Highjacking the Rust programming language for high performant in-situ analytics

UNDERGRADUATE THESIS

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Saurabh Manish RAJE
ID No. 2015A7TS0045P

Under the supervision of:

Dr. Bruno RAFFIN
&
Prof. Sundar Shan BALASUBRAMANIAM

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS
December 2018

Declaration of Authorship

I, Saurabh Manish Raje, declare that this Undergraduate Thesis titled, 'Highjacking the Rust programming language for high performant in-situ analytics' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ____________________________________

Date: ____________________________________

Certificate

This is to certify that the thesis entitled, "*Highjacking the Rust programming language for high performant in-situ analytics*" and submitted by Saurabh Manish Raje ID No. 2015A7TS0045P in partial fulfillment of the requirements of BITS F421T Thesis embodies the work done by him under my supervision.

Supervisor	*Co-Supervisor*
Dr. Bruno RAFFIN	Prof. Sundar Shan Balasubramaniam
Director of Reserch,	Professor,
INRIA Grenoble Rhone-Alpes	BITS-Pilani Pilani Campus
Date:	Date:

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS

Abstract

Bachelor of Engineering (Hons.)

Highjacking the Rust programming language for high performant in-situ analytics

by Saurabh Manish Raje

. . .

Acknowledgements

The acknowledg and the people to thank go here, don't forget to include your project advisor…

Contents

Chapter 1

The Rust programming language

1.1 The need for Rust

While modern day languages provide higher amounts of abstraction and ease of programming, they either come at the cost of safety, or runtime performance, or both. Systems development is still dominated by C/C++ due to the high performance that they offer. However, it is extremely easy to create fundamental loopholes in huge systems that may lead to invalid memory access and crash the entire system, if the programmer is lucky. In the worst case, there may be data corruption due to an invalid access that is extremely hard to trace.

Rust addresses the dire need for a fast and safe systems development language. It goes above and beyond in this regard to offer zero cost abstractions in the form of abstract traits and types that allow seamless integration of new features with an existing system. Furthermore, it also allows for functional programming with its lazy iterators API. This generates highly optimized machine code with performance comparable to C.

Compilable code written in Rust can never lead to data races, dangling pointers, double frees or memory leaks. This comes from a simple sacrifice of the mutable state. By imposing the invariant that each memory location must have a single mutable reference to it (that can not overlap in time with an immutable reference), all above problems are prevented. Furthermore, it allows the language to offer automated memory management at minimal overhead as opposed to traditional methods of garbage collection.

1.2 Some quirks of programming in Rust

The invariant of not having multiple mutable references to a memory object severely restricts syntactic expression of any given algorithm. The language hence suffers from a steep initial learning curve. The further sections shall elaborate some jargon that describes the memory model.

1.2.1 Ownership

The concept of ownership imposes the following rules[Cite the book here]:

- Each value in Rust has a variable that is its *owner.*
- This owner has mutable access to the value/memory object in question.
- There can be only one owner at a given time (read: scope).
- When the owner goes out of scope, the value is dropped.

Here it is important to note that all the analysis regarding ownership is carried out at compile time, and hence calls to drop objects are inserted by the compiler. This therefore provides memory safety and management at minimal runtime overhead.

1.2.2 Movement and copy

When variables are reassigned or passed around across functions, they adopt exactly one of two semantics, move and copy. Typically, any variable on the heap adopts the former while lightweight datatypes residing on the stack adopt the latter.

The move semantic changes the owner of the variable. This means that the scope of the variable is now the scope of it's new owner. The previous owner is an invalid reference to the variable, and the compiler would trigger compile-time errors for any attempt to use the previous owner.

The copy semantic on the other hand creates a deep copy of the data contained in the variable. We now have two different variables at separate locations in the memory, and having separate owners. The data that they contain, however, is the same.

1.2.3 Borrows and lifetimes

The former design requires unnecessary moves for situations (for example in case a function intends to read some data, it must be moved in and out). As an alternative, the language offers it's own notion of references. When a reference is created to any variable, it is called a borrow of that variable. Each reference has a lifetime that defines how long the reference is usable. This lifetime can never be more than that of the data referred to. Again, these checks are carried out at compile time. By a way of default, all references are immutable. However, by explicitly obtaining the reference with the *'&mut'* specifier, it is possible to get a mutable reference. [ADD A FIGURE OF BORROW CHECKER THROWING AN ERROR]

Furthermore, the compiler imposes a check to ensure that the lifetimes of two mutable references, (or that of one mutable reference with an immutable reference) do(es) not overlap.

1.2.4 Generic types and traits

The language offers powerful abstractions in the form of generic types and *traits.* A *trait* is essentially a set of functions (and possibly some types as well). Typically, one or more functions in the trait are not implemented. They have a particular signature to be followed as is. However, one or many functions can be defined and implemented in the trait. In such a case, after implementing just the abstract functions, the programmer has the rest of the functions of the trait at his disposal without having to implement them. The best example of this is the **Iterator** trait. After providing the implementation of the **next()** method, the entire functional API is available for use on the given iterator. This feature is analogous to interfaces in Java, for example. Traits can be extended from each other in order to create an object oriented hierarchy.

This dovetails with the notion of a generic type. Rust offers the possibility of using generic types in trait or function definition. A generic type is constrained by certain trait implementations. This essentially means that the signature is valid for any and all types that implement the given trait(s). This is extremely useful since any future types being introduced in the system would automatically have various functionalities supported with a few trait implementations. Consider the following example:

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];
    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

This function uses a generic type **T** and describes how to find the "*largest*" element out of a list of elements of type **T** (it will borrow this list). The type **T** is generic in that it can be anything that implements ParitalOrd and Copy traits.

Chapter 2

Concurrency in Rust

2.1 The Thread API

The Rust language provides a basic thread API that contains primitives *spawn* and *join*. *Spawn* creates a thread and passes it a closure - that it took in - as an argument. The *join* on the other hand forces the calling thread to wait on a given *JoinHandle* that was created by a spawned thread. Furthermore, it also offers synchronization primititves such as Mutexes, Condition Variables, Barriers and Reader Writer Locks.

2.2 Rayon

There is a more abstract (functional) alternative for shared memory parallelism called the **Rayon API**. As far as the programmer is concerned, a sequential functional code can be parallelized with nearly no additional effort. Internally, Rayon uses recursive task splitting to balance load across threads.

Specifically, it provides a parallel iterator on which various operations (same as in the sequential domain) can be performed one after the other, forming a pipeline. Internally,

- This iterator is split recursively into two halves until the balanced binary tree (of splits) reaches a specific depth.
- Each split is a task that is pushed on the stack.
- In case there are any idle threads, they steal a task from this stack.
- The stealer continues the above procedure, with the depth measure reset to 1.

[ADD A RAYON LOG HERE] Below is an example of a code that uses the Rayon API to parallelize the sum of a list of numbers.

```
extern crate rayon
use rayon::prelude::*;
fn main() {
    let inp = vec![1, 2, 3, 4, 5, 6, 7];
    println!("{}",inp.par_iter().fold_with(0, |acc, x| acc + x).sum::<u32>());
}
```

Chapter 3

Adaptive task splitting

3.1 An alternative to Rayon

In the context of high performance computing, the abstraction provided by Rayon currently comes at a high cost of task creation overhead. This is because the heuristic for splitting is decoupled from availability of workers. The fundamental issue being that tasks are created irrespective of whether there are idle threads to steal them or not. This heuristic has been designed to reduce the idle time as much as possible, but that unfortunately doesn't help if the total overhead of excessive task creation dominates.

It is also important to note that Rayon splits the complete iterator in half. However, it is trivially true that splitting half of the *remaining* iterator is better for load balancing. One can relate this to the sunk cost fallacy wherein the computation that has already taken place is now irrelevant to load balancing considerations from that point onwards.

As described in the previous section, Rayon splits tasks only till a specific depth is reached. This also implies that if the depth has been reached by all threads, but there is somehow an idle thread in the system, there would be no task splitting and hence that thread would remain idle forever.

In conclusion, the programmer would hence pay for an extremely high task creation overhead, while the load distribution would be more skewed towards the threads that started first.

This motivates the design and implementation of an Adaptive API developed in-house at INRIA.

3.1.1 The adaptive task splitting heuristic

In the adaptive API, the thread that starts the computation regularly listens on a channel for steal requests. Upon identifying one such request, it splits the remaining work into two halves and gives the latter half to the stealer. In case there are nested parallel iterators, the inner iterators will not create tasks until the outer ones have stopped splitting. This would ensure that the stolen tasks are coarse grained.

3.1.2 Contributions to the API

As a part of this thesis, some changes were made to the API interface. Previously, the usage of the API was slightly more tedious with the constraint of having to use some types exported by this API. This meant that the API would not actually provide parallel iterators, but would require the programmer to implement a split function on a predefined container. This would allow the internal scheduler of the API to split the work and create tasks. This interface was completely changed (in the context of this thesis) to support Rayon parallel iterators. After these changes, it is sufficient to create a Rayon parallel iterator and call some adaptive methods on it (while the Adaptive API is in scope). This would bypass Rayon's scheduler and follow the adaptive task splitting.

Chapter 4

A visualisation library for Rayon

4.1 Beyond benchmarking and profiling

Comparing performance is traditionally achieved by benchmarking various alternative implementations and then possibly profiling some of them to take a closer look at how performance can be improved. However, in the case of parallel iterators, this becomes more complicated. Benchmarking only gives a relative estimate of the performance, while profiling parallel iterators is not currently supported. This motivates the in-house development of a visualisation library (called Rayon Logs) that allows to create SVGs of parallel iterators wherein task creation and stealing can be seen. This has been indispensable to compare the Adaptive API with Rayon time and again. The following sections describe various features of this framework, many of which have been improved upon within the scope of this thesis.

4.2 Visualisation of a parallel algorithm

The Rayon Logs library illustrates each task that was created during the execution of the algorithm, as a box. The lenth of a box in the drawings is indicative of the amount of time spent in that task. These boxes are arranged in a hierarchial tree-like representation of the various tasks (a task placed below another one implies that the latter must end before the former starts). It also adds the relevant edges in case the outputs of multiple tasks have been merged together.

Each thread in the system is represented by a color, and the boxes are filled with the color corresponding to the thread that ran the task. Furthermore, the color of the box also depicts the speed at which the given task was executed. The speed of a task is defined as the size of input divided by the time taken. This speed is then normalised across all tasks of a given type, and a shade of the color is assigned to each box. A darker shade indicates a slow task. This allows for an instantaneous visual comparison of the speed of execution, which might be the limiting constraint in the speedup.

Each box furthermore, has an annotation that displays the:

- Exact time spent in the given task.
- Size of the input of the given task.
- The ID of the thread that executed the given task.
- The speed with which the task was executed. This is defined as the work divided by the time.

This display concludes with a group of boxes, wherein the length of each box indicates the amount of idle time spent by the given thread. This SVG as a whole is animated so that one can understand which tasks ran in parallel with each other. [ADD AN SVG HERE]

4.3 Visual comparison of algorithms

This library supports an experimentation API in which we can add several algorithms and compare their runs. The API exports a thread pool to which these algorithms can be added. Each algorithm is then run for a specified number of times. This hence generates a common histogram with a distribution of various run times for each algorithm. Furthermore, it also displays some statistics such as mean and median run times. It also offers the possibility of tagging particular parts of the algorithm for which the time spent will be measured separately.

Such a log terminates with visualisations of median runs and best runs of all the algorithms.

4.4 Changelog

During this thesis, the following improvements were made to the Rayon Logs library:

- The speed normalisation for the experimentation API was changed from per algorithm normalisation to normalisation across all algorithms. This required some major changes to the internal data-flow of loging.
- The experimentation API itself was modified to support any number of algorithms being compared. Earlier it could only compare two algorithms. This was achieved by reimplementing the thread pool (to which these algorithms were attached) with a builder pattern.
- Added computation and display for idle times in the experimentation API. The idle time has been defined as the difference of the total run time and sum of the time taken by all logged tasks.
- Added computation and display for median run times in the experimentation API. This would be useful since there is weak correlation between mean run time and mean idle time, however there is a stronger correlation between the total runtime and the idle time of the median run of each algorithm.
- Integrated the experiments API with the existing Hwloc-RS library for thread binding. The aforementioned thread pool can hence be bound to specific cores of a NUMA machine with two possible binding policies. One may choose to use all cores of a given NUMA node first before using the cores in another NUMA node, or may bind threads to cores in a round robin fashion across available NUMA nodes.
- Completely redesigned parallel iterator logging. Previously, the leaf level tasks were being displayed in the case of parallel iterators, with no hierarchial information. This did not permit illustration of nested parallel iterators. This overhaul of logging semantics now generates a hierarchial display of iterators being split and fused. It can also properly display the nested parallel iterators in this hierarchy.

4.5 Some examples of autogenerated logs

Chapter 5

Case study: Infix solvers

In order to better comprehend the Rayon API and the Adaptive API, a small case study of infix solvers was carried out. The following problem was solved using sequential and parallel iterators, logs were generated and performance was compared to understand the implications of using these abstract APIs.

5.1 The problem

There exists an expression of integers interleaved with the addition or multiplication operator. The objective is to evaluate this expression with the precedence of multiplication over addition and return the integral result.

5.2 The sequential algorithm

In order to solve this problem using iterators, it is necessary to do a stateful iteration over the input. The state represents the partial output for the subset of the input that has already been scanned. This can be represented by a tuple in which the first element represents the sum of all partial products encountered, and the second element represents the running partial product. Such a stateful iteration is implemented using the fold function over sequential iterators in Rust.

5.2.1 Code

```
fn sequential_solver(inp: &[Token], outp: &mut u64) {
    let ans = inp.iter().fold((0, 1), |tup, elem| match elem {
        Token::Num(i) => (tup.0, tup.1 * *i),
        Token::Mult => tup,
        Token::Add => (tup.0 + tup.1, 1),
    });
    *outp = ans.0 + ans.1
}
```

5.3 Parallel variants

Since the parallel reduction requires an associative binary operator, it is necessary to modify the state representation such that it can capture a partial result over *any* contiguous subset of the input, as opposed to a subset of the input taken invariably from the start. Furthermore, it should be possible to combine two such partial results into one. This intermediate result has hence been defined as a triplet of integers *(a, b, c)* where a is the product of all integers that were encountered till the first sum operand, b is the sum of all intermediate products except the last one and c is the last contiguous product of integers.

This representation is hence initialised as (0, 0, 1) and at the occurence of any integer, c is multiplied with that integer. Whenever + is encountered, if a is 0, we set $a = c$. Else, we set b += c. The occurence of * is requires no action.

Combination of two partial results *(a, b, c)* and *(d, e, f)* produces the partial result *(a, b+c*d+e, f)*.

5.3.1 Rayon Fold

The aforementioned stateful iteration can be done over a parallel iterator in the Rayon API. As opposed to a single final state in the sequential fold, Rayon's fold function produces several final states in parallel. This requires a parallel reduction that will be carried out using the classic reduction tree to produce one single result.

In the case of infix solvers, the fold shall produce several such partial results, all of which will be reduced in parallel into one single result which will be in the form *(a, b, c)*, however, it will represent the entire input. Hence, the integral result is trivially computed as $a+b+c$.

5.3.2 Rayon Split

Another strategy could be to exploit the *par_split()* function in Rayon to slice the input at all occurence) of the +. These slices invariably represent contiguous products of integers, which can be computed using a nested parallel iterator which carry out a parallel reduction using the binary associative multiply operator. Finally, another parallel reduction using the + operator gives the integral result that is expected from the input.

5.3.3 Adaptive Fold

The Adaptive API exports an identical interface to fold and reduce partial results, and hence this solution is syntactically similar to `Rayon Fold`

5.3.4 Code

```
pub fn solver_par_split(inp: &[Token]) -> u64 {
    inp.as_parallel_slice()
        .par_split(|tok| *tok == Token::Add)
        .map(|slice| {
            slice.into_par_iter()
                .filter_map(|tok| match tok {
                    Token::Mult | Token::Add => None,
                    Token::Num(i) => Some(i),
                })
                .product::<u64>()
        })
        .sum::<u64>()
}

pub fn solver_par_fold(inp: &[Token]) -> u64 {
    inp.into_par_iter()
        .fold(PartialProducts::new, |mut products, tok| match *tok {
            Token::Num(i) => {
                products.update_product(i);
                products
            }
            Token::Add => {
                products.append_product();
                products
            }
            Token::Mult => products,
        }).reduce(PartialProducts::new, |left, right| left.fuse(&right))
        .evaluate()
}

pub fn solver_adaptive(inp: &[Token], policy: Policy) -> u64 {
    inp.into_adapt_iter()
            .with_policy(policy)
            .fold(PartialProducts::new, |mut p, token| {
                match token {
                    Token::Num(i) => p.update_product(*i),
                    Token::Add => p.append_product(),
                    Token::Mult => {}
                }
                p
            }).reduce(|left, right| left.fuse(&right))
            .evaluate()
}
```

5.4 Speedup curves

5.5 Logs

5.6 Conclusions

The performance for `Rayon Split` was quite unstable. It depended completely on the density of the + operator. This is as expected since the + operator dictates task splitting in the first level, varying it's density would lead to too few or too many tasks and hence change the performance.

The `Rayon Fold` was only slightly faster than sequential version. As per the logs, this was mainly because of excessive task splitting which lead to a high task creation overhead and also slowed down the computation of each task. Furthermore, the cost of initialising the partial results and updating them didn't get amortized due to the high number of tasks that were created.

The logs reveal that the speedup for `Adaptive Fold` was sublinear mainly because of lower speed of execution of each parallel task. The idle times were quite low and task creation was much lower than Rayon. However, load seemed to have been balanced more or less effectively. The lower speed of computation can be attributed to the difference in the way Rust optimizes a sequential fold over a tuple versus operations over the PartialProducts struct. Furthermore, the overhead of reduction adds on to the constant factor in the overall linear work in the parallel algorithm.

Chapter 6

Graph based in-situ analytics

6.1 The need for in-situ analytics

Parallel simulations produce large amount of data. The traditional approach consists of writing this data to disk, reading it back from the disk and analyzing it. But this approach is extremely slow. An alternative would be to analyze the data online, as soon as it is provided by the simulation, before writing it to the disk. Thus the data is reduced in size before being written to the disk, which leads to a better performance.

6.2 The proposed algorithm

A variety of molecular (mechanical and biological) simulations, generate a large number of (from a few million to a few hundred million) points in a three dimensional space. The interactions between such molecules/points determines the position of the points generated in the next time-step of the solution. Hence the following (two dimensional) algorithm becomes a crucial step in the analytics on the data produced by these simulations:

1. Hash the set of points into squares in the 2D space using four different hash functions.
2. For each square in a given hash, make an undirected graph with an edge between two points *iff* the euclidean distance between them is less than a given threshold T. This is an empirical parameter of the algorithm.
3. Fuse all graphs hence formed.
    - For each point, unionize the four adjacency lists (formed in `Step 2`) corresponding to the four hash functions.

   - Concatenate all such adjacency lists (one for each point) into a single graph.

4. Output a set of connected components for this graph.

The aforementioned hash functions tile the entire 2D space into several squares, each of side $2 \times T$. For a given point (x, y), a hash function hence outputs the co-ordinates of the square in which this point lies.

Hence, one such hash function would map (x, y) to $(\frac{x}{2 \times T}, \frac{y}{2 \times T})$. The other three hash functions mentioned are obtained by shifting the origin of the 2D space to $(0, \frac{T}{2}), (\frac{T}{2}, 0)$, and $(\frac{T}{2}, \frac{T}{2})$ respectively.

The above design implies that for any two points that are at a distance of T or less, they will be hashed into the same square in atleast one of the four hashes. Therefore there would be $4 \times O(4n\frac{T}{R}^2)$ distance computations where R is the range of the 2D space.

6.3 Code

6.4 Profiling

6.5 Some optimizations