

Available captures for the rook; a performance analysis on a parallelized version of the problem

Louis Boulanger

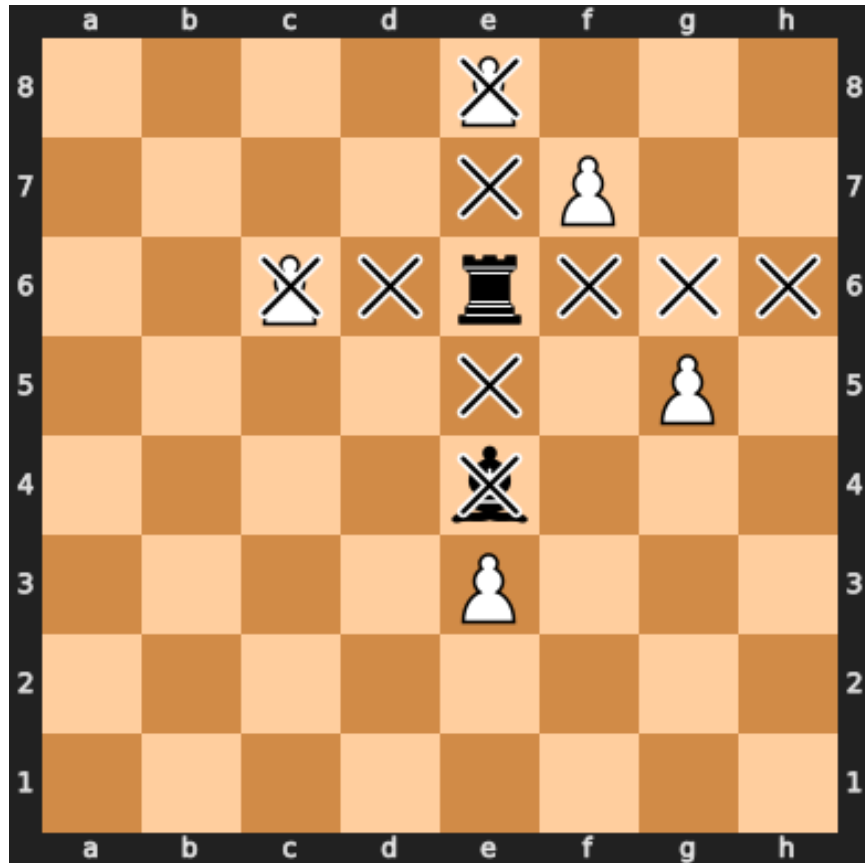
December 26, 2020

Contents

1	Introduction	1
2	Base algorithm	2
3	First attempt at parallelization using Rayon	3
3.1	Theoretical performance analysis of the parallel version . . .	4

1 Introduction

This project aims to implement a solution for the problem of the number of available captures for the rook on a chessboard: on a chessboard, where pieces are either a **black rook**, **black bishops** or **white pawns**, how many pawns can the rook capture?



In this example, the rook can only capture 2 pawns: e8 and c6. The pawn on e3 is blocked by the bishop on e4.

2 Base algorithm

The base algorithm is quite simple: given a board and the position of the rook, we need to iterate on the squares in the 4 directions: if the first piece encountered is a bishop, there is no capture on this position; and if it's a pawn, there is a capture. The result is the sum of the number of captures.

```
/// Computes the number of pawns the rook can capture in the
/// board's current configuration.
pub fn get_rook_captures(&self) -> usize {
    let start = self.get_rook_position();
```

```

// Looking at all directions (up, down, left, right):
Direction::all()
    .iter()
    .map(|d| {
        match start
        // we look at the line in that direction
        .line(*d, self.size)
        .iter()
        .filter_map(|p| self.get_piece(p))
        // and get the first piece on the line:
        .next()
        {
            // If there aren't any, then there
            // is no capture
            None => 0,
            Some(k) => match k {
                // If it's a bishop, no capture either
                PieceKind::Bishop => 0,
                // If it's a pawn, we capture it
                PieceKind::Pawn => 1,
                // If it's another rook, no capture
                PieceKind::Rook => 0,
            },
        }
    })
    // ... and we sum the number of captures.
    .sum()

```

3 First attempt at parallelization using Rayon

We can attempt to parallelize the algorithm in a straightforward manner, by looking at the 4 different directions in parallel. This version of the algorithm is quite similar to the sequential one: using Rayon's powerful parallel iterators, we can simply iterate in parallel on the directions.

```

/// Computes the number of pawns the rook can capture in the
/// board's current configuration, in parallel.
pub fn get_rook_captures_par(&self) -> usize {
    let start = self.get_rook_position();

```

```

Direction::all()
  // We use a parallel iterator here
  .into_par_iter()
  .map(|d| {
    match start
    .line(d, self.size)
    .iter()
    .filter_map(|p| self.get_piece(p))
    .next()
    {
      None => 0,
      Some(k) => match k {
        PieceKind::Bishop => 0,
        PieceKind::Pawn => 1,
        PieceKind::Rook => 0,
      },
    }
  })
  .sum()

```

3.1 Theoretical performance analysis of the parallel version