

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

Louis Boulanger

January 3, 2021

Contents

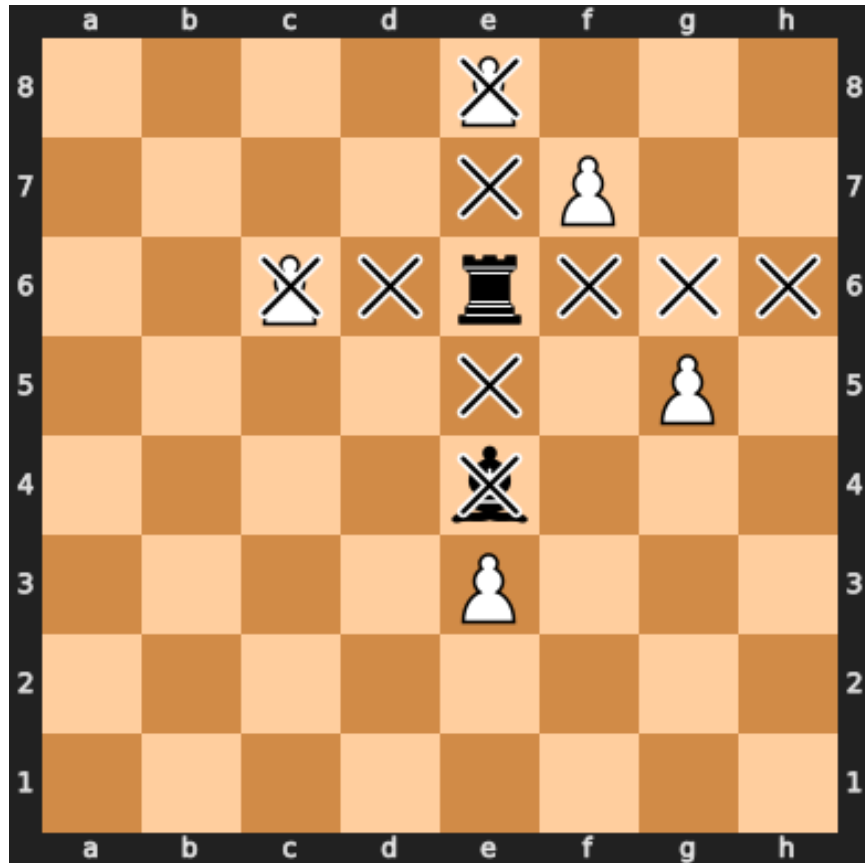
1	Foreword	1
2	Introduction	1
3	Base algorithm	2
4	First attempt at parallelization using Rayon	3
4.1	Theoretical performance analysis of the parallel version . . .	4
5	Adding more rooks	4

1 Foreword

This document has been generated from an **org-mode** file. The source can be found here (Source), and a PDF version here (PDF). I aimed to have both the GitHub-ready org-mode file and the PDF to be readable, but unfortunately, GitHub doesn't support mathematical expressions to be rendered natively on the preview; so, in order to view correctly the mathematical formulas throughout the analysis, I recommend reading the PDF version.

2 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.

3 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()
}

```

4 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. This function
/// assumes that there is only one rook on the board.
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()
}

```

4.1 Theoretical performance analysis of the parallel version

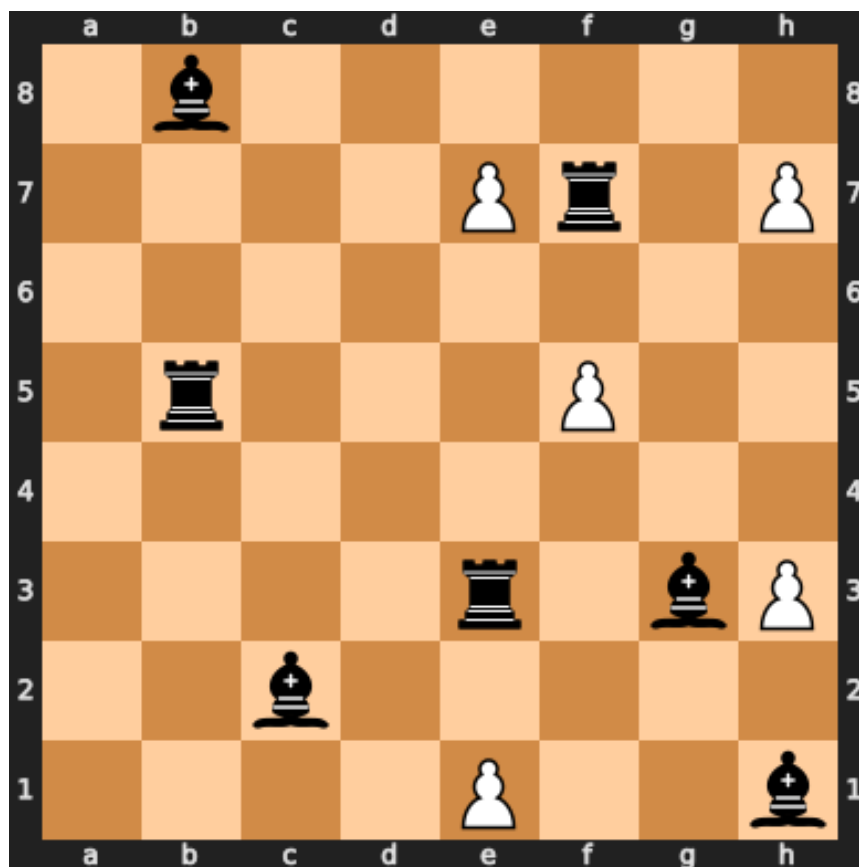
Let's analyze the parallel algorithm. We first formally define a board as a 2-dimensional space of size N^2 . There are 4 parallel branches, each iterating on successive squares of the board. Deciding on the capture is $\mathcal{O}(1)$, so the exploration of a line takes $\mathcal{O}(N)$ at most.

We then have a work of $W = \mathcal{O}(4N)$, and a depth of $D = \mathcal{O}(N)$. It is obvious here that having more than 4 processors for the task is going to be detrimental for the performances.

5 Adding more rooks

The previous example isn't really scaleable: since we can only consider 4 directions at once, the parallel algorithm can only be at most 4 times more efficient than the sequential one. Let's consider the case where there is more than one rook on the board, and we need to count the total number of captures for all rooks. When two rooks can capture the same pawn, the

capture is only counted once: we want the number of pawns that can be captured.



In this example, we have three rooks:

- **f7** can capture the pawns *e7*, *f5* and *h7*: **3** captures.
- **b5** can capture the pawn *f5*, but since it has already been counted previously, it is not counted here. **3** captures.
- **e3** can capture *e7* (but already counted) and *e1*: **4** captures

In total, there are 4 pawns that can be captured out of the 5 on the board: the pawn on *h3* is blocked by the nearby bishop.