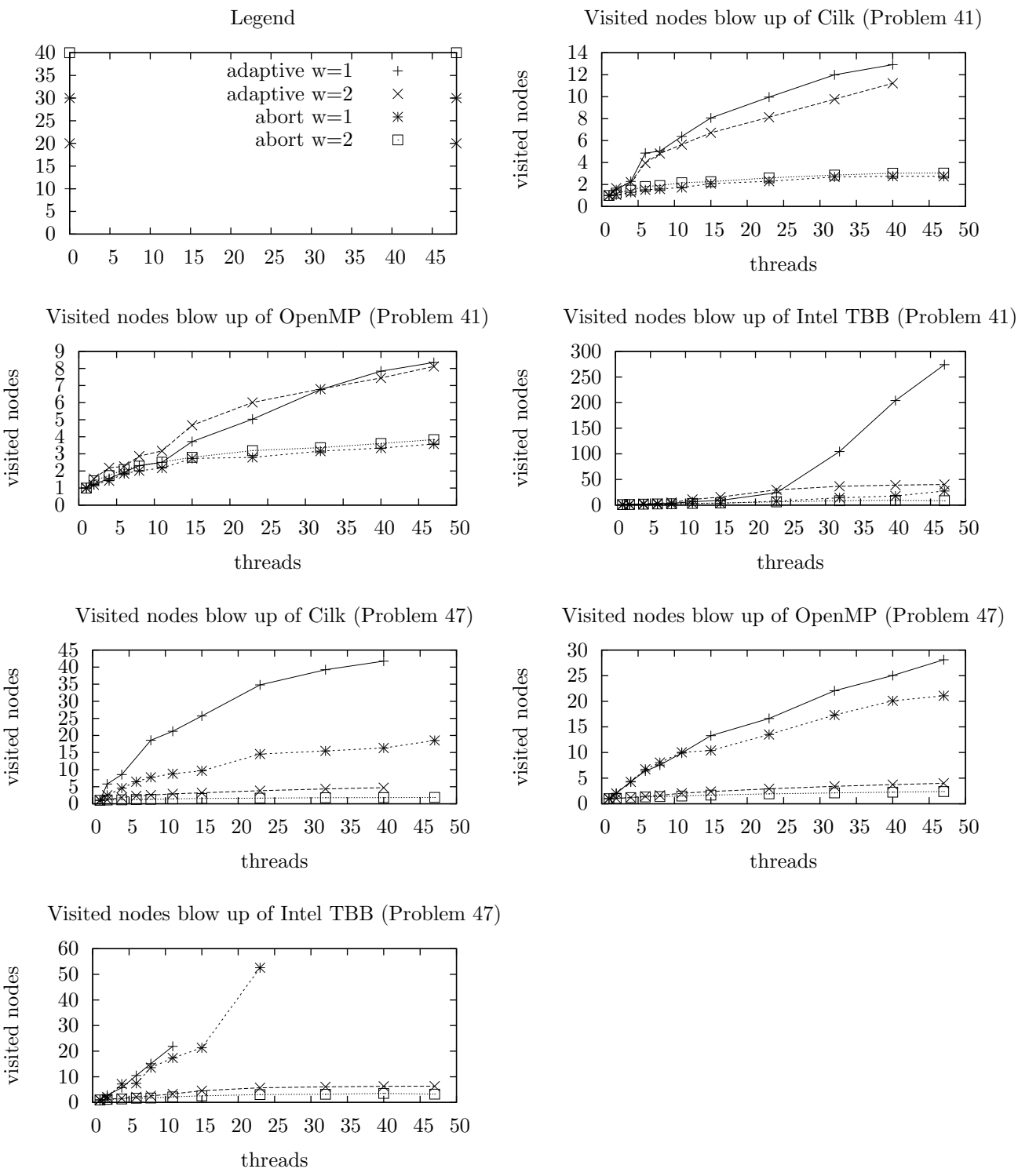

図 3: 訪問ノード数の増大割合 (問題 41,47)

を用い, 各言語につき都合 $2 \times 2 = 4$ 通りのパラメータでの結果を表示している.

どの場合でもワーカ数とともにノード数の増大傾向は著しい. そして, 強制終了を行わない場合に特に顕著で, あらゆる台数効果は容易に打ち消されてしまうであろうことが明らかである. 強制終了を行えば, 40 ワーカで 2 倍-10 倍以内である. なお, 問題によってこの振れ幅は大きく異なるが, これら二つの問題は他の問題に比べるとこの増大傾向は大きかった.

なお, 適応的な待機を行わないアルゴリズム (基本的な YBWC もしくはそこで $w = 2$ としたもの) では, 適応的なアルゴリズム以上にノード数の増大は激しく, しかもそれは (適応的ではないため), 台数によらず一定であった. そのためグラフからは測定結果を除外している.

6.3 逐次性能

下表は, 問題 41 に対して各種言語, アルゴリズム, w (逐次的に実行する子ノード数) に対する, 1 ワーカ利用時の経過時間およびその比を表している. タスク生成は常に 12 段まで行なっており, 訪問ノードの集合はどれも全く同じである.

C (Andersson) が今回元にした逐次プログラムで, 2 段目の C が, 前号で説明したように, 今回の並列化の準備として書きなおした逐次の C プログラムである. C (Andersson) に比べて 50% - 100%程度のオーバーヘッドになっているが, 純粋な逐次の C プログラムもすでに 50%程度のオーバーヘッドであり, 総じて今回用いた処理系が逐次性能に与える影響は小さいと言える.

言語処理系	アルゴリズム	w	時間	時間比
C (Andersson)	-	-	89.8	1.0
C	-	-	136.6	1.52
Cilk	abort	1	138.1	1.54
Cilk	abort	2	137.0	1.53
Cilk	adaptive	1	211.3	2.35
Cilk	adaptive	2	140.8	1.57
OpenMP	abort	1	166.5	1.85
OpenMP	abort	2	146.3	1.63
OpenMP	adaptive	1	133.5	1.49
OpenMP	adaptive	2	138.8	1.46
Intel TBB	abort	1	163.5	1.82
Intel TBB	abort	2	142.5	1.59
Intel TBB	adaptive	1	130.5	1.45
Intel TBB	adaptive	2	133.3	1.48

6.4 台数効果 (スループット)

6.2 節での結果から既に明らかなように, この問題に対して理想に近い台数効果が得られることはほとんど期待できない. ここではノード数増大の効果を一旦打ち消すため, スループット—単位時間あたりの訪問ノード数—の台数効果を見る.

図 4 がその結果である. どの処理系もスループットの向上は 40 コアで 25-30 倍程度得られており, 不規則で動的な問題としては, そこそこ良好であると言える. しかし, スループットの向上もアルゴリズムおよびパラメータ w によって異なり, 訪問ノード数増大とのトレードオフが存在することが読み取れる.

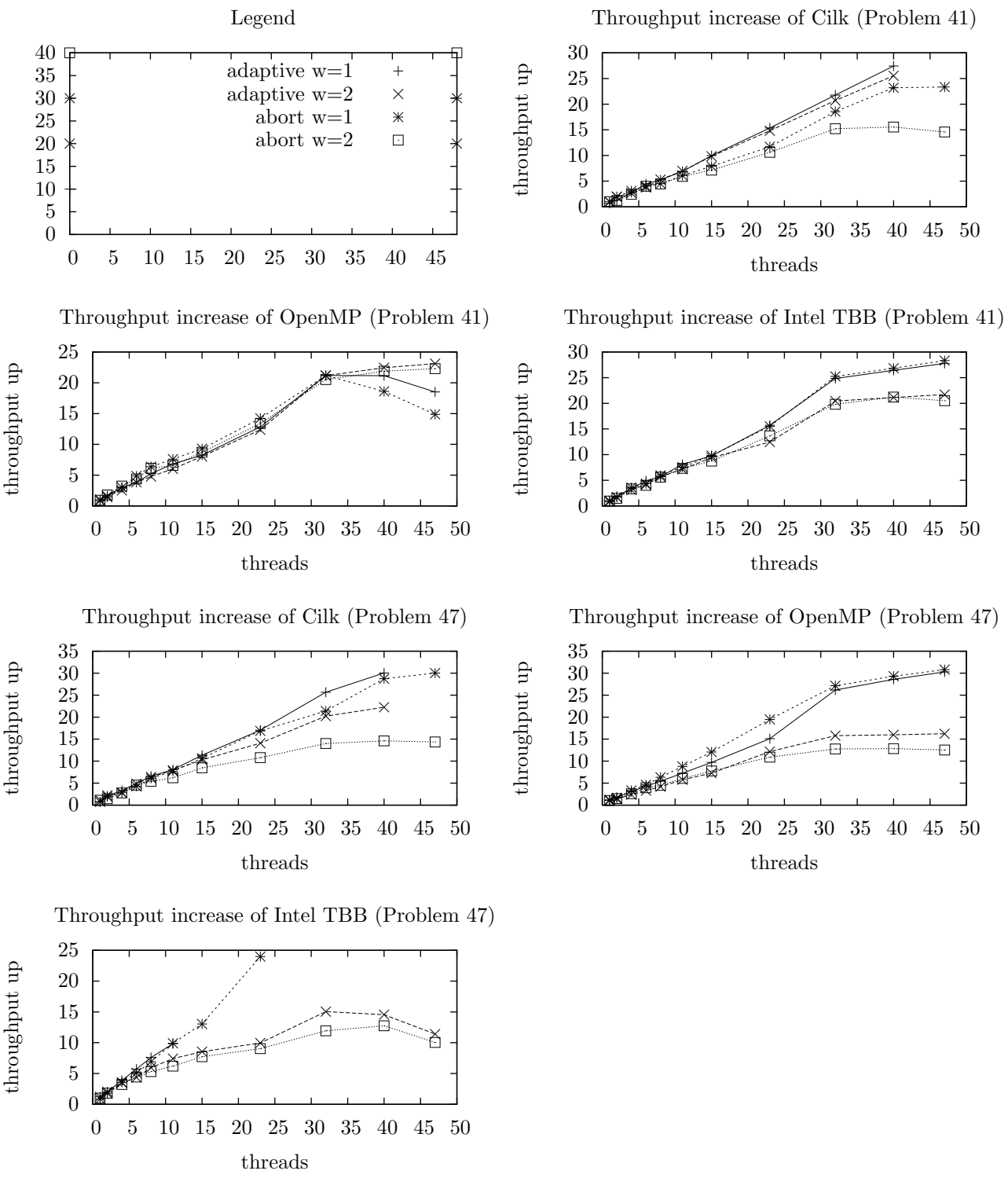

図 4: スループットの台数効果 (問題 41, 47)

6.5 台数効果 (実行時間)

最後に, Andersson の逐次プログラムと比べた, 本当の台数効果が図 5 である. つまりこれは, 図 3 にある計算量の増大, 6.3 節で述べたオーバーヘッド, 図 4 の台数効果を総合したものである. 総合的に, 探索ノード数を抑えることの重要性が現れており, 多くの場合勝者は強制終了をし, かつ $w = 2$ の場合である. それでも得られた台数効果は, 40 ワーカを用いて 5-6 倍である.

7 まとめ

共有メモリ上で動的負荷分散を行うタスク並列処理系を用いて, $\alpha\beta$ 探索の並列化の際の肝となる方式をどう記述するかを述べた. 基本手法として

- YBWC: 最初の子ノードの終了を待ってから残りを並列化
- 指し手の評価順序の保存と, スケジューラとの相互作用
- 適応的な待機: 未開始タスクの枝刈りやそれらへの探索窓の伝搬
- 強制終了: 開始済みタスクの枝刈り

があり, それらが各処理系でどう表現されるかを述べた. ゲーム木探索においては, タスク並列処理系と言っても, タスクの生成と終了待ちが拘束に出来れば良いというものではなく, 実行時の評価順序に基づいた適応的な処理や無駄な探索の除去などを, 記述できることが重要である. 特に, work-first 実行と parent-first 実行は, 大差がないと思われがちであるが, 逐次での評価順序を元にして並列化する場合, 極力その順序を守るということには, 実質的な意味がある. 以下に主な考慮事項と各言語における対応する構文や特徴を述べる

	Cilk	OpenMP	Intel TBB
YBWC	spawn, sync	task, taskwait	task_group, run, wait
指し手の評価順序	work-first 実行	parent-first 実行 (GCC)	parent-first 実行
適応的な待機	SYNHCED 変数, work-first 実行	shared 節	タスク間の変数共有 [&x]
強制終了	inlet, abort	手動	手動

よく知られたことだが, 逐次探索の枝刈り効率を落とさないように並列化を行うことの難しさがここでも確認された.

参考文献

[1] Don Dailey and Charles E. Leiserson. Using cilk to write multiprocessor chess programs. In *Advances in Computer Games 9*, 2001.

[2] Rainer Feldmann. *Game Tree Search on Massively Parallel Systems*. PhD thesis, University of Paderborn, 1993.

[3] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language*, pages 212–223, 1998.

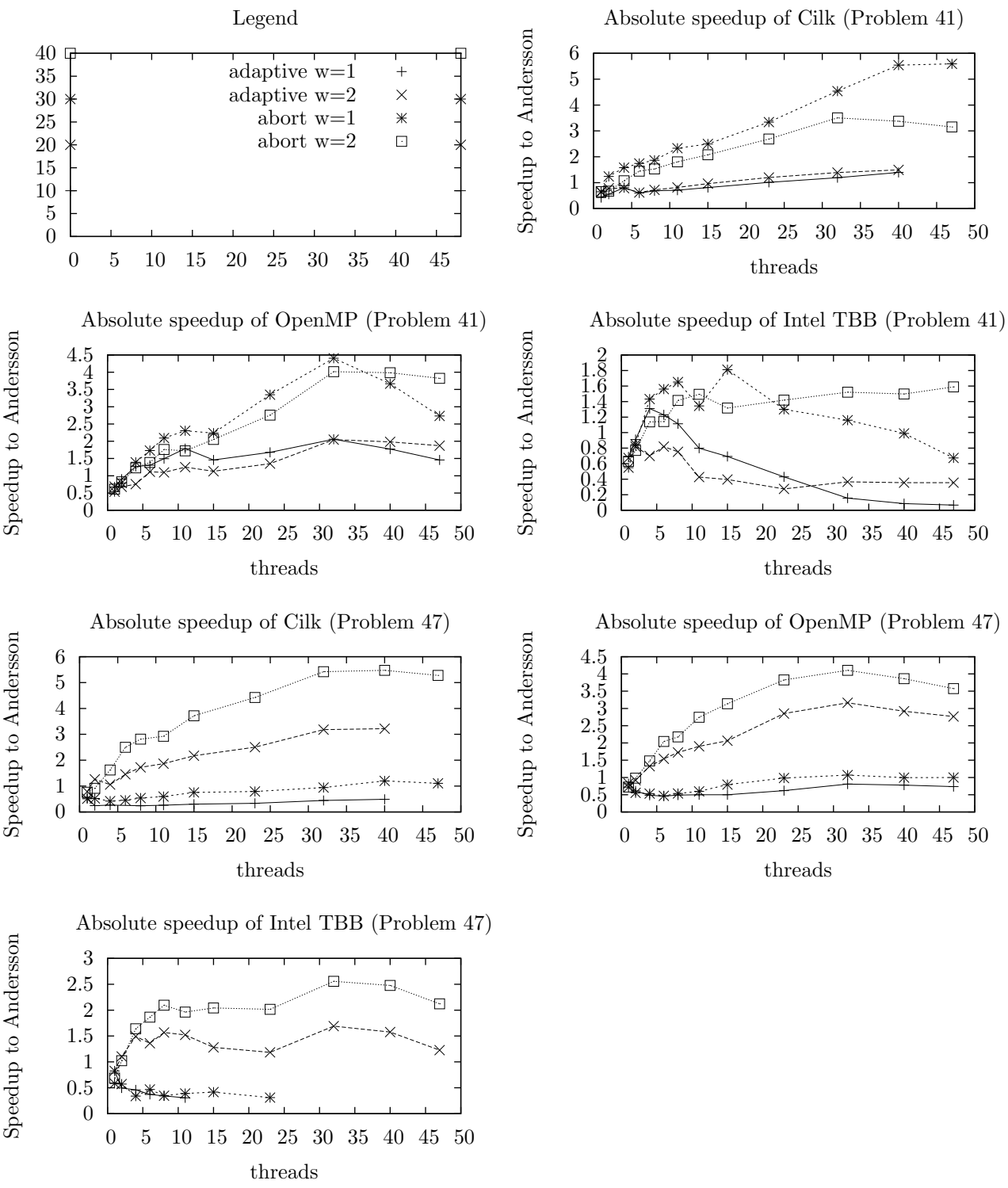

図 5: Andersson のソルバに対する真の速度向上 (問題 41, 47)

[4] GOMP—an OpenMP implementation for GCC. http://gcc.gnu.org/projects/gomp/.

[5] Christopher F. Joerg and Bradley C. Kuszmaul. Massively parallel chess. In *the Third DIMACS Parallel Implementation Challenge*, 1994.

[6] Bradley C. Kuszmaul. The startech massively parallel chess program. *Journal of the International Computer Chess Association*, 18(1), 1995.