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
Dr. Frederic WAGNER
&
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
Dr. Bruno RAFFIN
Director of Reserch,
INRIA Grenoble Rhone-Alpes
Date:

Supervisor
Dr. Frederic WAGNER
Associate Professor,
ENSIMAG, Grenoble
Date:

Co-Supervisor
Prof. Sundar Shan BALASUBRAMANIAM
Professor,
BITS-Pilani Pilani Campus
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

The objective of this thesis is two-fold. The first and foremost facet is to provide a fast, scalable and safe solution for in-situ analytics for the data produced by large scale molecular simulations. This aligns closely with the overall research theme of the *Team DATAMOVE*. A simulation models a huge system of particles which evolves over time. Computationally, this evolution is traced over a discrete set of timesteps. While progressing from one timestep to another, it is essential to analyse some domain-specific interactions between particles and hence determine the configuration for the next timestep. In-situ analytics permit the compression of this loop by moving these analytics to the generation of the configuration and process the data as soon as it is generated. This speeds up the entire process by reducing the memory read and write cost.

The secondary objective is to switch over to a new language called Rust which is still in its infancy (having launched in May 2015). Given that the entire Rust ecosystem is open-source, it is quite feasible to modify/add on to the existing setup to suit the specific constraints posed by the above use case. However, given that the language is quite new, several features (for example the support for streams) are still in unstable builds and hence active participation in the community is also essential.

In summary, this thesis aptly titled as *Hijacking the Rust programming language for high performant in-situ analytics* not only presents novel research and development in high performance computing for big data, but also adds some indispensable features to the promising Rust framework to allow similar research threads to switch over to this futuristic language.

Acknowledgements

Quite literally none of the work presented here would have been feasible without the support and supervision of Prof. Frederic Wagner. He has singlehandedly developed the Adaptive and Logging APIs being used in this thesis. Several members of the *Datamove* team have also given helpful pointers all along. A special mention here goes to Pedro Buel who has time and again helped in using the R framework for plotting data, and Adrien Faure who has helped debug some poorly written Rust code. Finally, a special thanks to Dr. Bruno Raffin and Prof. Sundar Balasubramaniam for supervising and co-supervising the thesis.

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

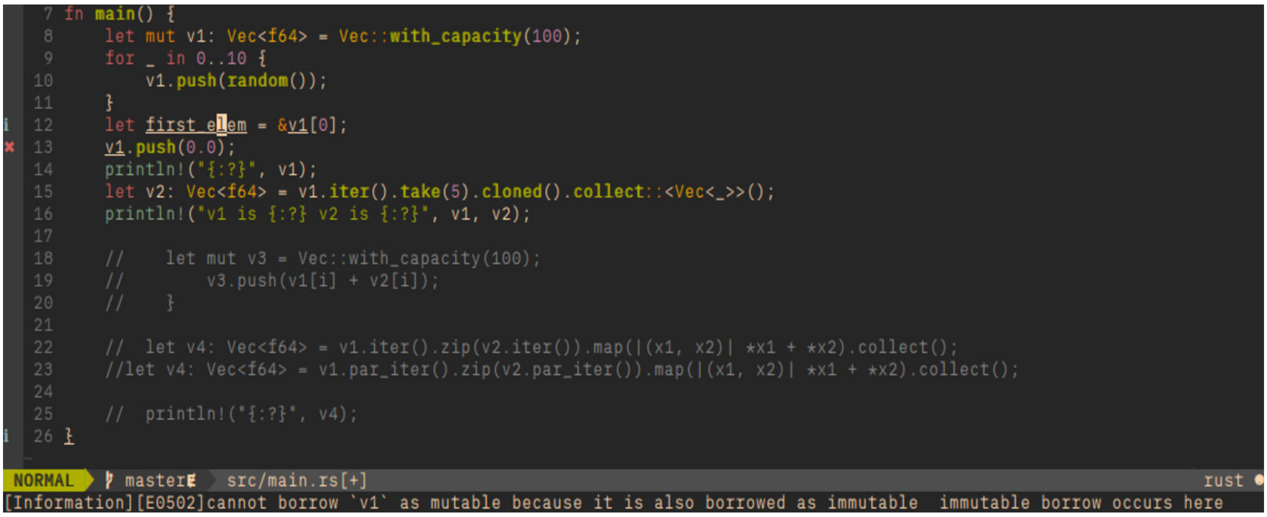
When variables are reassigned or passed around across functions, they adopt exactly one of two semantics, move and copy. Typically, any variable on the heap adopts the former while lightweight datatypes residing on the stack adopt the latter.

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



```
fn main() {
    let mut v1: Vec<f64> = Vec::with_capacity(100);
    for _ in 0..10 {
        v1.push(random());
    }
    let first_elem = &v1[0];
    v1.push(0.0);
    println!("{:?}", v1);
    let v2: Vec<f64> = v1.iter().take(5).cloned().collect::<Vec<_>>();
    println!("v1 is {:?} v2 is {:?}", v1, v2);

    //    let mut v3 = Vec::with_capacity(100);
    //        v3.push(v1[i] + v2[i]);
    //    }

    //  let v4: Vec<f64> = v1.iter().zip(v2.iter()).map(|(x1, x2)| *x1 + *x2).collect();
    //let v4: Vec<f64> = v1.par_iter().zip(v2.par_iter()).map(|(x1, x2)| *x1 + *x2).collect();

    //  println!("{:?}", v4);
}
```

```
[Information][E0502]cannot borrow `v1` as mutable because it is also borrowed as immutable  immutable borrow occurs here
```

FIGURE 1.1: The "borrow checker" in the compiler guarding against lost updates.

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. As shown in Figure 1.1, the compiler throws a compiler time error if an immutable reference and a mutable reference are held on the same object.

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.

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

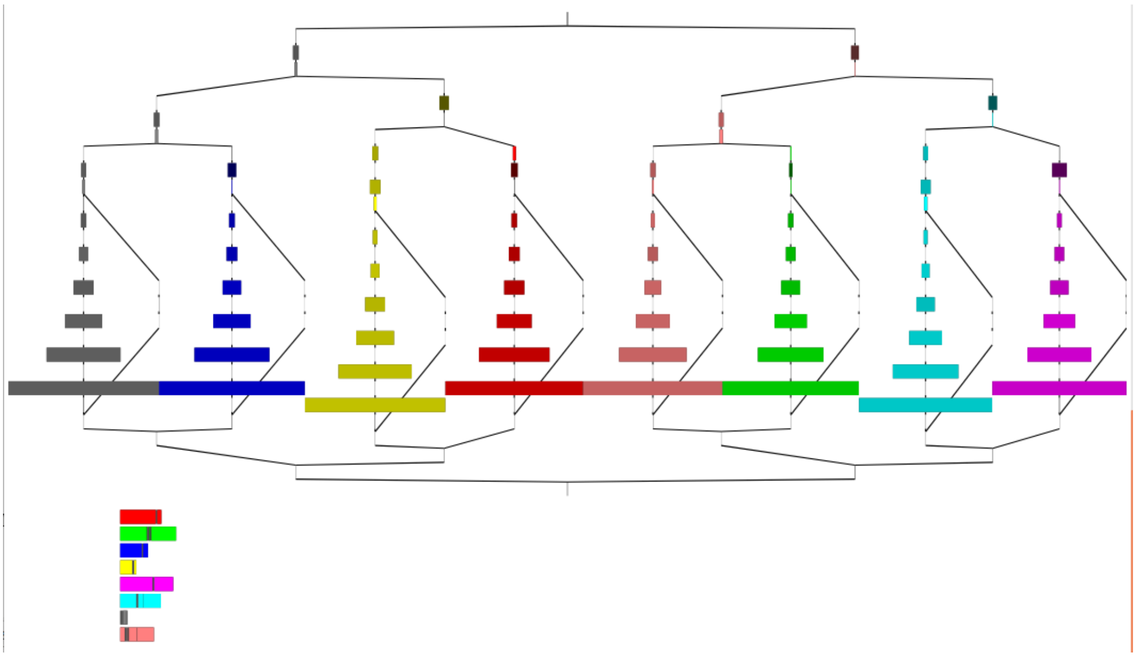


Figure 4.1: An image generated from the Adaptive API solving the infix problem

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. All this can be seen in Figure 4.1.

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

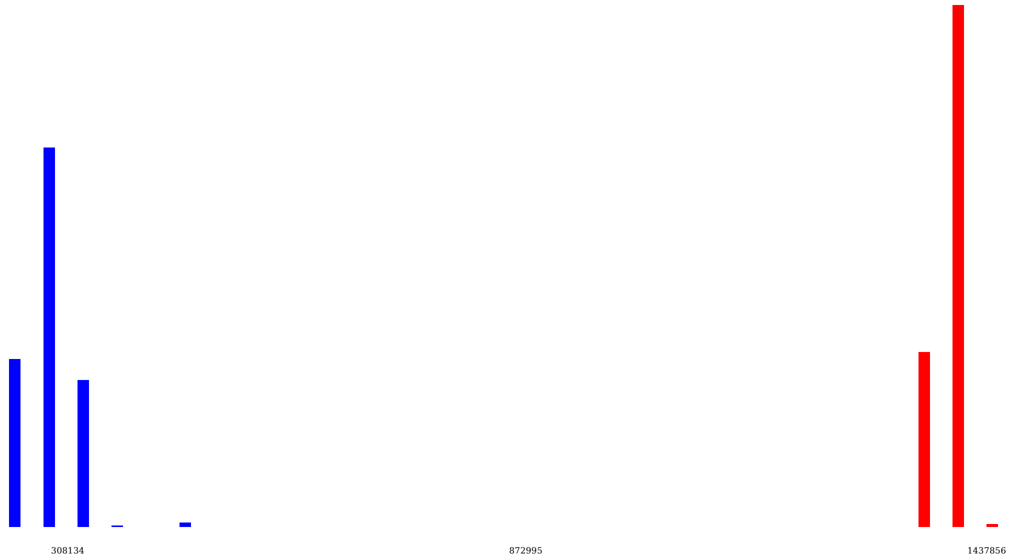
During this thesis, the following improvements were made to the Rayon Logs library:

- 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 A complete autogenerated log

Comparing sequential and adaptive

Distribution of execution times over 500 runs red is sequential, blue is adaptive,



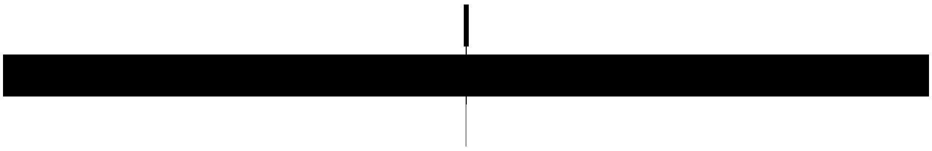
The Mean statistics are

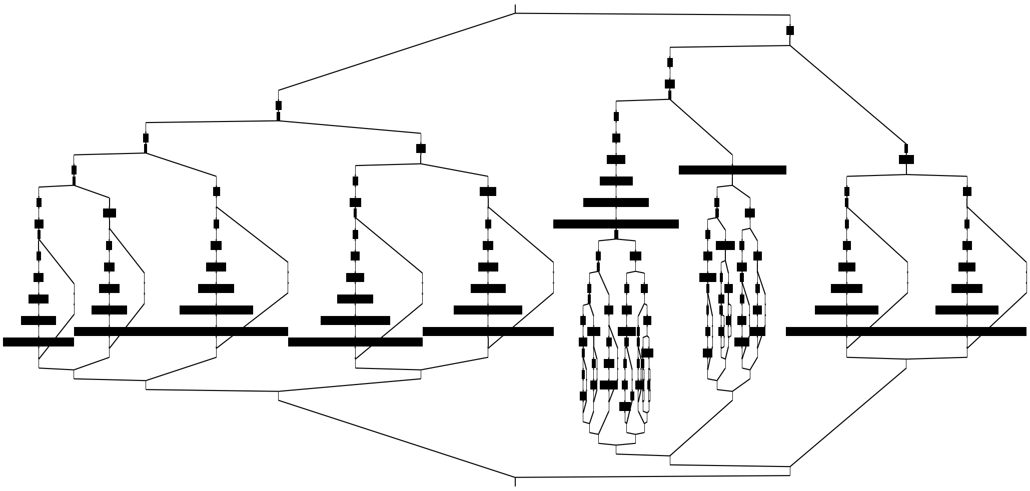
algorithm	net time	sequential times	idle time
sequential	1.3683196	0:1.363824024	9.578635834
adaptive	0.360837362	0:2.118832832, 1:0.24136067	0.4276589359999998

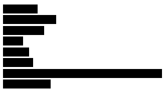
The Median statistics are

algorithm	net time	sequential times	idle time
sequential	1.367425	0:1.359933	9.572251999999999
adaptive	0.356592	0:2.086321, 1:0.272518	0.38593900000000003

Comparing median runs

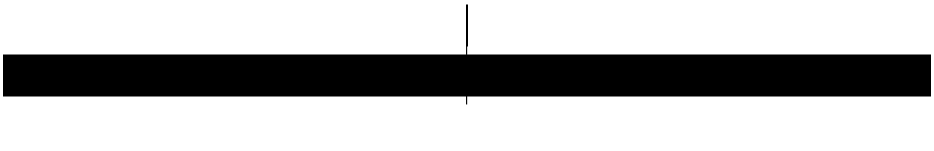


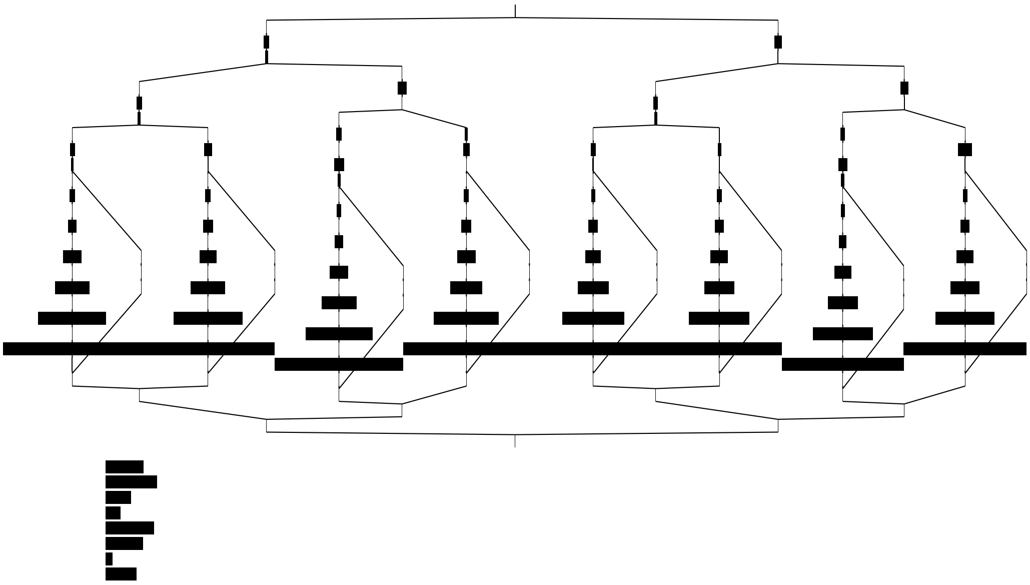




The average speeds for median run of sequential are 1
The average speeds for median run of adaptive are 0.657342460960381

Comparing best runs





The average speeds for best run of sequential are 1
The average speeds for best run of adaptive are 0.6415500076008999

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.

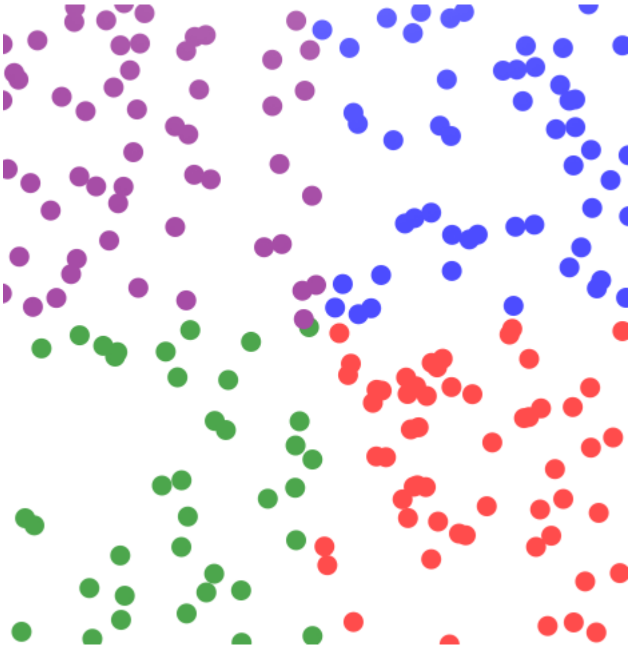


Figure 6.1: Hash squares for a random distribution of points.

- 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. The Figure 6.1 is indicative of the squares in which the points are hashed.

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, instead of the naive $O(n^2)$. This constant factor is quite meaningful because the threshold distance is up to 4 orders of magnitude less than the range.

6.3 Profiling

This sequential algorithm was profiled using valgrind and the key areas of optimization were identified to be `Step 2` and `Step 3`.

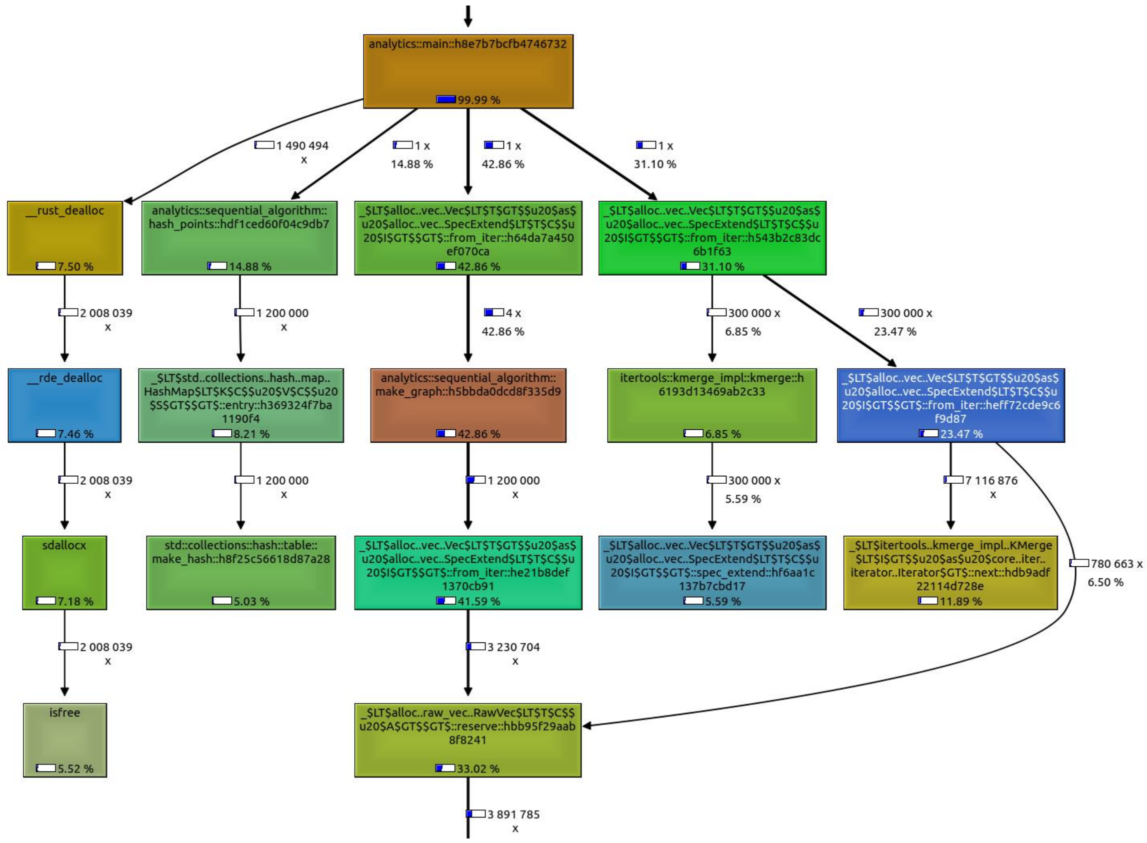

analytics::main::h8e7b7bcfb4746732
99.99 %
1 490 494 x
1 x 14.88 %
1 x 42.86 %
1 x 31.10 %
__rust_dealloc
7.50 %
analytics::sequential_algorithm::hash_points::hdf1ced60f04c9db7
14.88 %
_LTalloc..vec..VecLTT$GT$$u20asu20$alloc..vec..SpecExtend$LTTC$$u20IGT$$GT$::from_iter::h64da7a450ef070ca
42.86 %
_LTalloc..vec..VecLTT$GT$$u20asu20$alloc..vec..SpecExtend$LTTC$$u20IGT$$GT$::from_iter::h543b2c83dc6b1f63
31.10 %
2 008 039 x
1 200 000 x
4 x 42.86 %
300 000 x 6.85 %
300 000 x 23.47 %
__rde_dealloc
7.46 %
_LTstd..collections..hash..map..HashMapLTK$C$$u20VC$$u20SGT$$GT$::entry::h369324f7ba1190f4
8.21 %
analytics::sequential_algorithm::make_graph::h5bbda0dcd8f335d9
42.86 %
itertools::kmerge_impl::kmerge::h6193d13469ab2c33
6.85 %
_LTalloc..vec..VecLTT$GT$$u20asu20$alloc..vec..SpecExtend$LTTC$$u20IGT$$GT$::from_iter::heff72cde9c6f9d87
23.47 %
2 008 039 x
1 200 000 x
1 200 000 x
300 000 x 5.59 %
7 116 876 x
sdallocx
7.18 %
std::collections::hash::table::make_hash::h8f25c56618d87a28
5.03 %
_LTalloc..vec..VecLTT$GT$$u20asu20$alloc..vec..SpecExtend$LTTC$$u20IGT$$GT$::from_iter::he21b8def1370cb91
41.59 %
_LTalloc..vec..VecLTT$GT$$u20asu20$alloc..vec..SpecExtend$LTTC$$u20IGT$$GT$::spec_extend::hf6aa1c137b7cbd17
5.59 %
_LTitertools..kmerge_impl..KMergeLTI$GT$$u20asu20$core..iter..iterator..IteratorGT::next::hdb9adf22114d728e
11.89 %
780 663 x 6.50 %
2 008 039 x
3 230 704 x
isfree
5.52 %
_LTalloc..raw_vec..RawVecLTT$C$$u20AGT$$GT$::reserve::hbb95f29aab8f8241
33.02 %
3 891 785 x

FIGURE 6.2: Call graph generated by kcachegrind.

6.4 Some optimizations

The following sections detail the sequential optimizations that were carried out to improve the run-time of the algorithm.

6.4.1 Optimizing graph fusion

It was evident from the profile that a function `kmerge()` was taking up more than 16% time, and hence the contribution of `Step 3` in the overall run time was unusually high. This function was being used (from an external library) to merge two sorted iterators. However, it had been written to be used for any number of sorted iterators, and hence the implementation internally used a heap to order these iterators. While this would have been a nice strategy for a significantly large number of iterators, in the case of two iterators, it led to an unnecessary overhead.

Therefore, this was replaced with a handwritten function to fuse two sorted iterators into one.

6.4.2 Optimizing graph creation

A rather complicated optimization was carried out to reduce the number of distance computations within a hash square as well (`Step 2`). The above design relied on the threshold in order to reduce the number of distance computations, however with the following geometric optimization, the correlation between the threshold and the performance was significantly reduced.

The objective of this optimization is to build a smaller set of points out of all the points in a square, following which, distance computation will be done only for the points in the smaller set. This is done in the following manner:

1. Rehash the points in a square by tiling it with even smaller squares, such that no two points in the same inner square are more than T distance apart. Hence the side of the inner square must be $\frac{T}{\sqrt{2}}$.
   - This implies that all the points in an inner square form a clique of the final graph.
   - However, there might exist edges between two points that are not in the same clique.
   - Hence, distance computation must still be done between all points of the first clique and all the points of the second clique for all unordered pairs of cliques that can be formed.
   - However, if there exists a subset of *relevant points* in a clique, such that for any point outside the clique, the point within the clique that is closest to it must be a part of this subset of *relevant points*, then distance computation is not required for any point that is not in this subset.
2. The problem hence reduces to determining that for a given square with a lot of points inside it, which are the points that can never be the ones closest to a point outside this square.
3. This can be further simplified to compute a set of *relevant points* for each side of the square and then take an union of all such sets of *relevant points*. The Figure 6.3 illustrates the set of *relevant points* in green for a given outer square. The points in red will not be a part of the relevant points set, but they form a clique along with the points that surround them (in green).
4. Hence for any two sets of relevant points $R1, R2$ obtained from two different inner squares (within the same outer square), distance computation is only done between

The following section describes how to compute the set of relevant points.

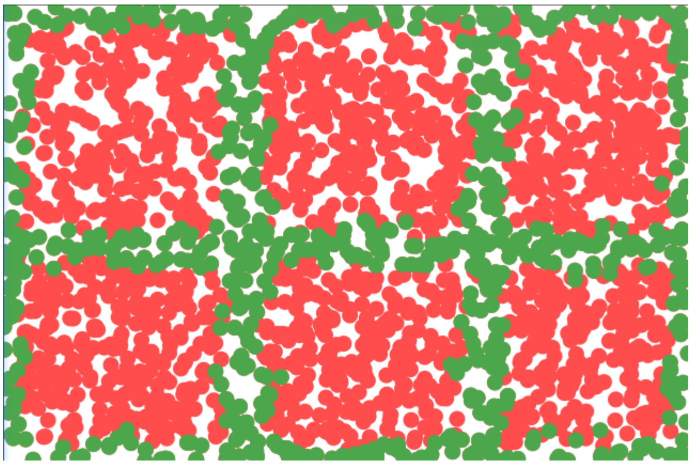


Figure 6.3: Comuting relevant points

6.4.3 Computing the relevant points

The figure 6.4 demonstrates the rationale behind the computation of the relevant points.

1. For a given point A, draw two lines with angles 45° and 135°.
2. Conjecture: For any point B below these lines, there can never exist any point in the area S (on the left side of the vertical line passing through B) such that the distance between that point and B is less than the distance between that point and A.
    - Consider the point P which is on the boundary of S.

    $$d(P,C) = d(P,F) + d(F,C) \tag{6.1}$$

    and since slope of AC is -1,

    $$d(A,F) = d(F,C) \tag{6.2}$$

    and in triangle APF,

    $$d(P,A) < d(F,A) + d(P,F) \tag{6.3}$$

    therefore

    $$d(P,A) < d(P,C) < d(P,B) \tag{6.4}$$

    - It is easy to see that this holds if we move point P anywhere in the area S.
3. The same argument is made for any point in the area S' using points E and D.
4. The conclusion in such a scenario is that point B can never be in the set of *relevant points* for the line l given that points A and E are relevant.

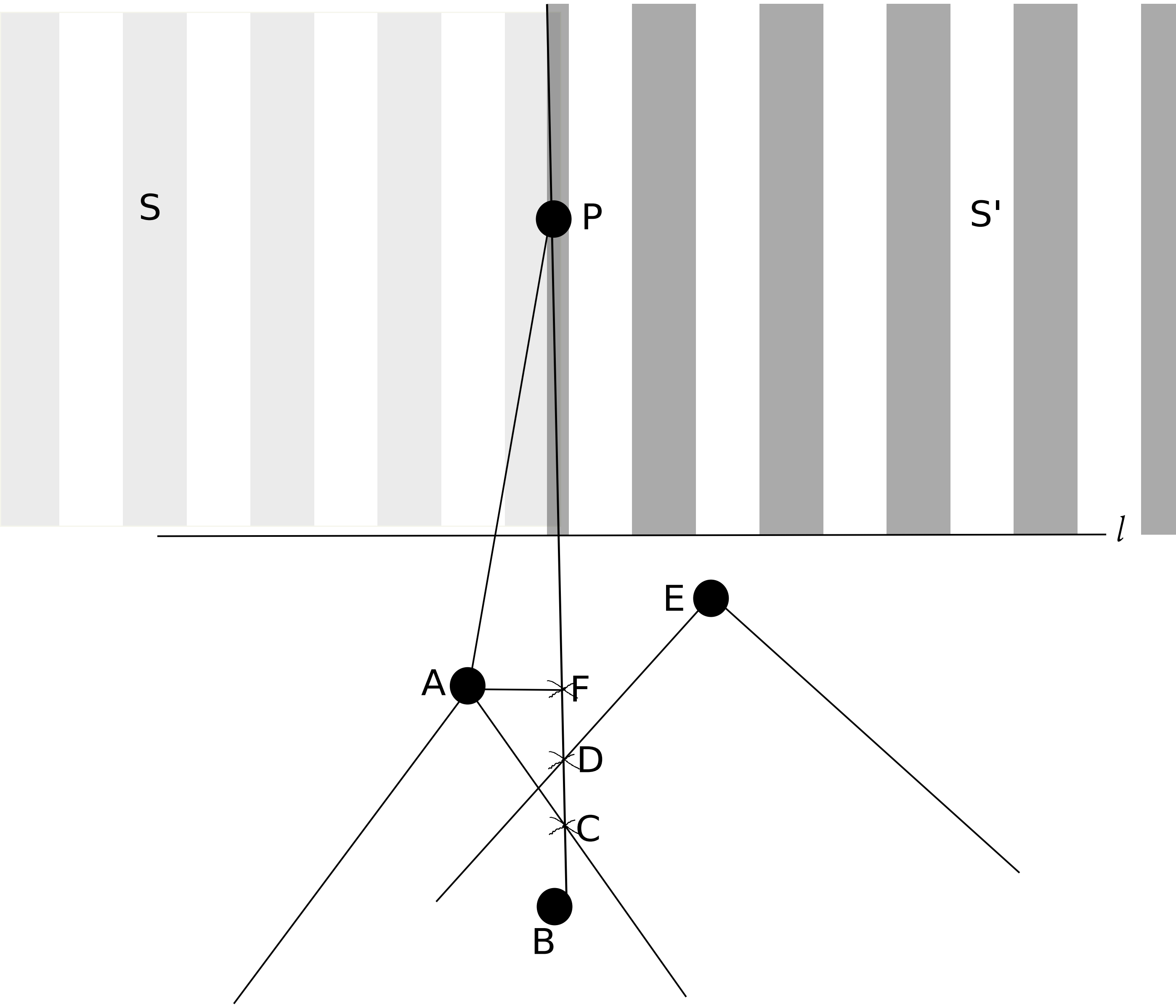


Figure 6.4: The rationale behind the optimization.

The algorithm to find the *relevant points* hence proceeds in a similar fashion by evaluating for each side of the square, which points are relevant and then taking the union of all such points. However, this optimization itself costs $O(n \log n)$ where n is the number of points in the smaller square. Furthermore, the number of points it ends up reducing is also completely dependent on the density of the inner square. Hence, some benchmarking was carried out to find out at approximately how many points per square should this optimization be performed.

The Figure 6.5 shows the dependence of the run-time on the threshold parameter in the final hybrid algorithm. This was programmed to switch on the optimization once the number of points crosses 500 in any given square.

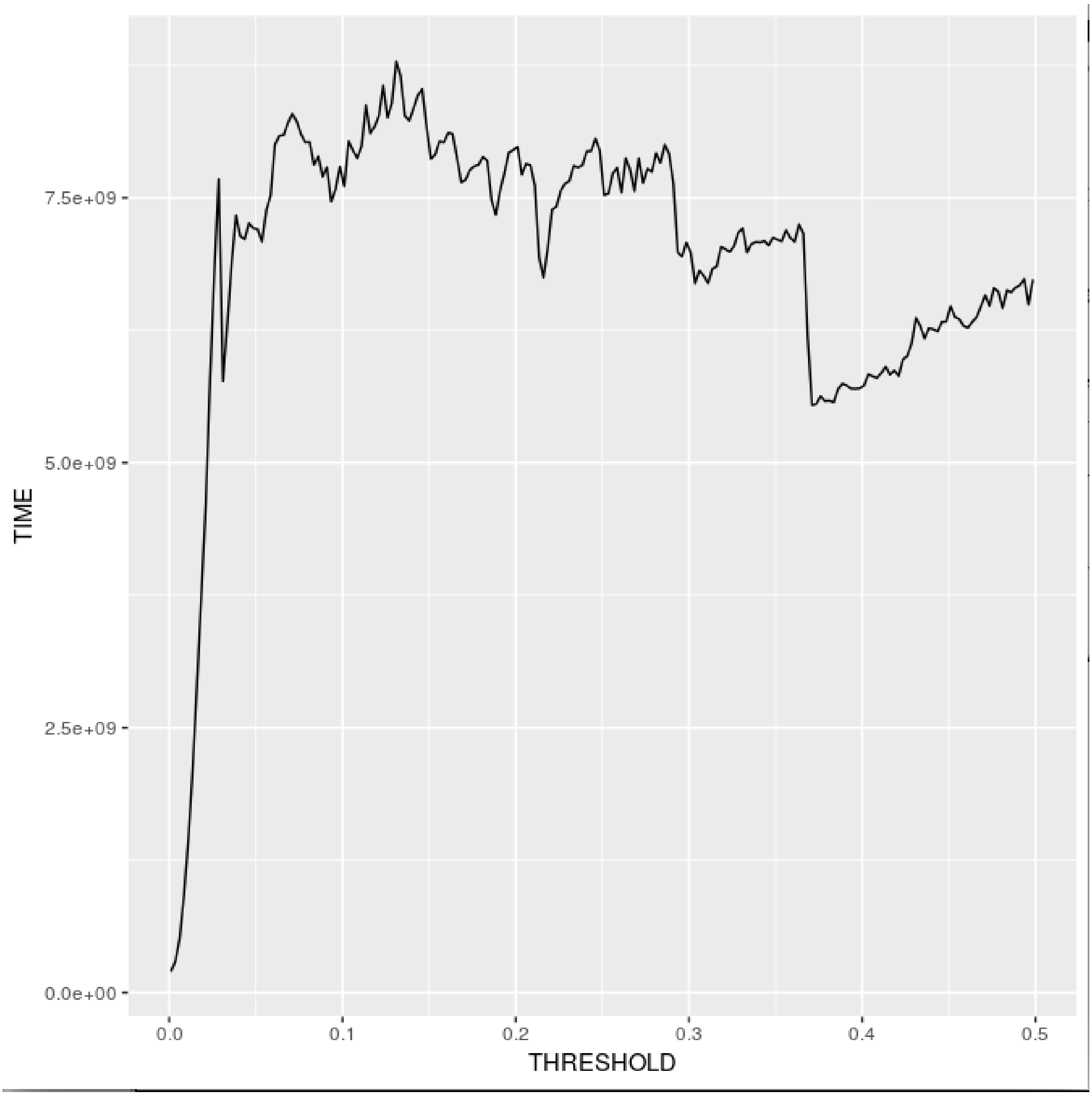


Figure 6.5: Time taken (in nanoseconds) versus Threshold (Range of the 2D space is (0,1))

Chapter 7

Parallel in-situ analytics

As per the profiling done (figure 6.2), it is clear that the function that can offer the singlemost speedup is the one that makes the graph from the hash squares. Hence, it was decided to use parallel iterators to implement this in a multithreaded fashion. The following section describe the attempts to parallelize this using Rayon and using the Adaptive API.

7.1 Rayon parallelization

This algorithm has the following three levels of parallelization:

1. Parallelization over the 4 hash functions.
2. Parallelization over the outer hash squares.
3. Parallelization over the outer loop of the $O(n^2)$ distance computations where n is the number of points in a given set of *relevant points*.

After the 3^{rd} parallelization, points are parallely written to the proper rows of the adjacency lists if the distance is less than T. The 2^{nd} level of parallelization uses a parallel fold to finally output the adjacency list for a given hash function. The intermediate state of this parallel fold has to be a collection of adjacency lists. Therefore, this could either be a `Vec<Vec<Point>>` or a `LinkedList<Vec<Point>>`. This leads to two different ways of aggregating the results.

- In case `Vec<Vec<Point>>` is used, the result of the parallel fold is several such vectors. They need to be concatenated with each other to finally give a single `Vec<Vec<Point>>`. This can be done in parallel by:

    1. Wrapping around each `Vec<Vec<Point>>` in a `LinkedList`.

2. Next, a parallel reduction can be carried out in which two `LinkedLists` formed above can be fused in constant time.
3. This gives a single `LinkedList` which contains several `Vec<Vec<Point>>s`, which can be converted into a sequential iterator.
4. A sequential fold is then performed over this sequential iterator. The state of this fold is now a `Vec<Vec<Point>>`, which is hence what the final result would be.

- In case `LinkedList<Vec<Point>>` is used as the fold state, the parallel reduction into a single `Vec<Vec<Point>>` can be done by:
    1. Carrying out the same reduction as in `Step 2` above, but for a parallel iterator over `LinkedList<Vec<Point>>`. This gives a single `LinkedList<Vec<Point>>` as the result.
    2. Converting this `LinkedList<Vec<Point>>` into a sequential iterator over `Vec<Point>` and then `collect()`-ing the iterator into a single `Vec<Vec<Point>>`.

A better performance is expected in the first case because the `LinkedList` formed in `Step 3` contains `Vec<Vec<Point>>`, which are fewer in number as compared to the number of `Vec<Point>` that the `LinkedList<Vec<Point>>` contains in the second method. Hence the last step of a sequential iterator would consume less time in the first method and so the algorithm remains more parallel.

The former algorithm is named *"Rayon Parallel"* and the latter is (optimistically!) named *"Rayon Parallel Opt"* in the speedup curve 7.1.

7.2 Parallelization using Adaptive API

Since the interface exported by the Adaptive API is identical to Rayon, the only difference here is to import the Adaptive API instead of Rayon. The approach and the implementation remains the same.

The implementation for this entire system has been open-sourced `here`.

7.3 Speedup curves

The speedups were obtained as shown in Figure 7.1. Here, it is important to note that the Adaptive API is undergoing some changes, hence the *"Adaptive Parallel Opt"* algorithm actually uses the Rayon API for the 3^{rd} level of parallelization. Therefore, the speedups are quite similar for the algorithms.

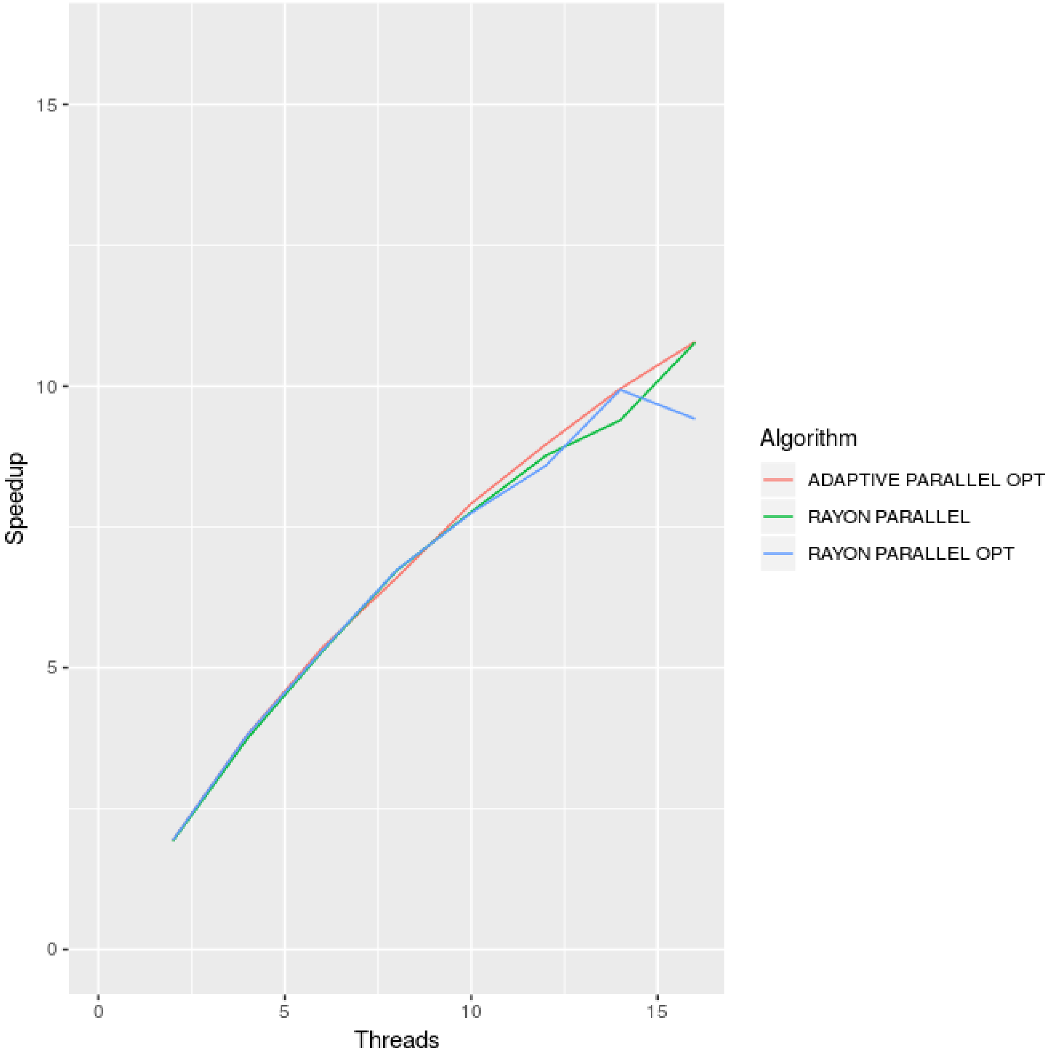


FIGURE 7.1: Speedup curves for the various parallel algorithms

Chapter 8

Future work

This research shall continue for two more months during which the in-situ analytics implementation shall be extended to support streams. This is a very new feature in the Rust language and is not stable right now. The idea behind this is to extend the current functional implementation such that given the availability of on-line data in streams, the resultant system shall be able to carry out the processing as soon as the data is generated. It is expected to perform better due to the parallelism that the conceptual pipeline shall offer. Furthermore, parallelization for graph fusion shall also be explored and an overall speed comparison shall be made for the sequential offline versus parallel offline versus the streaming algorithm.