Empirical Analysis of Rayon Usage in Rust

William Fedrick Rosellón Prakoso[1]

Supervisor(s): Andreea Costea, Ruben Backx[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: William Fedrick Rosellón Prakoso
Final project course: CSE3000 Research Project
Thesis committee: Andreea Costea, Ruben Backx, Przemyslaw Pawelczak

Abstract

Rust provides performance, memory safety and the promise of "fearless concurrency", owing to its robust ownership system. Rayon has become a widespread library for data-parallelism within Rust, enabling programmers to incorporate concurrency into their programs seamlessly. This work presents the most common Rayon usages across the top 1000 open source Rust repositories on Github, measured by stargazer count and filtered for usage of Rayon. We perform a manual analysis on 5 repositories, identifying themes within selected instances. We contextualise our themes to characterize how Rayon is used for complex workloads, where unsafe code is combined with Rayon, and where parallel iterators are insufficient.

1 Introduction

Rust is a programming language that has gained adoption, consistently found as most admired programming language by Stack Overflow [1], [2], [3]. Rust provides performance, memory safety and the promise of "fearless concurrency", detection of concurrency errors at compile time, through its type system and ownership model [4]. Rayon is a library for data-parallelism in Rust which has seen widespread adoption, appearing on the crates.io most downloaded list [5]. Though the Rayon documentation describes best practice, the real world usage, interaction with complex, data dependent workloads and unsafe code remain unstudied.

Through an investigation of Rayon use cases, focusing on limitations that developers may face when using Rayon, we aim to determine how Rayon is used in open-source Rust projects to realize data-parallel concurrency. To facilitate this research, a frequency analysis was performed followed by a manual analysis. The manual analysis consisted of inspection of instances of Rayon use from chosen repositories. The analyses aimed to address the following questions,

RQ1. Which Rayon primitives are most commonly used in practice?

RQ2. How is Rayon used for workloads that are not trivially parallel, and what techniques are used to manage this complexity?

RQ3. How is unsafe code used in combination with Rayon, and when is it necessary?

RQ4. When are parallel iterators alone insufficient, and what Rayon or Rust primitives are used to adapt to this?

In section 2 we give the necessary background in Rust, Rayon and a brief overview of prior work. Next section 3 will go into detail on how the repositories were selected and the analysis was performed. In section 4 we present our results, followed by section 5 where we address a number of threats to validity. We then detail related work in section 6 and present our final conclusions in section 7.

2 Background

We now introduce the necessary background, first giving a brief overview of concurrency in Rust. We focus on the ownership and borrowing system, unsafe code and basic compiler rules. Then we introduce Rayon as a library for Rust, including some common methods and example usage.

2.1 Concurrency in Rust

Rust contains three ownership rules, stating that each value has an owner, there is only one owner of a value at a given time and a value is dropped if its owner goes out of scope [6].

Passing a value to a function or assigning a variable will move the ownership. As an alternative to transferring ownership Rust allows borrowing. This allows us to pass a reference to a variable. The Rust compiler ensures that there only exists either one mutable reference or multiple shared references at any time. Through these rules Rust can provide many safety guarantees at runtime, such as prevention data races and memory leaks.

This places limitations on developers when writing concurrent code. Consider Listing 1. We have a shared table, where the task to parallelize is independently reading each table entry, modifying it and writing back. The compiler will not allow the shared mutable references and give an error, despite the operations being independent.

Listing 1: Compiler error due to multiple mutable references

```
let mut table = vec![1, 2];
thread::scope(|s| {
  s.spawn(|| {table[0] *= 2;});
  s.spawn(|| {table[1] *= 2;});
});
// Error: cannot borrow `table` as mutable
     more than once at a time
```

Listing 2 shows a solution in safe Rust. We use a Mutex and synchronize writing to the table. Though this allows sharing the reference, it adds the cost of synchronization.

Listing 2: Use of synchronization to share mutable references

```
let table = Arc::new(Mutex::new(vec![1,
    2]));
// Threads must acquire lock to write

thread::scope(|s| {
  s.spawn(|| {table.lock().unwrap()[0] *=
    2;});
  s.spawn(|| {table.lock().unwrap()[1] *=
    2;});
});
```

Alternatively, we may use unsafe Rust. Unsafe Rust allows programmers to declare code as `unsafe`, allowing additional actions for which the compiler cannot guarantee memory safety. For example, it allows dereferencing of pointers, Listing 3 incorporates this to avoid synchronization.

Listing 3: Shared mutable references through unsafe code

```
let mut table = vec![1, 2];
let addr = table.as_mut_ptr() as usize;
```

```
3
4  thread::scope(|s| {
5    for i in 0..2 {
6      s.spawn(move || unsafe {
7            let p = addr as *mut i32;
8            *p.add(i) *= 2;
9      });
10   }
11 });
```

This is a notable area for study within Rust, as it places the responsibility for memory safety on the programmer. Rayon addresses these constraints and encapsulates the required unsafe functionality in a safe interface. For example, the shared table state may be managed using parallel iterators, specifically using the method `par_iter_mut()`, as shown in Listing 4.

Listing 4: Managing shared state using Rayon parallel iterators

```
1  let mut table = vec![1, 2];
2  table.par_iter_mut().for_each(|x| *x *= 2)
       ;
```

2.2 Rayon

Data-parallelism is running a single operation on multiple pieces of data in parallel. Limitations on single core hardware, and thereby the rise of multicore, has made data-parallelism a powerful tool [7]. Rayon is a Rust library that implements data-parallelism through a parallel iterator abstraction [8]. This allows programmers to use a functional paradigm, mirroring what regular Rust iterators provide.

Listing 5: Reducing a list in parallel

```
1  let sum = (1..=100)
2    .into_par_iter()
3    .reduce(|| 0, |a, b| a + b);
```

Some important functions Rayon provides are `spawn()`, `scope()` and `join()`.

Listing 6: Conditional dispatching using scope()

```
1  rayon::scope(|s| {
2    if has_left() {s.spawn(|_| process_left
       ())};
3    if has_right() {s.spawn(|_|
       process_right())};
4  });
```

Above we see that `spawn()` runs one closure in the thread pool. `scope()` creates a local scope which we can spawn closures into. We note that the underlying implementation to distribute work is the same for both abstractions, despite the different interfaces. Thus parallel iterators provide data-parallelism for collections, while directly interacting with the thread pool allows parallelism over irregular task structures.

3 Methodology

We perform an empirical study of Rayon usage within Rust, with the aim to understand Rayon usage in practice. We use a purposive sampling method in order to find instances of Rayon usage that reflect non-trivial workloads, unsafe code in conjunction with Rayon and limitations of Rayon's primary abstraction, parallel iterators. This was chosen to increase the depth of analysis, and to promote discovery over achieving a fully representative sample. We define instances as a function, closure or method that calls a Rayon primitive. We define a Rayon primitive as a single call to a function, method or usage of a struct (e.g. par_iter(), ThreadPool).

3.1 Repository Selection and Frequency Analysis

The initial candidate pool for the study was the top 1000 GitHub Rust repositories, ranked by stars. This was chosen to target real-world relevant projects. The projects were then filtered for usage of Rayon, through identifying Rayon as a listed dependency in the Cargo.toml files of each repository. This resulted in 172 repositories. 155 of these repositories were identified to contain Rayon primitives, with other projects containing Rayon as a transitive rather than direct dependency. The set of primitives searched was derived from the Rayon documentation, and refined iteratively to minimize false positives, such as `map()` or `collect()` which appear in the Rust standard library.

An automated frequency analysis was performed over the 155 identified repositories using ripgrep [9]. The number of repositories containing a primitive and the total usages of each primitive were considered.

Further, purposive sampling was performed on the 155 identified repositories, where the top 5 were selected according to the following criteria:

- GitHub star count
- Number of unique primitives used
- Usage of lower level primitives (e.g. join(), scope())
- Usage of unsafe in files containing primitives

The criteria were combined in an arithmetic sum. Github star count, unique primitives and usage of unsafe were normalized in the range of appearing values to obtain values from zero to one. Usage of lower level primitives was assigned zero if no primitives were present, else one. This selection aims to bias the selected repositories towards containing many primitives, including lower level primitives, and the usage of unsafe with Rayon.

3.2 Repository Analysis

For each of the 5 repositories selected for analysis, the domain, size (lines of code) and a brief description was recorded. Then for each primitive occurring in the repository, 5 instances were analysed. If 3 consecutive instances do not provide new information, the next primitive is considered. Instances were first analysed to determine presence of:

- A non-trivial workload, for example containing data dependencies or ordering constraints
- Usage of unsafe with files containing primitives
- A limitation of parallel iterators, such as additional concurrency mechanisms or data duplication

A non-trivial workload was defined as any case where tasks are not fully independent. Then a textual analysis was recorded guided by the questions identified as relevant.

3.3 Thematic Analysis

We perform a thematic analysis to find themes across the instances. We follow a similar methodology as described by Ahmed et al., adapting the six phases presented by [10]. We define three relevant phases to our analysis, identified from the six presented by Ahmed et al.

The first step is the generation of initial codes. We use the closed questions from subsection 3.2 to guide the generation of codes from the textual analysis. The aim for our initial codes was to create labels relevant to the stated research questions.

Following this, we group similar codes, aiming to capture more general themes that identified during initial coding. We use the same metrics as for generating the initial codes to find similarities between codes.

During our final stage we ensured internal coherence of themes, meaning that instances within the same theme are similar. Further, we ensured themes were distinct from one another, requiring splitting and merging of themes.

Ahmed et al. suggest determining whether to generate semantic or latent codes. Semantic codes are identified from the explicit meaning that can be extracted from the data. Latent codes interpret the data to find underlying intent and motivation [11]. We aim to generate latent codes, as it was observed during the repository analysis that instances did not need to share superficial similarities to use similar techniques. Further, we allowed assigning of multiple codes to an instance, to investigate which patterns co-occur frequently.

Finally, each theme is given a name to reflect its meaning. These names should convey both the theme and how this relates to our questions.

4 Results

We now present our findings from the frequency analysis, repository ranking and thematic analysis. We address research **RQ1** in our frequency analysis, and focus on **RQ2-4** in the thematic analysis.

4.1 Frequency Analysis

In order to answer **RQ1**, we performed a frequency analysis on the entire set of repositories. We consider two criteria: the number of unique repositories using a primitive and the total usage count over all repositories.

Table 1: Number of unique repositories containing primitives

Primitive	Count
.par_iter	114
.into_par_iter	98
rayon::ThreadPoolBuilder	73
.par_bridge	39
rayon::spawn	31
rayon::scope	25
rayon::join	20

Table 2: Primitive usage counts

Primitive	Count
.par_iter	1562
.into_par_iter	1253
rayon::ThreadPoolBuilder	245
.par_chunks	195
.par_bridge	119
rayon::spawn	94
rayon::join	91
rayon::scope	71

We see the first criteria in Table 1, the most common primitive is `.par_iter()`, the standard entry point for iterators, mirroring the `.iter()` function provided in the Rust standard library. This was found in 114 of the repositories. This seems to confirm that parallel iterators are the most frequent way to interact with Rayon. Second we find `.into_par_iter()`, which similarly mirrors `.into_iter()`, and occurred in 98 repositories.

Third is `ThreadPoolBuilder`, occurring in 73 repositories. This is a unique primitive, as it is often used to customize a thread pool. In the repositories analysed, we see a common use case is setting the number of threads in the global thread pool.

Both `spawn` and `scope` are used for submitting tasks to the thread pool, but we note that `spawn` was used in 31 repositories, and `scope` in 25. As scope is only necessary in more complex cases we may have expected a larger discrepancy. Additionally, for `join` we find usage in fewer repositories than `scope`, occurring in only 20. As both spawn and scope have more occurrences than `join`, this suggests that spawning many local tasks is a more frequent requirement for developers than running two tasks in parallel.

If we instead consider the total usage count of each primitive across all repositories, the findings are similar except that join is more frequently used than scope, having 91 and 71 occurrences respectively. `join` being used more frequently in absolute count but in fewer repositories suggests that it is used more intensively in those projects.

4.2 Selected Repositories

The five selected repositories for analysis are pola-rs/polars, astral-sh/uv, zed-industries/zed, ruvnet/RuVector and typst/typst. In Table 3 we see the computed scores of these repositories. As detailed in section 3 this score is the arithmetic mean of the normalized value for each criteria.

Table 3: Top repositories by normalized even weight ranking

Repository	Unsafe	Primitives	Low level	Stars	Score
pola-rs/polars	390	10	Yes	38364	0.760
astral-sh/uv	36	4	Yes	84298	0.508
zed-industries/zed	154	3	Yes	81450	0.487
ruvnet/RuVector	120	7	Yes	3886	0.478
typst/typst	0	6	Yes	53263	0.469

Our first observation is that all selected repositories contained lower level primitives, suggesting that projects that intensively use Rayon frequently reach for its lower level primitives. Further, there are large differences in how a repository obtained its ranking. Polars, a dataframe library for Rust, performed well on all criteria, leading to by far the highest score.

During the analysis, it was observed that this was in part due to the repository making extensive use of unsafe to implement efficient data structures. Notably, typst does not contain a single usage of unsafe in Rayon files, though usage can be found in other locations within the repository.

4.3 Thematic Analysis

We now detail some of the most relevant themes identified throughout the analysis, including select examples. Each theme will be considered in the context of our research questions, taking notice of themes that may co-occur frequently.

Ordering constraints

A theme that was identified numerous times was **Ordering Constraints**. This theme is notable, because it frequently coincides with the theme **Parallel iterator limitation** and **Partial concurrency**. This theme relates to **RQ2** and **RQ4**. When ordering constraints are present, lower level primitives may be needed to enable parallelism or may enhance the readability of code, despite parallel iterators being Rayon's idiomatic solution.

Listing 7 is an extract from the Polars repository. Its function is to infer the schemas of sources in parallel, then merge the results. There is a constraint on the merging of the schemas due to lack of commutativity, as detailed in the comment. The first result must be merged last with all other results. As a result, parallel iterators are insufficient and the programmer must use join to parallelize the operations. We note that, in the `join` call, the first closure only performs inference of the first schema, while the second closure performs both inference and merging of the remaining schemas.

Listing 7: Ordering constraints leading to partial concurrency

```
// Run inference in parallel with a specific merge order.
// TODO: flatten to single level once Schema::to_supertype is commutative.
let si_results = POOL.join(
  || infer_schema_func(0),
  || {
    (1..sources.len())
      .into_par_iter()
      .map(infer_schema_func)
          .reduce(|| Ok(Default::default()
    ), merge_func)
  },
);

let (inferred_schema, estimated_n_rows) =
    merge_func(si_results.0, si_results.1)
    ?;
```

As the join in Listing 7 is on two different operations with distinct return types, this code may not easily be expressed using parallel iterators. In contrast, consider Listing 8, to rewrite this using parallel iterators we can create a two-item collection, and directly apply parallel iterators. The different return types in Listing 7 prevent a similar change. The choice to still use join in Listing 8 suggests simplicity over idiomatic usage.

Listing 8: `join` usage where Parallel iterators are possible

```
let (result_a, result_b) = POOL.install(|| {
  rayon::join(
    || self.left.evaluate_on_groups(df, groups, state),
    || self.right.evaluate_on_groups(df, groups, state),
  )
});
```

As such, we may conclude in reference to **RQ4** that the assumption, of parallel iterators, that there is no ordering to operations may cause expression of parallelism to be difficult using parallel iterators. As such we find in response to **RQ2** that instances with **Ordering constraints** may more frequently use lower level Rayon primitives.

Nested iteration

Another recurring theme was **Nested iteration**. This theme commonly occurred with chunked collections and variable workload sizes and relates to **RQ2** and **RQ4**. This was observed to result in decisions around the unit of parallelism, where multiple options exist in how to partition the data.

In Listing 9 we see an example of nested iteration over a variable number of items, taken from the RuVector repository, a high performance vector database. First `par_iter` is used to create a parallel iterator over the `probe_clusters` collection, then a for loop is used within the iterator to calculate a set of results. This example is notable as we see that in a high performance library the programmer chooses to parallelise in clusters, though it would be possible to achieve concurrency over the individual items.

As this repository is highly concerned with optimisation, it is a notable choice that coarser parallelism is favoured. This may suggest that for certain workloads there may not be benefit to additional parallelisation, with larger partitions improving performance. A clear explanation would be the limited number of worker threads available to Rayon, if all workers are occupied there will be no benefit to additional parallelism. Conversely, there may be costs in the scheduling of additional tasks.

Listing 9: Nested iteration in RuVector

```
let results: Vec<(VectorId, f32)> =
  probe_clusters
  .par_iter()
  .flat_map(|cluster_id| {
    let mut local_results = Vec::new();
      if let Some(list) = self
        .lists
        .get(cluster_id) {
          for entry in list.iter() {
            let dist = self.calc_distance(query, &entry.vector);
            local_results.push((entry.id, dist));
          }
      }
      local_results
    }).collect();
```

We see a contrasting example in the Polars repository, shown in Listing 10. We note that Polars is similarly concerned with optimisation. This instance employs nested parallel iterators over a hierarchical data structure, again using `flat_map` to aggregate results. Together, these examples support the notion that nested collections require developers to choose what the most efficient unit of parallelism is, dependent on the system they are working in.

We summarize that this theme can contain either nested calls to Rayon, or a combination of Rayon and non-concurrent iteration. We observe a consideration of the unit of parallelisation in these instances, and the appropriate size of tasks to saturate the thread pool without excess allocation costs. Our findings for **RQ2** are that nested data structures are often addressed using parallel iterators. In relation to **RQ4** we find that though this is not a parallel iterator constraint, it does cause programmers to consider the unit of parallelism chosen, and where to apply concurrency.

Listing 10: Nested parallelism using par_iter

```
self.chunks.par_iter().flat_map(move |arr| {
  let arr = &**arr;
  let arr = unsafe { &*(arr as *const dyn Array as *const Utf8ViewArray) };
  (0..arr.len())
    .into_par_iter()
    .map(move |idx| unsafe { idx_to_str(idx, arr) })
})
```

Custom parallel iterator implementation

An additional theme relating to **RQ2** and **RQ4** is **Custom parallel iterator implementation**. In order to extend the functionality of parallel iterators to more complex data structures, programmers may implement Rayon's parallel iterator interface manually. This enables idiomatic Rayon usage throughout a codebase despite proprietary types and implementations of concurrency.

Listing 11 is such a case, where we see a custom implementation of `par_iter`. Here the instance uses **Nested iteration** to facilitate concurrency over the chunked data structure. We observe that this extension allows the programmer to move the complexity from caller to the data structure. This is in contrast to what we observed in **Ordering constraints**, where programmers chose to apply lower level Rayon primitives.

In response to **RQ2** we find that for non-trivial workloads containing proprietary data structures it may be possible to encapsulate complexity, such as Nested iteration, by implementing the parallel iterator interface. With respect to **RQ4**, we observe that though proprietary data structures may be a limitation of parallel iterators, this may be circumvented through extension of the interface.

Listing 11: Custom parallel iterator implementation

```
pub fn par_iter(&self) -> impl ParallelIterator<Item = Option<Series>> + '_ {
  self.chunks.par_iter().flat_map(move |arr| {
    let dtype = self.inner_dtype();
    // SAFETY:
    // guarded by the type system
    let arr = &**arr;
    let arr = unsafe { &*(arr as *const dyn Array as *const ListArray<i64>) };
    (0..arr.len())
      .into_par_iter()
      .map(move |idx| unsafe {
      idx_to_array(idx, arr, dtype) })
  })
}
```

Unsafe for performance

The most notable theme encountered primarily concerning **RQ3** was **Unsafe for performance**. A frequent example is the usage of unsafe for writing a shared data structure. To do so, unsafe code can be used to directly write a memory address, circumventing Rust's borrow checker.

In Listing 12 we see an application of unsafe to write an output pointer. This example is also a case of nested iteration, as par_chunks is used to chunk the input followed by an inner loop over each index. In the snippet we see unsafe used twice, once to dereference the pointer and assign the value, and once to set the length.

Listing 12: Parallel writes to shared output buffer in perfect_sort

```
POOL.install(|| {
  idx.par_chunks(chunk_size).for_each(
  |indices| {
    let ptr = ptr as *mut IdxSize;
    for (idx_val, idx_location) in indices {
// SAFETY:
// idx_location is in bounds by invariant of this function
// and we ensured we have at least idx.len() capacity
      unsafe { *ptr.add(*idx_location as usize) = *idx_val };
    }
  });
});
// SAFETY:
// all elements are written
unsafe { out.set_len(idx.len()) };
```

Writing directly to an output buffer circumvents the local storage necessary to accumulate the results locally and may therefore be more efficient. This theme interacts with **Nested iteration**, as seen in Listing 12. The programmer has chosen to use an unsafe block to write directly to the output buffer, over processing of the entire list in parallel using, for example, `par_iter`.

To address **RQ3** we state that unsafe may be used with Rayon to realize a gain in memory or runtime efficiency. Additionally, we note that this theme may be seen with **Nested iteration**, allowing the programmer to circumvent local allocation in threads.

Unsafe for method call
Within instances containing `unsafe` the most common theme relating to **RQ3** was **Unsafe for method call**. This theme was labelled when the usage of an unsafe key word was not directly related to concurrency, but rather to call an unsafe method, access an unsafe data structure or perform some other non-concurrency related unsafe action.

We highlight this theme as it shows that the high counts for co-occurrence of `unsafe` and Rayon shown in Table 3 do not necessarily imply that concurrency in Rayon frequently requires `unsafe`. In response to **RQ3**, this implies `unsafe` usage with Rayon is frequently not concurrency related.

5 Threats to Validity

We now consider the limitations of our analysis, highlighting sections that may limit reproducibility, accuracy or introduce bias. We further briefly address each with a justification or attempted solutions.

Repository Selection
We considered the top 1000 GitHub repositories, ranked by stargazer count. As the considered repositories are widely used and well maintained, this does not capture smaller projects, private codebases and repositories on other platforms. As a result the results may not be reflective of usage within these projects.

An alternative was considered in crates.io [5]. GitHub was favoured, as some projects are not built as crates for distribution and thus exclusive to GitHub. Conversely, as crates.io requires GitHub to sign up, there are few projects on crates.io that are not found on GitHub [12]. As such, although not exhaustive, GitHub was deemed the most appropriate source.

Purposive Sampling
To find the dataset for manual analysis, we used purposive sampling. This allowed us to find repositories that were rich in the features we wished to analyse. As such, our findings should not be generalized to represent the typical usage patterns of Rayon, given that `.par_iter` was identified as the most commonly occurring primitive in our frequency analysis. This limitation was accepted, as we aimed to investigate complex behaviour, finding limitations of Rayon's parallel iterators, and where they interact with unsafe Rust.

Manual Analysis
The instance and thematic analyses were performed by a single researcher. This leads to potential bias being introduced in the generation of our results. Further, no peer review was performed, enhancing this risk.

This risk was reduced by the structure introduced in section 3 through the separation of generating observations from the identification of themes within these. In addition, the guiding questions provided for the instance and thematic analysis likely reduce the possible bias in findings.

Primitive Selection
Due to the method of searching for the primitives used, it was in some cases not possible to distinguish between standard library calls and Rayon primitives. For example `reduce` is a function present in both Rayon and the Rust standard library, like many other functional style calls. As such, including them resulted in unrealistically high counts for these values.

This limitation, in combination with the list being empirically determined to provide a sufficient overview, means that the results do not fully capture all calls made to Rayon. As such, it is possible that there are other primitives that were not included in the analysis, which may have changed the results of either the frequency analysis or the purposive sampling.

The impact this had on the results was minimized through iterative improvement, aimed at minimizing false positives. Further, the set of primitives was derived from the Rayon specification and the repositories themselves. As such we believe the results are still representative of the broader library usage.

6 Related Work

This section briefly details works relating to three key concepts within this study. We focus on works that provide contextual information on concurrency and parallelism, investigate unsafe usage in Rust or the notion of safe abstractions.

6.1 Parallel Programming

Foundational work in data-parallelism includes Pingali et al., who investigate regular and irregular parallelism, suggesting an alternative approach to the dependency graph which may be more suitable for irregular algorithms [13]. They present a data-centric view, where a program is described by actions on data structures, without considering their underlying implementation.

Abdi et al. focus on evaluating Rayon across benchmarks, finding that regular parallelism can be handled safely while irregular parallelism forces unsafe code [14]. Additional findings are that irregular parallelism is common within the benchmarks tested, and that avoiding unsafe code can have noticeable execution time costs.

6.2 Unsafe Usage in Rust

Unsafe usage within Rust has been empirically analysed by Astrauskas et al. [15]. Their key findings are that, though most unsafe code is simple and encapsulated, unsafe features are used extensively in practice. In particular, they find that interoperability and complex sharing of data structures are the most common motivations of unsafe code. Further, a comprehensive investigation into real-world unsafe usage within Rust was performed by Qin et al., developing a taxonomy of concurrency bugs and their causes [16].

6.3 Safe Abstractions

Safe abstractions are encapsulations of unsafe operations in a safe API, so that the caller can use the code as if it were safe. Jung et al. explore safe abstractions, and present RustBelt as a framework for formally proving safe encapsulation [4]. Miri, introduced by Jung et al., is a tool for detecting undefined behaviour within Rust programs, through analysis of Rust's Mid-level Intermediate Representation (MIR) [17]. Notably, Miri cannot exhaustively prove lack of undefined behaviour while RustBelt focuses on a formal proof to guarantee encapsulation — thereby assuring users that the code is safe to use.

7 Conclusions and Future Work

Rayon is a prevalent library for data parallel concurrency in Rust, but its idiomatic usage is unstudied. Through a preliminary frequency analysis followed by a manual analysis, we present the most common Rayon usage (**RQ1**), and a number of themes relating Rayon usage for non-trivial workloads (**RQ2**), the interaction of Rayon with unsafe (**RQ3**) and limitations of the parallel iterator interface (**RQ4**).

We find that `par_iter` is the most frequently used Rayon primitive, supporting parallel iterators as the idiomatic Rayon usage pattern. Common functions to interact directly with the thread pool are `spawn`, `join` and `scope`. Further, many repositories call `rayon::ThreadPoolBuilder` to configure Rayon's threadpool.

The analysis identified a number of themes. **Ordering constraints** were observed as a non-trivial workload. When ordering constraints were present, developers were inclined to use primitives beyond the parallel iterator interface, as such this theme presents a possible limitation of parallel iterators and relates to both **RQ2** and **RQ4**. Further, we find that developers may aim to extend the functionality of parallel iterators through **Custom parallel iterator implementation**. This allows for the usage of parallel iterators throughout an application while giving developers control over the underlying implementation of the concurrency. From this theme we conclude that the limitations of parallel iterators can be circumvented through custom implementations of the underlying `par_iter` method.

An additional theme in relation to **RQ2** and **RQ4** is **Nested iteration**. This theme displayed a usage of nested Rayon calls, or a combination of Rayon calls and non-concurrent iteration. A cause for this theme was nested data structures requiring multiple steps for iteration. We find this to be a non-trivial workload, as it may involve a dynamic number of tasks, that can be addressed using parallel iterators. We also observed developers considering the benefits of smaller units of parallelisation weighed against additional memory overhead when this theme was present.

In relation to **RQ3** we find two prevalent themes, **Unsafe for method call** and **Unsafe for performance**. **Unsafe for performance** was a notable theme for **RQ3**, showing that a frequent usage of unsafe with Rayon code is to allow for either memory or runtime efficiency gains. This may be seen with **Nested iteration**, for example to allow the programmer to aggregate output without thread local allocations. **Unsafe for method call** was noted because it explained the high co-occurrence of unsafe and Rayon found in the repositories.

A number of questions remain to be clarified. First, **Nested iteration** displays a trade off in some cases, where developers may need to choose the appropriate unit of parallelism. Though we have identified this consideration, further work may aim to develop tested and general guidelines as to the best implementation. This may require considering the expected environment, including available threads and the size of tasks. Relating to **Custom parallel iterator implementation**, we suggest future work consider the trade-off between extending the parallel iterator interface and implementation of a proprietary, application specific interface.

Finally we note that our analysis has focused on Rayon for realising data parallel concurrency, future work may aim to compare the realisation of this functionality in Rayon and other data parallelism libraries. Specifically, it may be investigated whether unsafe code is used in similar cases, and if the primary abstraction presented is stronger or weaker than Rayon's parallel iterators.

References

[1] "Stack overflow developer survey 2023," https://survey.stackoverflow.co/2023/, 2023.

[2] "Stack overflow developer survey 2024," https://survey.stackoverflow.co/2024/, 2024.

[3] "Stack overflow developer survey 2025," https://survey.stackoverflow.co/2025/, 2025.

[4] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Safe systems programming in rust," *Commun. ACM*, vol. 64, no. 4, p. 144–152, Mar. 2021. [Online]. Available: https://doi.org/10.1145/3418295

[5] "Crates.io downloads," https://crates.io/crates?sort=downloads, 2024.

[6] S. Klabnik and C. Nichols, "The rust programming language," https://doc.rust-lang.org/book/, 2018.

[7] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, 2005. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[8] "Rayon documentation," https://docs.rs/rayon/, 2024.

[9] "Ripgrep github repository," https://github.com/burntsushi/ripgrep.

[10] S. K. Ahmed, R. A. Mohammed, A. J. Nashwan, R. H. Ibrahim, A. Q. Abdalla, B. M. M. Ameen, and R. M. Khdhir, "Using thematic analysis in qualitative research," *Journal of Medicine, Surgery, and Public Health*, vol. 6, p. 100198, Aug. 2025.

[11] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006. [Online]. Available: https://doi.org/10.1191/1478088706qp063oa

[12] "Non-github account creation," https://github.com/rust-lang/crates.io/issues/326.

[13] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," *SIGPLAN Not.*, vol. 46, no. Jun., p. 12–25, Jun. 2011. [Online]. Available: https://doi.org/10.1145/1993316.1993501

[14] J. Abdi, G. Posluns, G. Zhang, B. Wang, and M. C. Jeffrey, "When is parallelism fearless and zero-cost with rust?" in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 27–40. [Online]. Available: https://doi.org/10.1145/3626183.3659966

[15] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428204

[16] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, Jun. 2020, pp. 763–779.

[17] R. Jung, B. Kimock, C. Poveda, E. S. Muñoz, O. Scherer, and Q. Wang, "Miri: Practical undefined behavior detection for rust," *Proceedings of the ACM on Programming Languages*, vol. 10, pp. 1383–1411, Jan. 2026.