プロジェクト課題
「オセロAIの並列化」

2023/7/21

情報工学系B2 山口悠地

概要

1. 背景
2. 実装
3. まとめと課題

1. 背景

2. 実装

3. まとめと課題

✅ ゲームAIでの(定数倍)高速化は非常に重要

1. 探索時間が数倍になればユーザー体験は相当悪化する
2. 探索時間に制限をかけて手を探索する場合、高速化できればより多くの手を探索できてより良い手が打てそう

⇨ 並列化を試してみよう(本レポート)

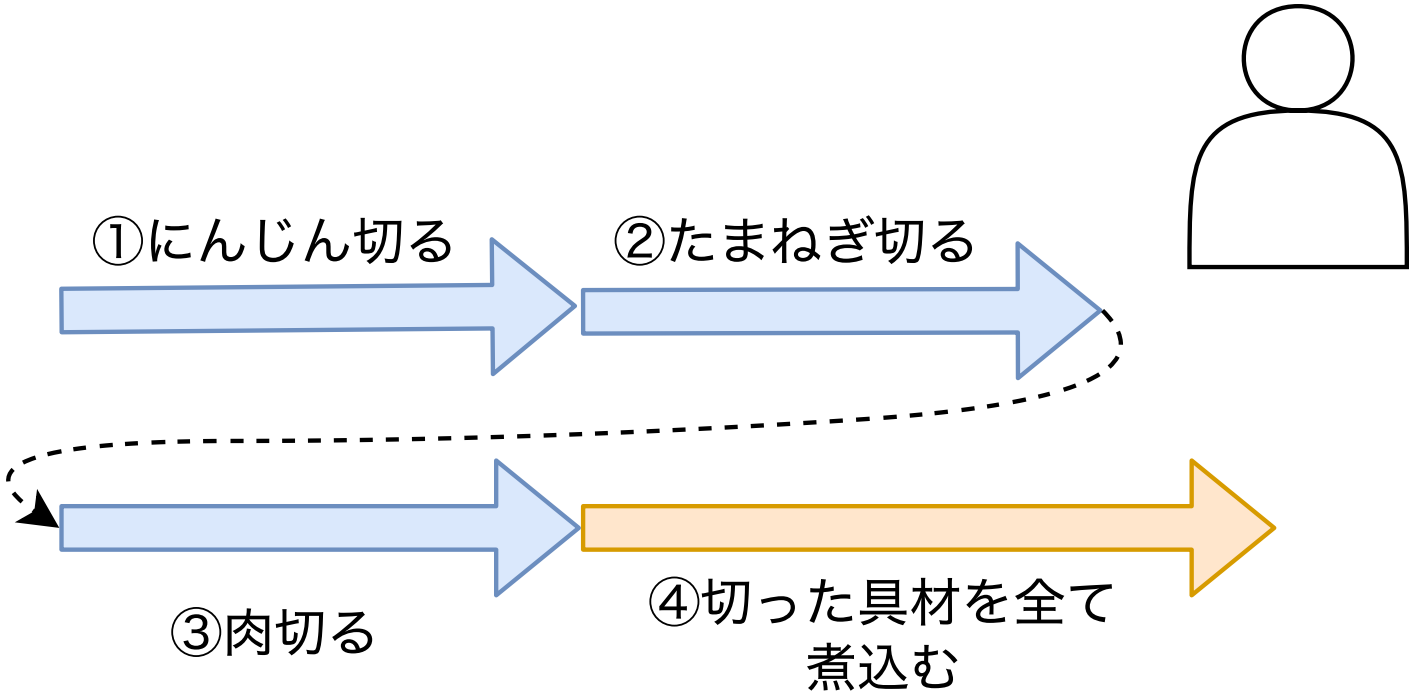
シチューの作り方

1. にんじんを切る
2. たまねぎを切る
3. 肉を切る
4. 具材を全て煮込む

背景 / 並列化とは

いちばん簡単なやりかた

⇨ 最初の処理をする。前の処理が終わったら次の処理を行う。

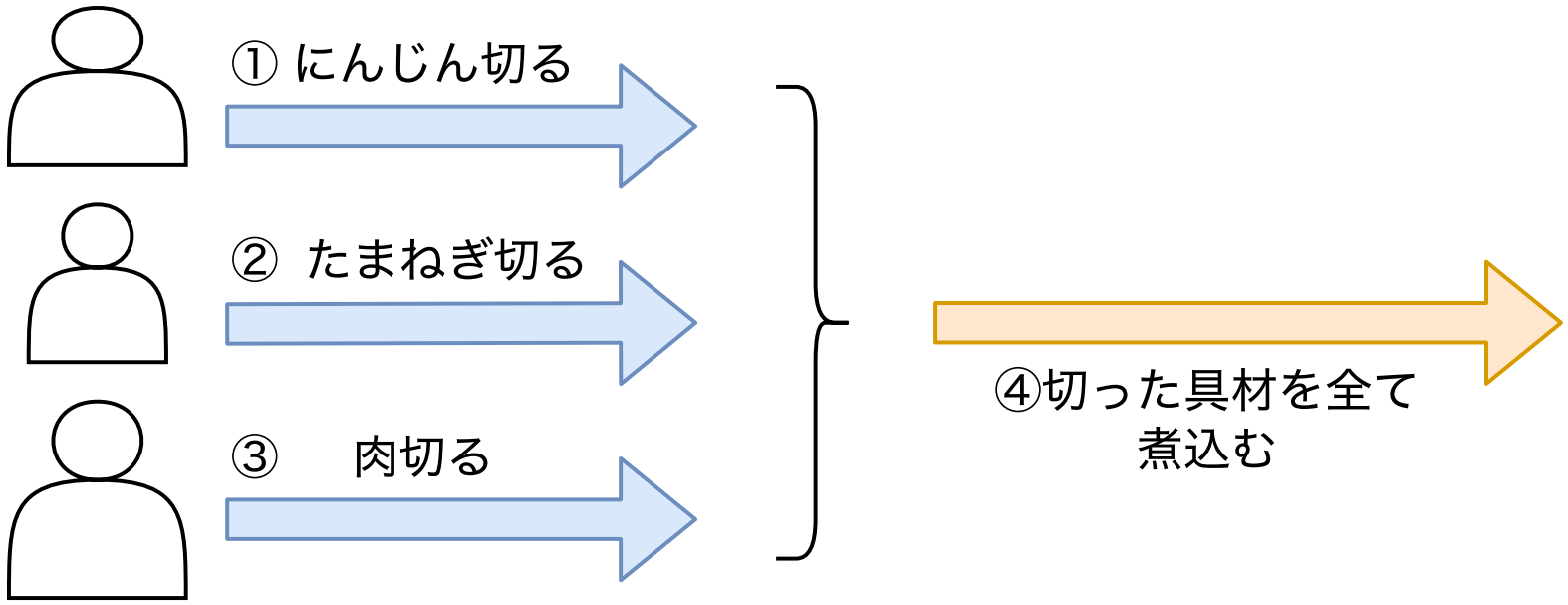


並列シチューの作り方

1. にんじんを切る
1. たまねぎを切る
1. 肉を切る
4. 具材を全て煮込む

背景 / 並列化とは

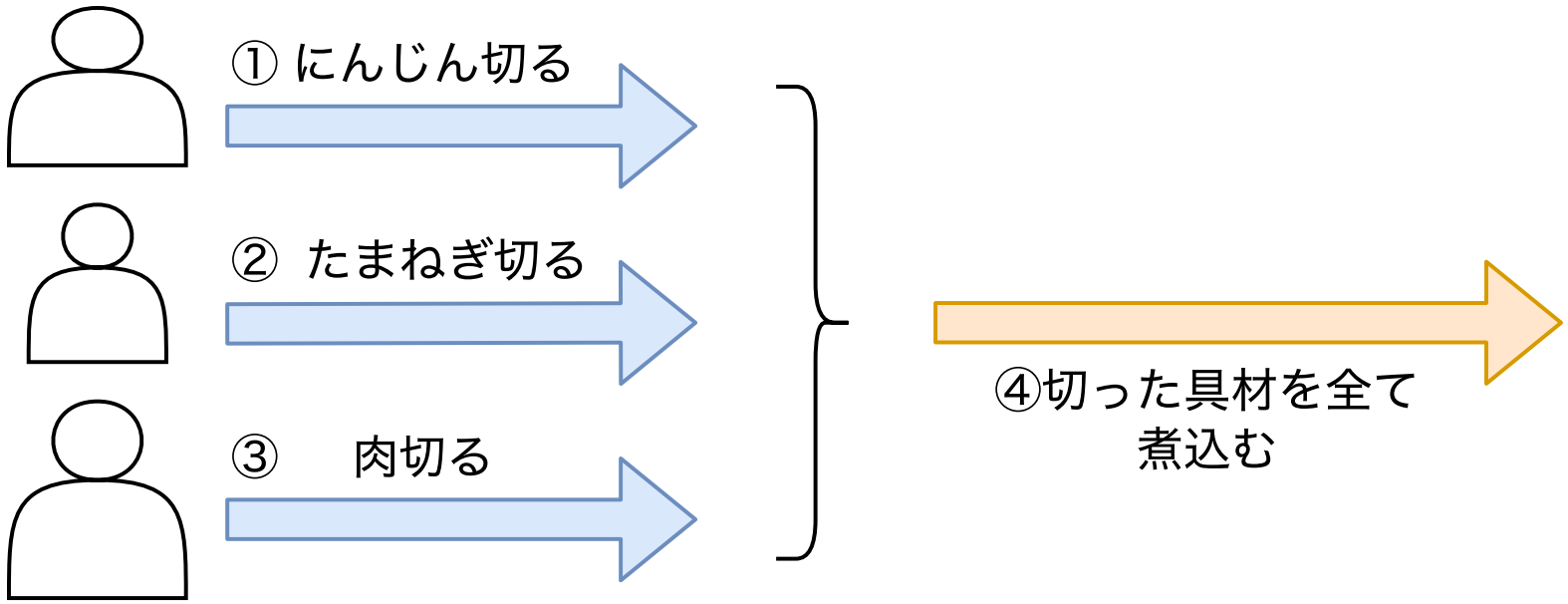
にんじん・たまねぎ・肉　を切る作業は独立(相互に依存していない)
⇨　料理人を３人に増やして同時にやってもOK！



背景 / 並列化とは

現代のCPUはたいてい複数のコア(≈ **料理人**)を持つ

⇨ 複数の処理を同時に行うことができる
⇨ このように同時に処理を行うことを**並列処理**という



並列処理によって高速化が期待できる！

にんじん切る時間(10min) + たまねぎ切る時間(10min) + 肉切る時間(10min) + 煮込む時間(30min) = 60min

⬇　並列化

具材を切る時間(10min) + 煮込む時間(30min) = 40min

✅ ゲームAIでの(定数倍)高速化は非常に重要

1. 探索時間が数倍になればユーザー体験は相当悪化する
2. 探索時間に制限をかけて手を探索する場合、高速化できればより多くの手を探索できてより良い手が打てそう

⇨ 並列化を試してみよう(本レポート)

定数倍高速化とかきましたが、正確にはたとえばPRAMという計算モデルで議論すると並列化したときのオーダーを評価できて、 たとえば配列の総和は並列化アリだと$O(\log n)$で行けるらしいです

1. 背景

2. 実装

3. まとめと課題

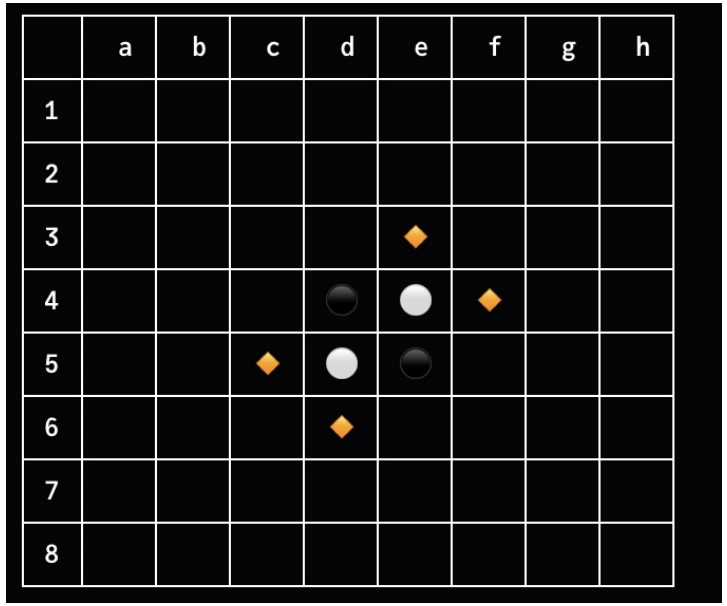
実装

- 実装にはJulia言語をもちいた
- オセロの実装は、ビットボードと呼ばれる実装形式を用いた
- MinMax法, $\alpha\beta$法とそれぞれの並列化を実装した

実装:
https://github.com/abap34/ParallelOthello.jl

ビットボード

オセロは8x8の盤面を持つ
⇨ 各盤面に1つのビットを割り当てて、盤面を64bit整数型で表現



⇨ (68853694464, 34628173824)

ビットボード

黒目線の盤面を表すP, 白目線の盤面を表すOとして

- 黒石がi番目のマスにある
  $\rightarrow P$のiビット目を1に
- 白石がi番目のマスにある
  $\rightarrow O$のiビット目を1に

として(P, O)で盤面を表現する

表1　インデックスの振り方

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

実装 / オセロ自体の実装

ナイーブな実装: 二次元配列による盤面の表現

```
julia> board
8×8 Matrix{Int8}:
 0  0  0   0   0  0  0  0
 0  0  0   0   0  0  0  0
 0  0  0   0   0  0  0  0
 0  0  0   1  -1  0  0  0
 0  0  0  -1   1  0  0  0
 0  0  0   0   0  0  0  0
 0  0  0   0   0  0  0  0
 0  0  0   0   0  0  0  0
```

⇨ これを使って空いているマスを調べてみる

実装 / オセロ自体の実装

```
julia> function empty_mask(board::AbstractArray)
           mask = similar(board)
           for i in 1:8
               for j in 1:8
                   if board[i, j] != 0
                       result[i, j] = 1
                   else
                       result[i, j] = 0
                   end
               end
           end
           return result
       end
empty_mask (generic function with 2 methods)

julia> @benchmark empty_mask(board)
BenchmarkTools.Trial: 10000 samples with 62 evaluations.
 Range (min … max):  860.226 ns …  8.267 μs  | GC (min … max): 0.00% … 86.54%
 Time  (median):     896.516 ns              | GC (median):    0.00%
 Time  (mean ± σ):   907.527 ns ± 84.304 ns  | GC (mean ± σ):  0.08% ±  0.87%

  860 ns          Histogram: log(frequency) by time        1.13 μs <

 Memory estimate: 128 bytes, allocs estimate: 1.
```

ビットボードは $\neg(P \vee O)$ でOK！！

```
julia> function empty_mask(board1::UInt64, board2::UInt64)::UInt64
           return ~(board1 | board2)
       end
empty_mask (generic function with 1 method)

julia> @benchmark empty_mask(game.playerboard, game.opponetboard)
BenchmarkTools.Trial: 10000 samples with 987 evaluations.
 Range (min … max):  54.163 ns … 523.556 ns  | GC (min … max): 0.00% … 84.08%
 Time  (median):     55.344 ns               | GC (median):    0.00%
 Time  (mean ± σ):   56.274 ns ±  11.122 ns  | GC (mean ± σ):  0.48% ±  2.22%

  54.2 ns        Histogram: log(frequency) by time      71.9 ns <

 Memory estimate: 32 bytes, allocs estimate: 2.
```

ビットボード	二次元配列を使った実装
55ns	900ns

⇨ ✅ **16倍〜高速化！！**

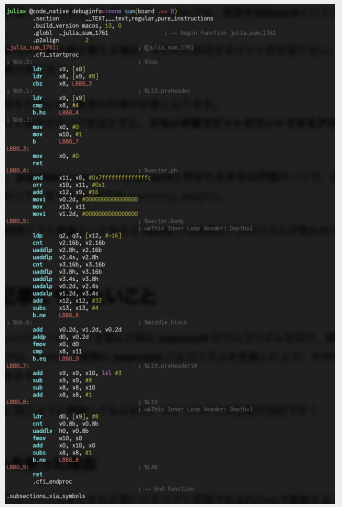
実装 / オセロ自体の実装

吐かれるネイティブコードを見ると...

ビットボード

```
julia> @code_native debuginfo=:none count_ones(game.playerboard)
        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 13, 0
        .globl  _julia_count_ones_1759          ; -- Begin function julia_count_ones_1759
        .p2align        2
_julia_count_ones_1759:                 ; @julia_count_ones_1759
        .cfi_startproc
; %bb.0:                                ; %top
        fmov    d0, x0
        cnt     v0.8b, v0.8b
        uaddlv  h0, v0.8b
        fmov    w0, s0
        ret
        .cfi_endproc
                                        ; -- End function
.subsections_via_symbols
```

二次元配列を使った実装



ビットボードに対する操作は基本的なビット演算で済む
⇨ 少ない命令数で動く
例) 石の数を数える: popcnt命令のみでOK！

実装 / オセロ自体の実装

そのほかの操作もビット演算のみで行える

例1) 合法手の列挙は、6回右シフトを繰り返すことで行える
(src/judge.jlの `legal_eachdirection` 関数)

例2) 石をひっくり返す操作も同様に右シフトのくり返しとXOR演算によって実現可能
(src/judge.jlの `reverse_eachdirection` 関数とsrc/game.jlの `reverse` 関数)

例3) 着手からインデックスの変換は先行ゼロカウント命令で行える
src/solvers/utils.jlの `bit_to_position` 関数

実装 / MinMax法の実装

- src/solvers/minmax.jl
- 再帰で実装
- 葉ノードの評価値は (自分の石の数) - (相手の石の数)

デモ

実装 / Julia言語によるマルチスレッド計算

例: 配列の総和をマルチスレッドで計算する

```
function psum(arr)
    # 計算する領域を分割
    chunks = Iterators.partition(arr, length(arr) ÷ Threads.nthreads
    ())

    # 各領域の計算を割り当てる
    tasks = map(chunks) do chunk
        Threads.@spawn sum(chunk)
    end

    # 各領域の結果を合計
    return sum(fetch.(tasks))
end
```

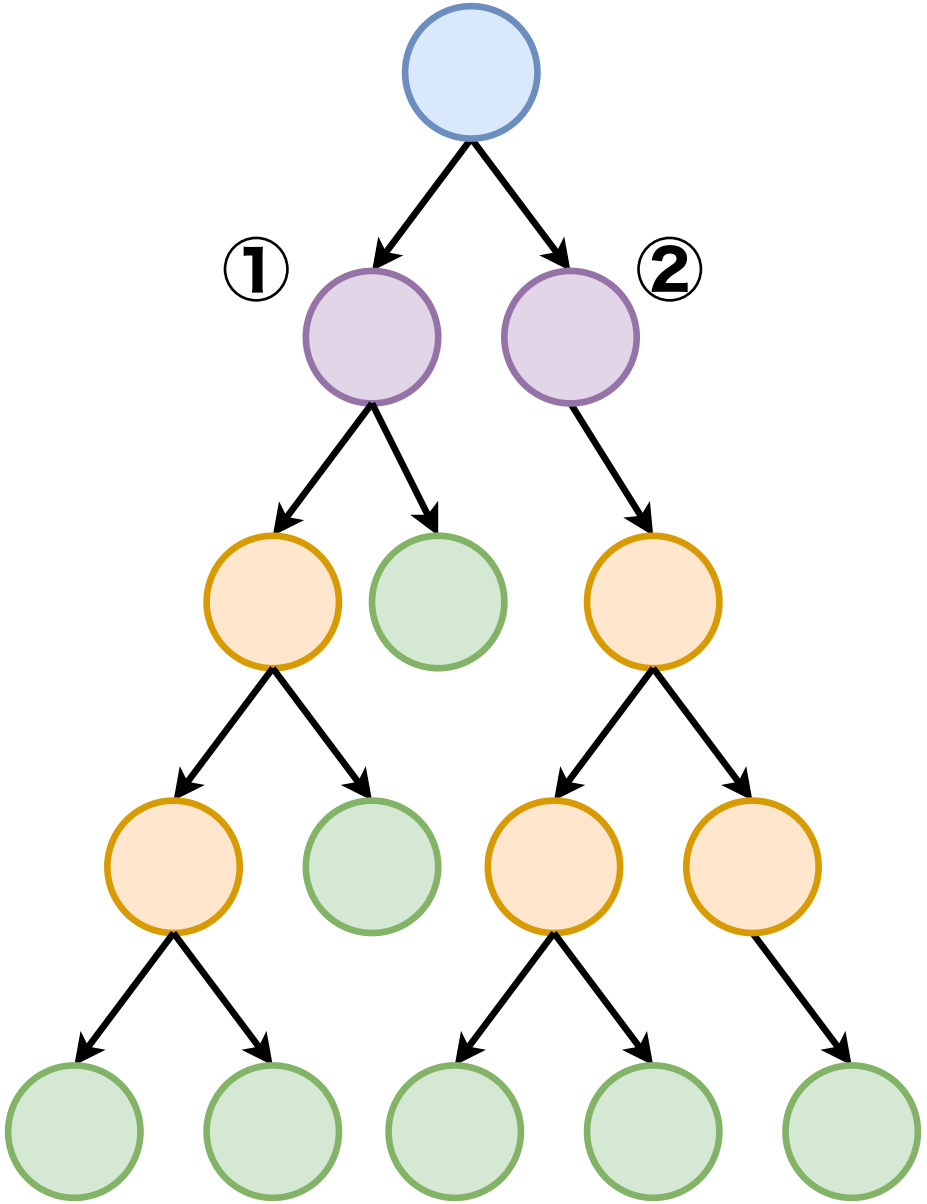
実装 / MinMax法の並列化

MinMax法にたいしてどのように
並列化を行うか？

各ノードを頂点とする
部分木の探索は独立

- 右図 ①と②のノードの評価
  は独立に計算できる

→ 並列化できそう



実装 / MinMax法の並列化

- 各ノード進むたびに並列にスレッドに割り当てていく

**同じアルゴリズム同士の対戦を100回くりかえすのに
かかった時間をしらべると...**

ふつうのMinMax法	並列化したMinMax法
19.88 [s]	12.14 [s]

- そこそこ速くなった

✅ オーバーヘッドに気をつける

あらたにスレッドにTaskを投げるのはノーコストではない

⇨ 並列化しまくる場合 **恩恵 < オーバーヘッド**　になる

↓

すこし改善を加えてみる

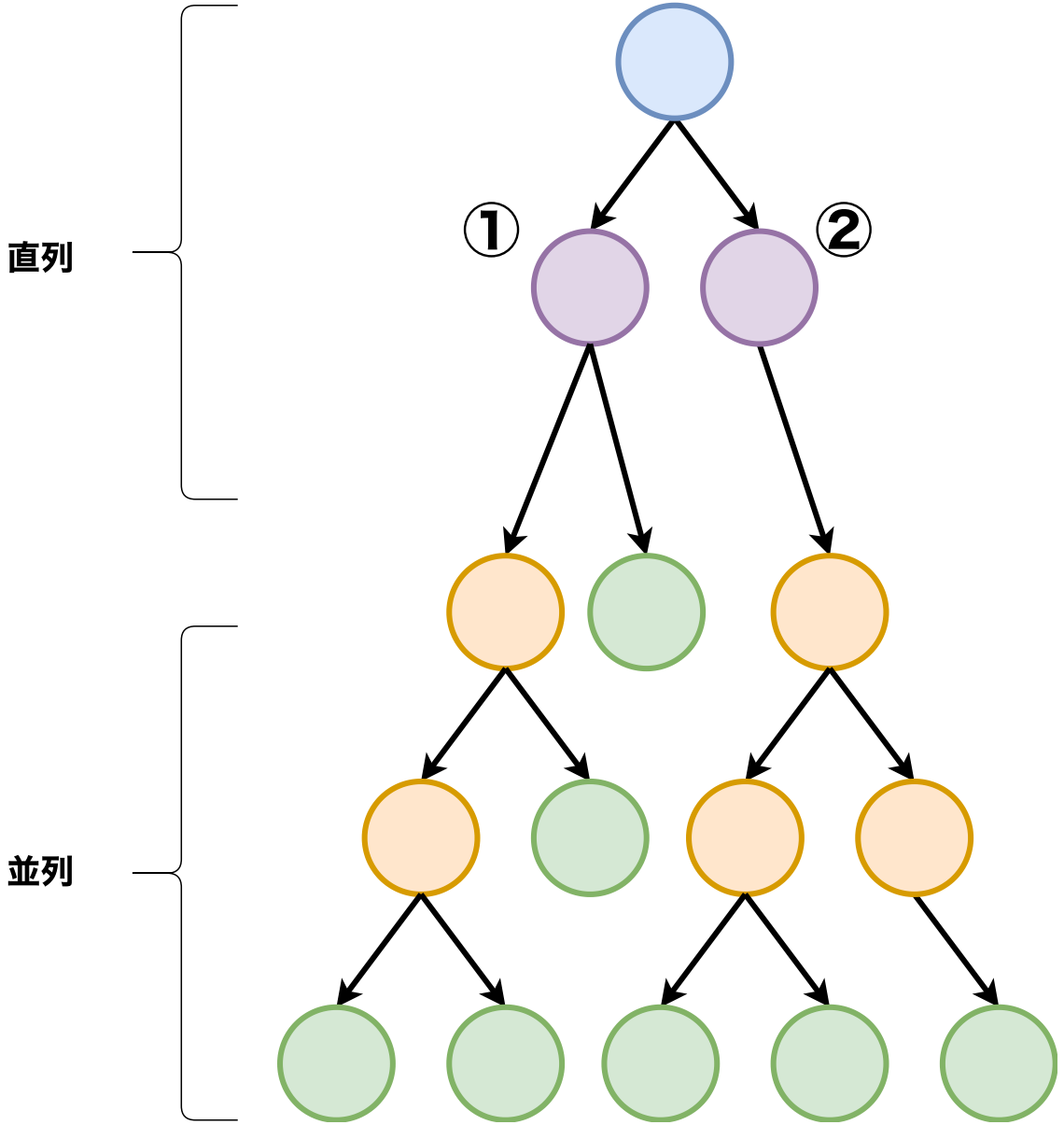
実装 / MinMax法の並列化 改良 ver

スレッド数が増えすぎないように
いい感じのところ以下のみで
並列化を行う

ふつう	並列化	並列化 ②
19.88 [s]	12.14 [s]	7.845[s]

⇨ ✅ **さらに高速化**

(今回は結局ルートノードの各子を根にする部分木ごとにスレッドを割り当てた)

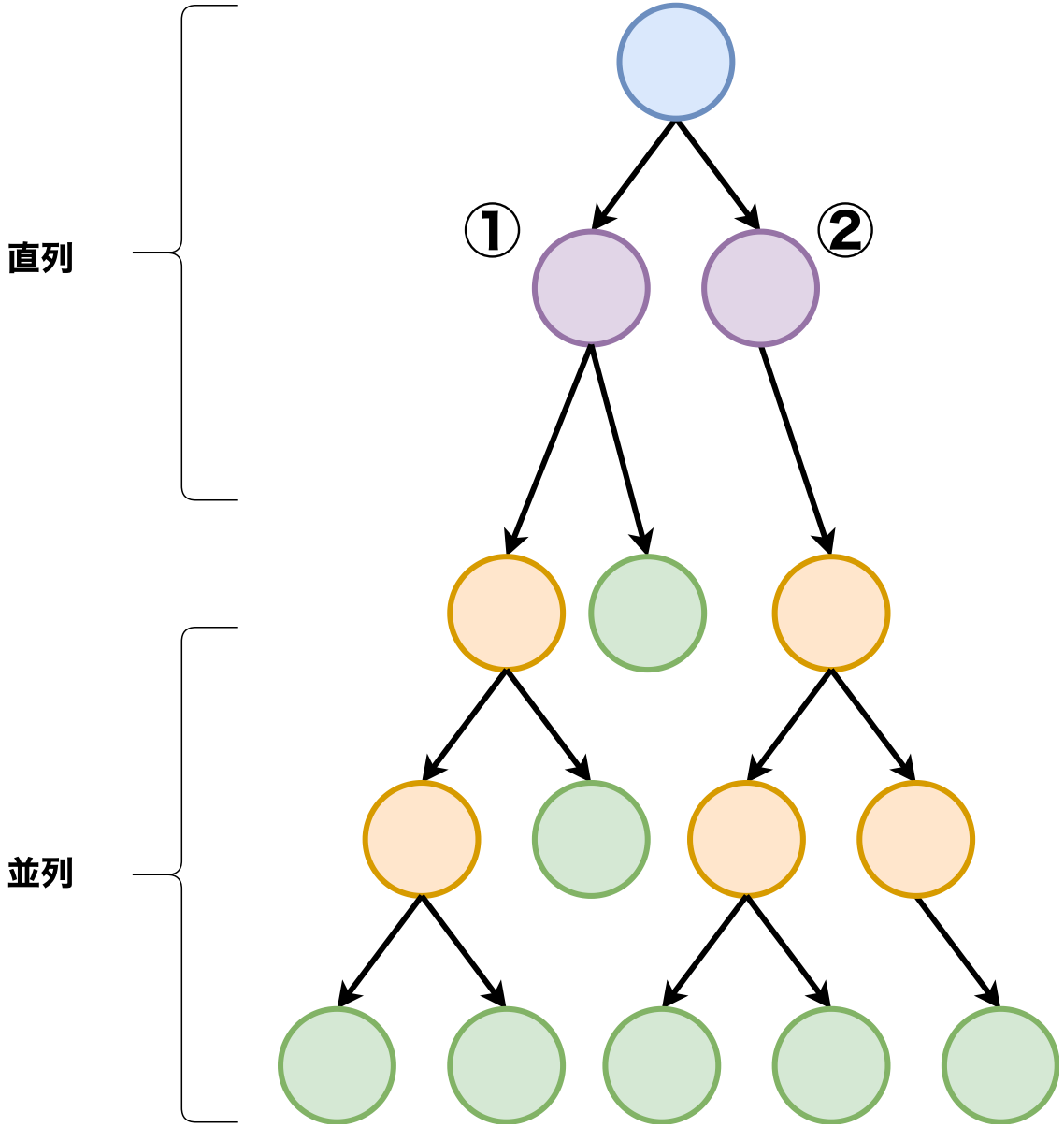


性能比較

	ふつう	並列化 ②
2手読み	2.744 [s]	1.534 [s]
3手読み	19.88 [s]	7.845[s]
4手読み	158.5 [s]	64.03 [s]

⇨ ✅ **2~3倍の高速化に成功！**

(+ cpu使用率のデモ)



$\alpha\beta$法 · · ·

枝刈りによる高速化

実装 / αβ法のアルゴリズム

$\alpha :=$ 自分の手番のノードにおける最大の評価値

$\beta :=$ 相手の手番のノードにおける最小の評価値　を保持して

- 相手番のとき
  αより小さい評価値のノードが見つかった場合
- 自分の番のとき
  βより大きい評価値のノードが見つかった場合

それ以上探索の必要はないので枝刈りしてよい

実装 / αβ法の効果

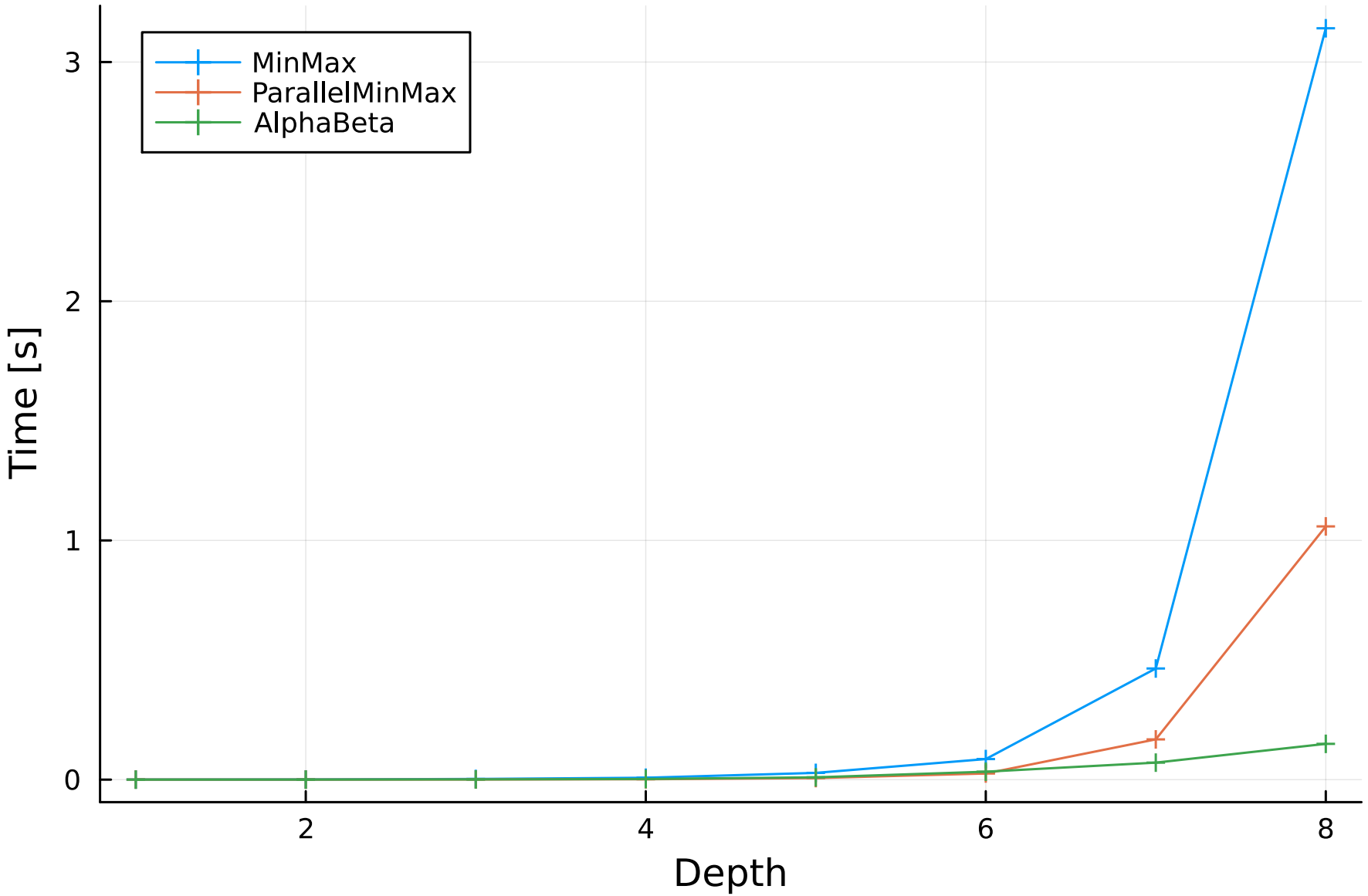
- 初手を8手読みしてみる

	MinMax法	$\alpha\beta$法
検討した手の数	455127	19801
時間 [s]	3.040 [s]	0.1545 [s]

⇨ **探索した手の数20倍以下**

⇨ ✅ **速度も20倍近く高速に**

実装 / αβ法の効果

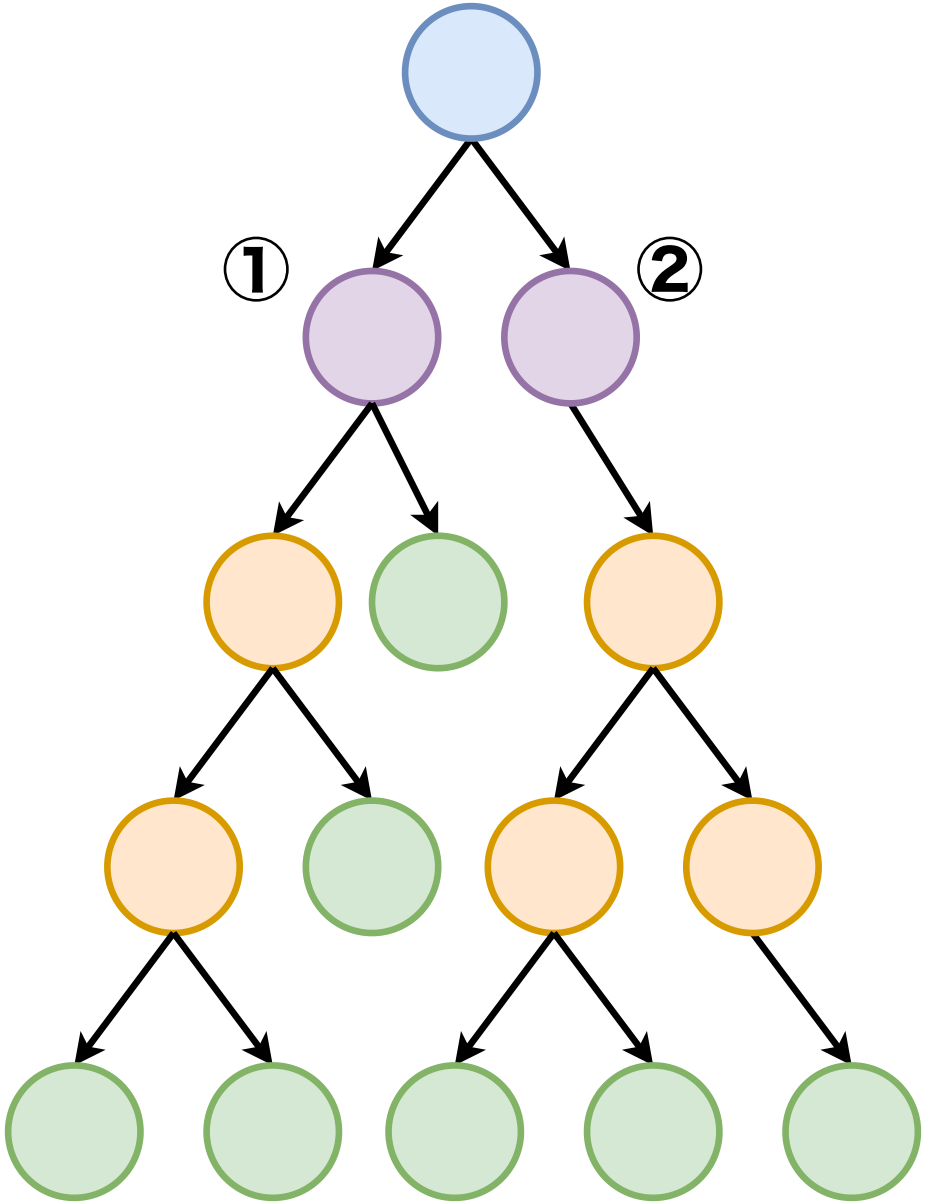


[MinMax法の場合]

各ノードを頂点とする
部分木の探索は独立

- 右図 ①と②のノードの評価は独立に計算できる

→ 並列化できそう



実装 / αβ法の並列化

- βカットの疑似コード

```
for hand in hands
    _score = search(next(hand))
    if _score < α
        break
    else
        score = min(score, _score)
        β = min(β, _score)
    end
end
```

⇨ ⚠️ある探索が他の探索に依存している！

(安全かつ高速に並列化するのにはひと工夫必要)

YBWC法

(Young Brothers Wait Concept)

実装 / αβ法の並列化

1. 子ノードのうち最初のw個だけ評価して
   αとβを更新する

2. 残りのノードを並列に評価する
   (ここでαとβを更新しない)

⇨ 探索幅はあまり狭まらないが、残りの部分は並列化できる
⇨ とくに$w = 1$の場合を**YBWC法**という(らしい)

(ハイパーパラメータ w を適切にチューニングする必要がある)

✅ 並列化により高速に！

	$\alpha\beta$法	$\alpha\beta$法 + RootSplit	$\alpha\beta$法 + RootSplit + YBWC
2手読み	2.740 [s]	2.631 [s]	1.912 [s]
3手読み	10.14 [s]	11.06 [s]	7.514 [s]
4手読み	38.01 [s]	17.94 [s]	15.75 [s]

1. 背景

2. 実装

3. まとめと課題

結論

- MinMax法, $\alpha\beta$法ともに並列化による高速化を確認した
  - $\alpha\beta$法は単純な並列化はできないので工夫した手法を用いる必要があった

- 一方で、単純に全てを並列化すればよいというわけではないこともわかった(MinMax法の並列化)
  - 並列化によるオーバーヘッドが大きくなると逆に遅くなる
  - 並列化による恩恵が大きいところを見極める必要がある

今後の課題

- ハイパーパラメータについて検討しきれていない
  - 並列化する深さをどこまで取るか
  - $\alpha\beta$法のYBWC法におけるw
- その他の探索手法についての検討
  - ゲームAIの強力な手法であるモンテカルロ木探索など