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

Louis Boulanger

February 15, 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**
5.1 Sequential algorithm 6
5.2 Parallel algorithm 7

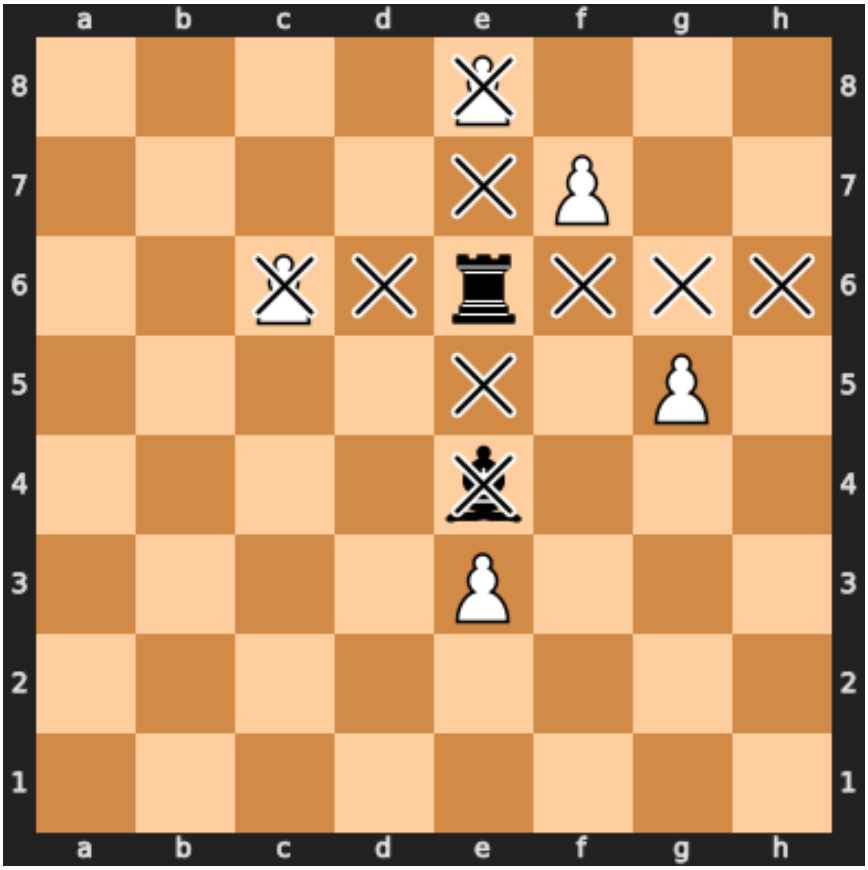
6 Benchmarking and analysis **8**

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.

```rust
/// 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()
}
```

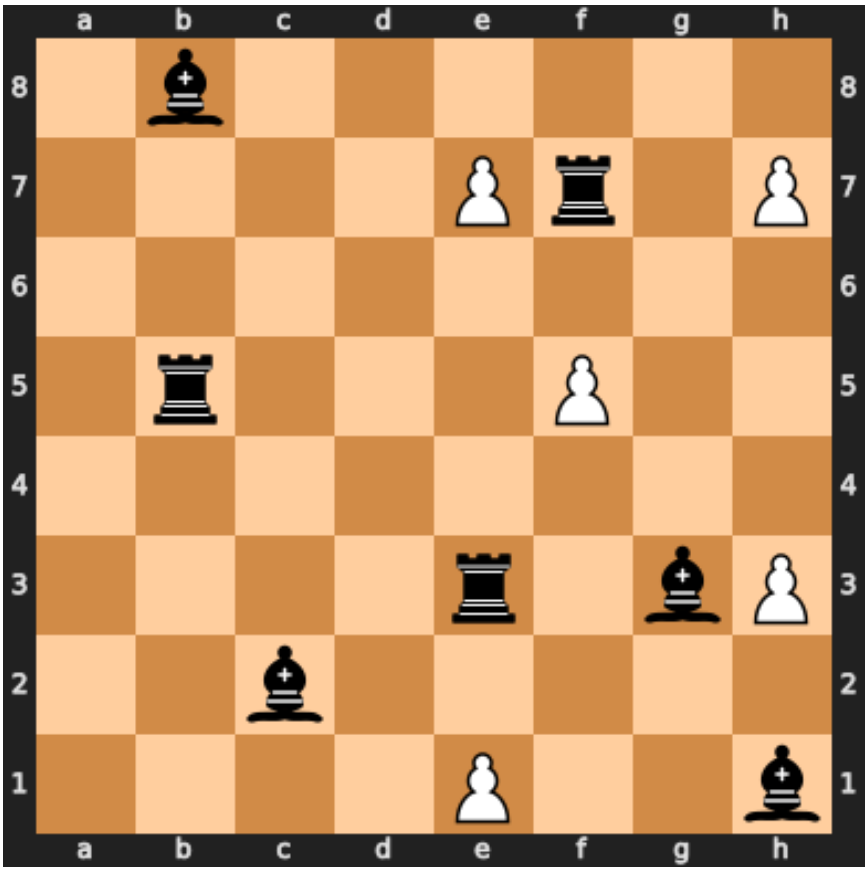
4.1 Theoretical performance analysis of the parallel version

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.

5.1 Sequential algorithm

We already designed the algorithm for the case with only one rook previously: since the problem only changes the number of rooks on the board, we can re-use the previous algorithm, and apply it to each rook on the board. Let's look at the sequential version first.

```
/// Calculates and returns the total number of captures
↪  available
/// for all the rooks on the board. If two rooks can capture
↪  the
/// same pawn, the capture is counted only once.
pub fn get_rooks_captures(&self) -> usize {
    let rooks = self.get_rooks_positions();

    rooks // For all rooks
        .iter()
        .map(|start| {
            Direction::all()
                .iter()
                .filter_map(|d| {
                    match start
                        .line(*d, self.size)
                        .iter()
                        .filter_map(|p|
↪  self.get_piece(p).map(|k| (k, p)))
                        .next()
                    {
                        None => None,
                        Some((k, p)) => match k {
                            PieceKind::Bishop => None,
```

```
                            PieceKind::Pawn => Some(*p),
                            PieceKind::Rook => None,
                        },
                    }
                })
                .collect::<HashSet<_>>()
        })
        .fold(HashSet::new(), |a, b|
↪   a.union(&b).copied().collect())
        .len()
}
```

The code is the same as the previous one, only that we apply it to all rooks on the board. In order to take duplicate captures into consideration, we collect the capture's position and not its presence; then, we aggregate these positions into a 'HashSet'. This way, duplicates are eliminated (using the union operator on sets). If we consider R rooks, we now have a complexity of $\mathcal{O}(4RN)$.

5.2 Parallel algorithm

Now that we have a sequential algorithm for our new problem, it's easy to convert it into a parallel one using Rayon.

```
/// Calculates and aggregates the number of captures for all
↪   the
/// rooks on the board. When two rooks can capture the same
↪   pawn,
/// only one capture is counted. The strategy here is to
/// parallelize on the rooks and not on the 4 directions.
pub fn get_rooks_captures_par(&self) -> usize {
    let rooks = self.get_rooks_positions();

    rooks
        .par_iter()
        .map(|start| {
            Direction::all()
                .iter()
                .filter_map(|d| {
                    match start
```

```
                        .line(*d, self.size)
                        .iter()
                        .filter_map(|p|
↪  self.get_piece(p).map(|k| (k, p)))
                        .next()
                    {
                        None => None,
                        Some((k, p)) => match k {
                            PieceKind::Bishop => None,
                            PieceKind::Pawn => Some(*p),
                            PieceKind::Rook => None,
                        },
                    }
                })
                .collect::<HashSet<_>>()
        })
        .reduce(HashSet::new, |a, b|
↪  a.union(&b).copied().collect())
        .len()
}
```

The code is identical to the sequential version, only that we parallelize on the different rooks' positions. The 4 different directions aren't considered in parallel anymore. With R rooks, we have a work of $W = \mathcal{O}(4RN)$, and a depth of $D = \mathcal{O}(4N)$. We can see now that it's much more scalable: with the execution time being $max(\frac{W}{d}, D)$, the algorithm allows up to $p = R$ processors to work in parallel.

6 Benchmarking and analysis

In order to benchmark the results, I used the `criterion` crate, which provides ways to benchmark and analyze the results of our computations. The benchmarks compare four different versions: the single rook, in parallel and sequential; and the multiple rooks, in parallel and sequential. Those four algorithms are run on boards of size 32 up to 288, by increments of 32. The parallel code was run with 4 threads, on a machine running Linux with Rust 1.48.0 with 4 physical cores (8 virtual).

The boards are generated by filling half the squares with an equal chance of a rook, a pawn or a bishop.

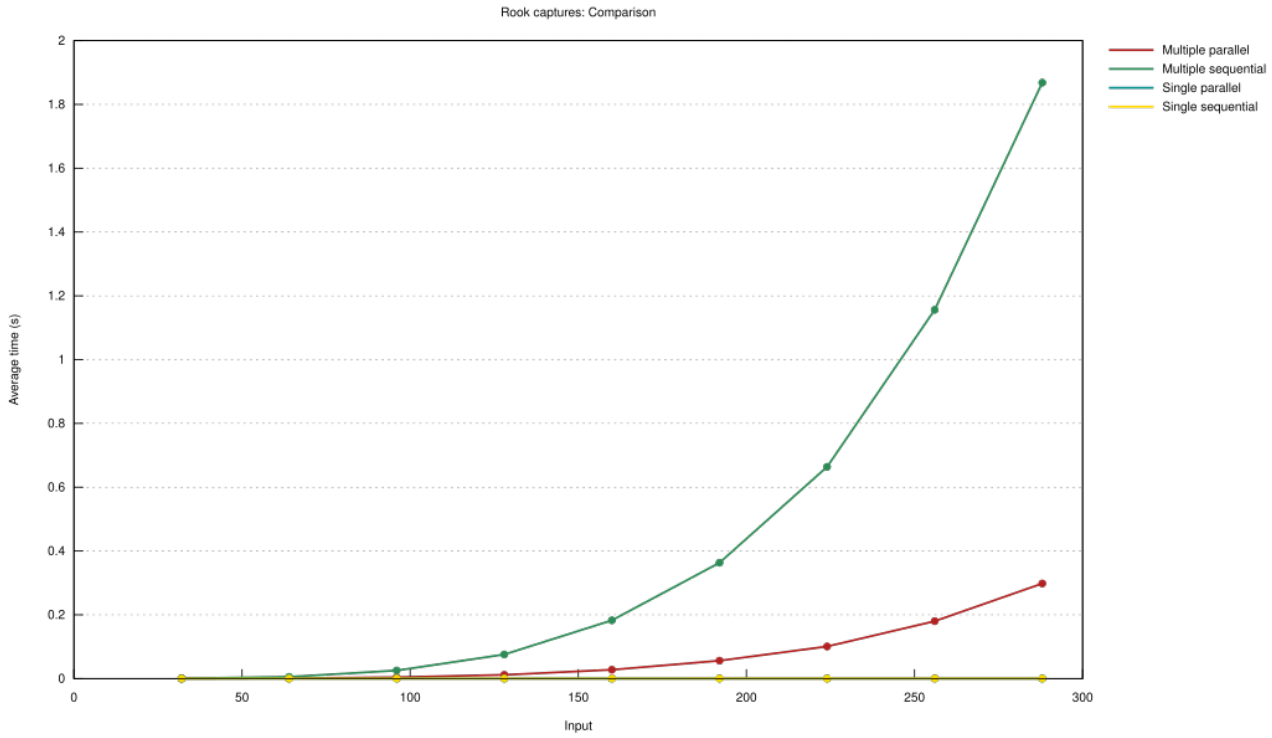


Figure 1: The four algorithms compared. The version with a single rook is too fast to be able to be compared with multiple rooks on large chessboards.

The speedup for the multiple rooks problem is extremely high when using the parallel version, as the number of rooks to check increases as the board increases itself: we can see here that as the number of rooks increases, the sequential version increases accordingly, as it needs to check in 4 directions for each additional rook, with potentially more squares to go through each time; but with the parallel version, the time of the computation increases in a much slower way, as the rooks are checked in parallel.