High Performance Rust

Jim Walker

August 15, 2019

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2019

Abstract

This dissertation examines the suitabilty of the Rust programming language, to High Performance Computing (HPC). This examination is made through porting three HPC mini apps to Rust from typical HPC languages and comparing the perfomance of the Rust and the original implementation. We also investigate the readability of Rust's higher level programming syntax for HPC programmers through the use of a questionnaire.

Contents

List of Tables

List of Figures

Listings

Acknowledgements

This template is a slightly modified version of the one developed by Prof. Charles Duncan for MSc students in the Dept. of Meteorology. His acknowledgement follows:

This template has been produced with help from many former students who have shown different ways of doing things. Please make suggestions for further improvements.

Chapter 1

Introduction

In the field of high performance computing, it is difficult to say what is the most popular programming language. Firstly, we must define what we mean by popularity. Do we mean how many CPU hours are spent running programs from a particular language? Or do we mean the languague in which most of the development of new high performance programs is occuring? Or even, do we mean which programming language is most well liked by HPC programmers? The Rust programming language promises 'High-level ergonomics and low-level control' to help 'you write faster, more reliable software' [17].

I think it might be easier to write this section once I know what isn't in it.

Write about motiviations for questionnaire here.

Chapter 2

Background

2.1 Parallel Programming Languages for HPC

HPC (High Perfomrance Computing) refers to computation performed on supercomputers. Supercomputers generally have more and faster cores than personal computers. They are normally networked together with fast interconnect to allow for high data throughput, and are used for highly numerical scientific programs. To fully leverage the potential of these supercomputers many computing cores, programmers use parallel computing techniques, in programming languages which run as fast as possible on the hardware.

The three main languages used in HPC are Fortran, C and C++. They are all well established within the field, as shown by table 2.1, which shows the proportion of compute time taken up by these languages on Archer, (Advanced Research Computing High End Resource), one of the UK's primary academic research supercomputers. Whilst Fortran takes up the majority of compute time on Archer, this dissertation will focus on C and C++, as they are more comparable to Rust (their similarities are discussed further in the next sections).

	ARCHER
Fortran	69.3%
C++	7.4%
C	6.3%
Unidentified	19.4%

Table 2.1: Breakdown of CPU usage by programming language [45]

There are two central paradigms to parallel computing, message passing parallelism and share memory parallelism. In message passing parallelism, processes work on private data, and share data by sending and receiving messages. This form of parallelism is very scalable, and can run on geographically distributed heteregenous nodes [14]. Examples

of message passing parallelism include the MPI (message passing interface) standard, and Go's channels.

Shared memory parallelism, by comparison, has processes that share access to a region of memory. Whilst programs using can run on multiple nodes through technologies like PGAS (Partitioned Globabl Address Space), these nodes are still normally required to be homogenous. Shared memory parallelism is most effective when it runs on a single node with many processing cores. Although this paradigm has recently grown in usage through the usage of many core architures on GPUs, this dissertation will concern itself with shared memory parallelism using OpenMP, due to Rust's interesting shared memory model, which is discussed in further detail in section 2.3.

does this count?

2.2 C and C++

The C programming language was developed in 1972, as a 'system implementation language' [28]. Its first purpose was to program the UNIX operating systems and the utilities which were fundamental to its use, like `cat` and `rm`. Since that point, the C programming language has always been associated with low level computing. In this case, low level computing means computing which is able to be compiled to very efficient machine code, and gives the programmer fine grained memory management.

Today, the Linux kernel, which provides the foundation for the operating systems used on the vast majority of the world's supercomputers, is 96% written in C [44]. Many of the programs that are run on these supercomputers are written in C [12, 11, 26]

Despite C's success, only seven years after it was first developed, Bjarne Stroustrup began working on an extended version of C, which was to become C++. In 1985, the first commercial edition of C++ was released [34]. Two of C++'s most notable extensions to C are the introduction of classes, to allow for object oriented programming, and templates, which allow for generic programming. C++ also uses a stronger type system than C, which prevents bugs caused by implicit conversion.

Like C, the design of C++ focused on system programming [35], and like C, it has become a common language of choice for developing HPC codes [26], a fact which is helped by the close similarity of the two languages. Many C programs are valid C++ programs. C and C++ are also considered two of the languages which, when compiled, run fastest.

more examples here

The speed of C and C++ is one of their most celebrated design features. However, there are other, less positive consequences of the design of these two languages, which require programmers to use them with care. This dissertation is principally concerned with the memory safety issues of C and C++, which can cause programs to crash, or return incorrect data.

Listing 2.1 demonstrates one of the C and C++'s memory pitfalls, known as use after free. The code is valid C and C++. Use after free occurs when a program attempts to

use a section of memory after it has been released back to the operating system. The freeing of `array` here means that the contents of it cannot be guaranteed when it is printed.

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    int* array = (int*) malloc(sizeof(int)*10);
    for (int i=0; i<10; i++){
        array[i] = i;
    }
    free(array);
    printf("%d\n", array[1]);
    return 1;
}
```

Listing 2.1: C and C++: Use after free

In larger programs, this can lead to calculations being made using incorrect data, which has been overwritten by the operating system, or another thread from the same program. Other common sequential memory pitfalls in C and C++ are:

- **Double Free:** Attempting to free memory which has already been freed can lead to undefined behaviour.
- **Heap Exhaustion:** The program tries to allocate more memory than the amount available. This can be the result of a memory leak, when data is not always freed after being allocated.
- **Buffer Overrun:** Attempting to access the n^{th} element of an array which is only of length n. This can lead to the reading incorrect data, or accidentally trashing other memory within the same program.
- **Data Race:** This type of non-deterministic bug occurs when two or more threads need to update a variable, but the outcome of this update depends on the timing of the threads accessing the variable. An example of a data race is given in listing 2.2

Whilst it is possible to write memory safe code with memory unsafe languages, it is hard to do so. It is impossible to know how exactly many bugs exist in HPC codes,and to know how many of those are caused by memory safety issues. As an indication, we can take data from Microsoft, which shows that 70% of their Common Vulnerabilities and Exposures (CVEs) are caused by memory safety issues [25]. There is no good estimate of the amount of memory safety errors that exist in C or C++ HPC programs, but if we take the Microsoft data to be indicative of general error sources, then the lack of memory safety in C and C++ should be a cause of concern for HPC programmers.

2.2.1 OpenMP

The first specification for the OpenMP (Open Multi-Processing) Fortran, C and C++ application program interface (API) was released in October 1998. Its aim was to ‘allow

users to create and manage parallel programs while permitting portability' [3]. It acts as an extension to the C and C++ language specifications, leaving responsibility for implementing it to compiler writers, just as with C and C++. New specifications of OpenMP are periodically released, and it is now recognised as a cornerstone of HPC, as can be seen from the large number of people who sit on the its Architecture Review Board [2].

OpenMP's parallelism model is based around shared memory parallelism. This is done to reflect the reality of the multi-core hardware which are used in HPC. Multi-core processors share memory with each other, and each core can access any memory address on that node.

An example OpenMP program is shown in listing 2.2. It is valid C and C++. A key feature of OpenMP are its `#pragma omp` statements, which issue parallelism related directives to the compiler. One of OpenMP's core strengths is its succinct abstractions to the underlying threading API, irrespective of the platform it's running on. Here, there `#pragma omp` statement signifies the part of the code to be parallelised, and importantly, does not do so at the cost of obscuring the programs serial intent. The example parallelises the for loop, and sets the number of threads through the OMP_NUM_THREADS environment variable. In this program, the variable `a` is set to zero, and it is then incremented in the for loop.

```c
#include <omp.h>
#include <stdio.h>
int main(){
    int a = 0;
    #pragma omp parallel for
    for (int i=0; i<10; i++){
        a++;
    }
    printf("%d\n", a);
    return 0;
}
```

Listing 2.2: C and C++: OpenMP Data Race

However, the output of this program in non-deterministic, as it includes a data race condition. If the main thread completes first, it will print the value of `a` that currently exists, not wait until all the other threads have completed.

This race condition leads to different values being printed by the program on different executions. To solve this problem, a `#pragma omp barrier` statement must be inserted at the end of the for loop. As with the other memory errors mentioned earlier however, these mistakes are not always found before using a program.

2.3 Rust

The Rust programming language was started life as a side project by an employee of the Mozilla foundation, before becoming adopted and launched by it in 2011 [10]. Rust's

design was stated to be an 'Unapologetic interest in the static, structured, concurrent, large-systems language niche' [15]. Like C and C++, Rust's early design aims included a the goal of becoming a systems language. Rust like C has structs, and shares behaviours between structs through composition with traits, but not with inheritence, like C++.

Rust's initial design ideas mainly diverge from C and C++ in its aims to provide the programmer with memory safety. Two ideas, immutability and ownership are used to achieve this improvement.

Ownership is one of Rust's best known features. It 'allows Rust to be completely memory-safe' [41], and works by using the compiler's borrow checker to ensure that Rust's ownership model is satisfied by a given program before compiling it.

In listing 2.3 we present a Rust program that does not satisfy the ownership model, and therefore does not compile. The first line of the main function heap allocates memory to a vector of 10 elements, and gives each element a value of four. This vector is labelled `vector`. It is created using a macro, which is similar to functions in Rust, except that they can take a variable number of arguments, formatted in different ways, like with semi-colons.

```
fn main(){
    let vector = vec![4;10];
    drop(vector);
    println!("{}", vector[2]);
}
```

Listing 2.3: Rust: Use after free

The `drop()` function is then called on the vector, which is similar to `free()`. `drop()` is automatically called on values when they go out of scope. It is more accurate to think of `drop()` as something akin to C++'s destructors, but both those and this function do, at their core, release memory back to the operating system. However, attempting to use a variable after it has been dropped is illegal in Rust, resulting in the error message below:

```
error[E0382]: borrow of moved value: `array`
 --> src/main.rs:6:20
  |
2 |     let vector = vec![4;10];
  |         ----- move occurs because `array` has type `std::vec::Vec<i32>`,
  which does not implement the `Copy` trait
3 |     drop(vector);
  |          ----- value moved here
...
6 |     println!("", vector[2]);
  |                  ^^^^^ value borrowed here after move

error: aborting due to previous error
```

This is rust's borrow checker complaining that the program does not follow the ownership model. When the value of `vector` is dropped, in the Rust ownership model, the ownership of `vector` is moved into the drop function, and is not returned. When the

program later tries to use (borrow) the variable, it is therefore unable to, as Rust only allows for values to have one owner at a time.

Allowing values to only have one owner at a time is worked around by functions borrowing mutable or immutable references to those variables. For example, if a function needs to mutate a vector, it will need to specify that type in it's function arguments, i.e. `v: &mut Vec<i32>`, v where v is of the type of a borrowed mutable reference to a vector containing 32 bit integers. This requirement also highlights how Rust's borrow checker is reinforced by a strong type system, which requires the function parameter to be of a certain type, and immutability by default, which makes it explicit which functions will change values that are passed to them.

Memory safety is further improved in Rust by the absence of null pointers through the use of optional values. If a function may return something or nothing, it returns an `Option<T>`, which can either be `Some(v)` where `v` is a value of type `T`, or `None`. Pattern matching can be used to succinctly unwrap these variables.

Rust also makes the promise that it is free of data races, with certain caveats. Data races are defined as:

- 'two or more threads concurrently accessing a location of memory
- one of them is a write
- one of them is unsynchronized'

— The Rustonomicon: Data Races and Race Conditions [39]

and are only absent from safe rust. This does not mean that Rust prevents programmers from creating deadlock situations entirely, only that a certain subsection of data races are prevented, and only in safe Rust. Unsafe Rust exists as another language within Rust, delimited within `unsafe` blocks. It exists because there a limits to such a safe Rust which do not accurately reflect the underlying hardware on which it runs. However, it is not seen as being the '*true* Rust Programming Language' [40], and therefore this dissertation will only examine safe Rust. I will also attempt to write Rust in an idiomatic style in attempt to write Rust which is as representative of Rust as possible. Idiomatic Rust tends to chain function calls and use pattern matching to achieve more succinct code.

Unlike C and C++, Rust comes with a build tool and dependency manager, Cargo, which wraps around the Rust compiler. Cargo allows users to specify a programs dependencies, which are automatically downloaded and integrated into that program from external repositories. In Rust, these dependencies are called crates. In this dissertation, I make extensive use of the Rayon crate, as explained in section 2.3.1.

Some work has been done to investigate the applicability of Rust to HPC , but it expertise in the language is still low within the community . As such, I will gain technical support from the Rust community through the official subreddit and community discord channels when I encounter a problem particular to Rust.

citations here

cite maybe?

2.3.1 Rayon

Rayon is the one of the most popular crates for parallelism in Rust, and features heavily in the Rust cookbook [9]. In a fashion similar to OpenMP, it abstracts complicated underlying threading technologies. Unlike OpenMP, Rayon concentrates on parallel iterators, and like Rust promises data race freedom.

Rayon's parallel iterators are conceptually descended from Rust's sequential iterators. An iterator is a function which provides access to the elements of a collection, so that an operation can be performed on a set of those elements. In Rust, iterators implement the Iterator trait, which provides access to the current item, and a `next()` function, which returns an optional value. An example of a sequential iterator is shown in listing 2.4, where a vector of 5 elements, each with value 2, are first each multiplied by five in the `map()` function, using an anonymous function. In Rust, these are called closures. The `fold()` function then returns the product of all the elements in the mapped collection, which is 10000. The first argument of fold provides the identity value for the fold, which is used to begin the operation.

Rayon's parallel iterators work in a similar way to Rust's sequential iterators, except that they give sections of the iterable collection to separate threads to calculate. The parallel iterator methods also have slightly different syntax, as demonstrated by listing 2.5. This listing produces the same result as 2.4, but the `fold()` is different.

```
fn main(){
  let v = vec![2;5];
  let s = v.iter()
           .map(|x| x * 5)
           .fold(1, |acc, x| acc * x);
  println!("{}", s);
}
```

Listing 2.4: Rust: sequential iterator

```
extern crate rayon;
use rayon::prelude::*;

fn main(){
  let v = vec![2;5];
  let s = v.par_iter()
           .fold(|| 5, |acc, x| acc * x)
           .reduce(|| 1, |x, y| x * y);
  println!("{}", s);
}
```

Listing 2.5: Rayon: parallel iterator

The first argument to the parallel `fold()` is an identity closure, which generates the identity value. This is done so that the different threads can have their own copy of the identity value. The output of the fold is different too, as each thread performs a fold on its section, and therefore does not return a single value. A parallel reduction is called on the resulting collection, which delivers the product of all the values left in the collection. In this way, the fold reduce pattern is 'roughly equivalent to map reduce' [32] in effect. It is also noteworthy that `par_iter()` uses a number of threads set by the environment variable RAYON_NUM_THREADS, which is similar to OpenMP.

Iterators are safer than for loops. They prevent threads trying to access data beyond array boundaries, without a performance cost. However, they do lack of flexibility compared to for loops. From within a for loop, the programmer can access the i^{th} element of the collection they are iterating over, or the $i-1^{th}$ element, if they choose to through simple index arithmetic. I will investigate the costs and the benefit of this trade

off in my dissertation.

2.4 Kernels

By Kernels I mean blah blah. I will use Kernels in a similar way to how Mini-apps have been used in research in the past.

Mini-apps are a well established method of assessing new programming languages or techniques within HPC [23, 30, 24]. A mini-app is a small program which reproduces some functionality of a common HPC use case. Often, the program will be implemented using one particular technology, and then ported to another technology. The performance of the two mini-apps will then be tested, to see which technology is better suited to the particular problem represented by that mini-app. Such an approach gives quantitative data which provides a strong indication for the performance of a technology in a full implementation of an application. I am going to use Kernels rather than mini-apps because more breadth and less time, more use cases, better indication

This dissertation will follow a similar approach of evaluating a program through the performance of a kernel, using the test data to find any weaknesses in the Rust or original implementation.

I will also evaluate the ease with which I am able to port a kernel into Rust. These observations will provide insight into what it is like to program in Rust, if its strict memory model and functional idioms help or hinder translation from the imperative languages which the ported programs are written in. This qualitative, partly experiential information will hopefully provide an insight into the actual practicalities of programming in Rust. For Rust to be fully accepted by the HPC community, it is necessary that the program fulfils the functional requirements of speed and scaling, alongside non functional requirements, of usability and user experience. The first factor provides a reason for using Rust programs in HPC, the second provides an impetus for learning how to write those programs

Why use reference implementations and not write my own?

2.5 Roofline

stuff makes sense here.

Chapter 3

Methodology

3.1 Kernel Selection

So that a breadth of usage scenarios were examined, three kernels were selected based on their conformity to the following set of criteria.

- **The part of the program responsible for more than two thirds of the processing time should not be more than 1500 lines.** To ensure that I fully implemented three ports of existing kernels, it was necessary to limit the size of the kernels that could be considered. This was an unfortunately necessary decision to make. Whilst it reduced the field of possible kernels, it helpfully excluded any overly complex mini-apps.
- **The program must use shared memory parallelism and target the CPU.** Rust's (supposed) zero cost memory safety features are its differentiating factor. The best way to test the true cost of Rust's memory safety features would be through shared memory parallelism, where a poor implementation of memory management will make itself evident through poor performance. Programs which target the GPU rather than the CPU will not be considered, as the current implementations for Rust to target GPUs involve calling out to existing GPU APIs. Therefore, any analysis of a Rust program targeting a GPU would largely be an analysis of the GPU API itself.
- **The program run time should reasonably decrease as the number of threads increases, at least until the number of threads reaches 32.** It is important that any kernel considered is capable of scaling to the high core counts normally seen in HPC.I will be running the kernels on Cirrus, which supports 36 real threads.
- **The program operate on data greater than the CPU's L3 Cache** so that we can be sure that the kernel is representative of working on large data sets. Cirrus has an L3 cache of 45MiB. As each node has 256GB of RAM, a central constraint when working with large data sets is the speed with which data is loaded into the cache. Speed is often achieved by programs in this area through vectorisation,

the use of which can be deduced from a program's assembly code. If there is a large performance difference between Rust and the reference kernels, we can use the program's assembly code to reason about that difference.

- **The program must be written in C or C++.** This restriction allows us to choose work which is more representative of HPC programs that actually run on HPC systems, rather than python programs which call out to pre-compiled libraries. Unlike Fortran, C and C++ use array indexing and layout conventions similar to Rust, which will make porting programs from them easier.
- **The program must use OMP.** This is a typical approach for shared memory parallelism in HPC. Use of a library to do the parallel processing also further standardises the candidate programs, which will lead to a deeper understanding of the kernel's performance factors.

I used this selection criteria to compile a long list of potential kernels to port to Rust. From this long list, I selected the Babel Stream, sparse matrix vector multiplication and K-means clustering.

3.1.1 Babel Stream

Babel Stream is a memory bench marking tool which was developed by the university of Bristol. Babel Stream was written to primarily target GPUs, but it is able to target CPUs too [36]. It is written in C++, supports OpenMP and allows one to set the problem size when executing the program, so we can be sure we exceed the size of L3 cache. My initial tests found the kernel to scale well, and although the program as a whole is quite large, when one ignores parallel technologies excluded by our selection criteria, the amount of code which needs to be ported to Rust falls well within our bounds. I found Babel Stream easy to install and run.

Babel Stream performs simple operations on three arrays of either 32 or 64 bit floating point numbers, a, b and c. The values of a are set to 0.1, b's to 0.2, and c's to 0.0. Stream performs five operations n times on the arrays, where n is a specified command line argument. The operations are listed below:

- **Copy:** Data is copied from the array a into array c
- **Multiply:** Data in c is multiplied by a scalar and stored in b
- **Add:** The values in a and b are added together and stored in c
- **Triad:** The program then multiplies the new values in c by the same scalar value, adds it to b and stores the value in a
- **Dot:** The dot product is performed on arrays a and b. This is when every nth element of a is multiplied by the nth element of b, and summed.

The resulting values in the arrays are then compared against separately calculated reference values, and examined to see if their average error is greater than that number types

epsilon value.

Babel Stream's operations are '*memory bandwidth bound*' [36], because they are so simple. Therefore, when implemented through different technologies, Babel Stream provides an insight into the memory bandwidth of that technology, and gives an indication of how the design choices of that technology influences its performance.

3.1.2 Sparse Matrix Vector Multiplication

The Sparse Matrix Vector Multiplication (SpMV) Kernel [43] forms part of the Parallel Research Kernels suite, developed by the Parallel Research Tools group. Sparse matrix vector multiplication (SpMV) is a common HPC operation, used to solve a broad range of scientific problems [29, 47, 5].

The kernel is mostly one file, sparse.c, which in total is 353 lines of code. The implementation is in C and OpenMP, and my tests found it to scale to a high thread count. As with Babel Stream, the program allows one to set problem size through command line arguments, allowing us to ensure the program operated on data greater than the CPU's L3 cache.

In the selection process, I found that the program's lack of dependencies made it easy to install and run.

The program represents its sparse matrix through the compressed sparse row (CSR) format. This format uses key information about the matrix to avoid storing all of the sparse matrix's redundant zeros in the computer's memory. The information used to do this are the number of rows and columns the matrix has, and the number of non zero values which exist in the matrix. These three values are used to build three vectors, one holding all the non zero values of the matrix, another vector of the same length holding the column indexes for all of those values, in order, and lastly a smaller vector which holds the index at which a particular row starts.

For example, if we wanted the element at 24,32 within the vector, we would look in the 24^{th} element of the row start vector, which would give us the y index of the element. If this did not match the y index we were looking for, in this case 32, we would then look at the next element until we found it. Once we have found the element, we can get the value from the value vector using the index we construct from adding the 24^{th} element of the row start vector, added to however many times we needed to look at the next value to before we found the appropriate y index.

The particular implementation of SpMV which we are porting to Rust uses a user defined grid size, over which a user defined periodic stencil is applied to find the number of non zero entries. The implementation parallelises its initialisation and the actual multiplication of the values using simple `#pragma` statements.

This kernel will hopefully provide a realistic idea of how well Rust can perform one of the most common HPC operations.

3.1.3 K-Means clustering

K-means clustering is a 'process for partitioning an N-dimensional population into K sets' [22], where the number of sets is less than N, and each set of is clustered around a local mean. K-means clustering finds many uses in HPC, particularly in data analysis [4, 27], and is so ubiquitous throughout HPC that implementations of it are already used to evaluate software and hardware [19].

My reference implementation for this code comes from Jaiwei Zhuang's CS205 project [48]. It is written in C and uses OPENMP, and is less than 200 lines long. The data processed by the program can be generated by a script, allowing me to work on an arbitrary amount of data. The kernel is written so that all processing is done by the CPU. It is of particular interest that the kernel reads its data from a NetCDF file, which is common in HPC. The code is well documented and concise.

After the Kernel has read in the program data, it performs the clustering process. The calculated clusters are held in the two dimensional vectors, `old_cluster_centres` and `new_cluster_centres`. The indexes at which values are stored in these cluster vectors correspond to the indexes at which the population is stored in the vector `x`.

First the `old_cluster_centres` array is filled with random data, and then the program begins its central processing loop.

add more info on variables purposes here

- **Expectation:** Assign population points to their nearest cluster centre, by looping over every member of the population, and finding the minimum distance between that point all the cluster centres. This is the stage which is parallelised.
- **Maximisation:** Next, the cluster centres are set to the mean, which is calculated in two steps.
  - **1:** The size of the cluster is calculated by looping over every point and finding its cluster centre, and then incriminating that cluster centres population count. The sum of the points in that cluster is also calculated and stored in the `new_cluster_centres` array.
  - **2:** The sum of the cluster is divided by the size of it, and stored in to the `old_cluster_centres` for use on the next iteration of the loop.

This loop continues until it reaches a pre-defined maximum iteration value, or the sum of the minimum distance values becomes less than a certain tolerance value. The program then writes data back out to the NetCDF file it read the data from originally.

I found this program quite difficult to install and run due to the NetCDF dependency. My first attempt to install NetCDF through the script included in the repository ended with me unable to boot into my laptop. Subsequent attempts to install NetCDF through package managers were also not successful, although they were less damaging to my system. To compile the kernel on Cirrus I had to make sure I had selected the correct combination of NetCDF and HDF5 library versions, mostly through guess work. However, once I had accomplished this task, compiling the program itself was easy. The

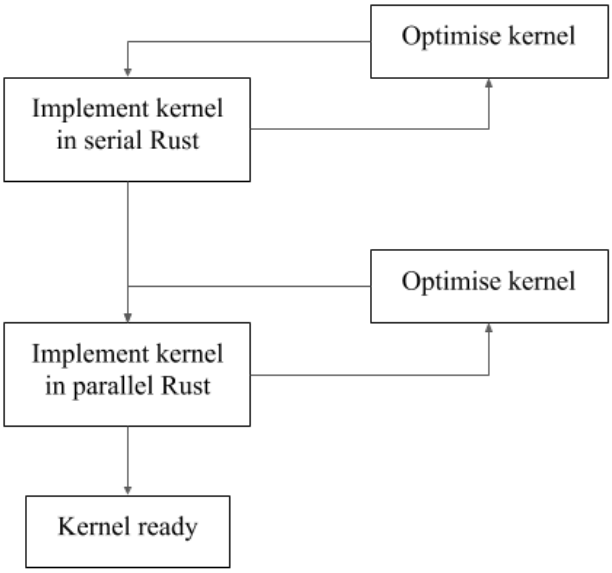

Figure 3.1: Flow diagram for implementation process

kernel then showed itself to be able to scale well enough for the interests of this project.

3.2 Implementation

Implementation of all three programs followed the same process, as outlined in Figure 3.1. The full process would take between three to four weeks to complete for each kernel. I first implemented Babel Stream, then the sparse matrix multiplication kernel and finally the K-means kernel, in that order.

3.2.1 Porting to Serial Rust

Once a candidate kernel is selected, it is implemented in Rust in serial. Any differences between the behaviour of the Rust and the original implementation are thought of as bugs, and are eradicated or minimised as far as is possible. For ease of development, the Rust crate Clap [16] was used to read command line arguments for the program, leading to Rust implementations of kernels being called with slightly different syntax. This difference was deemed to be superficial enough to be allowable. Kernel output was as similar as possible to aid data-collection from both implementations.

Babel Stream was in some ways one of the hardest to Kernels to port to serial Rust. This was partly due to it being the first program which I attempted to port, but also because of Rust's type system and the use of generics. The original C++ implementation of the program uses templates to allow the user to choose to use 32 or 64 bit floating numbers

when running the program. To achieve the same thing in Rust, generic types have to be used, which are defined through traits.

I found that using generics in Rust made reading error messages difficult, but easier to parse once the offending code was removed into a smaller example, and stripped of its generic type. Generics in Rust necessitate slightly cumbersome syntax, for example, `T::from(0).unwrap()` is used to generate a zero of type T. The first part of this expression generates an option type, which in this case is `Some(0.0)`, and is then unwrapped into simply `0.0`. Rust does this to allow programmers to deal with cases where a value of type T is impossible to generate from the input value, such as casting a value greater than $2^{32} - 1$ to a signed 32 bit value. In this circumstance, the value returned would be `None`, which the programmer would then have to deal with. As zero can always be successfully cast to a 32 or 64 bit floating point number, it is safe to simply unwrap the value here, but if it was a number that could not be cast to the type, then the program would crash at this point. A C or C++ program doing the same thing would not crash, but its result would be implementation specific undefined behaviour, or throw an exception.

should I discuss the design choices made here, and their implications for Rust in HPC

The Rust implementation of Babel stream, like the reference implementation, creates a stream object which calls certain functions on its own data sets. This was quite easy to implement as Rust has enough features of object oriented programming, such as allowing objects to contain data and behaviour, for these simple objects to work. However, Rust does not implement inheritance, which is considered by some to be a foundational aspect of object oriented programming [21], and instead uses trait objects to share behaviours. This design choice did not interfere with any of the simple kernels which were implemented, but would certainly be interesting to translate object inheritance from a larger program, maybe a mini-app, into Rust's trait feature.

Whilst the concept of borrowing did take some time to fully understand, I found that the compiler gave very helpful and accurate hints on how to make sure my program complied with the borrow checker. For example, in listing **??**, the programmer is informed that they 'cannot borrow self.c as mutable', and is shown where the function tries to mutate the value. The stream object's triad function, which alters the objects data, but take mutable ownership of the data, through using `&mut self`, where `&mut` is a mutable borrow. Once the programmer implements the compiler's suggested fix, this fragment of code will compile.

```
error[E0596]: cannot borrow `self.c` as mutable, as it is behind a
              `&` reference
  --> src/stream.rs:20:9
   |
18 |     pub fn triad(&self){
   |                  ----- help: consider changing this to be a
    mutable reference: `&mut self`
20 |         self.c[0] = self.a[0] + self.b[0];
   |         ^^^^^^ `self` is a `&` reference, so the data it refers
    to cannot be borrowed as mutable
error: aborting due to previous error
```

The sparse matrix vector multiplication kernel was quite simple to port to serial Rust, as

I was able to ignore parts of the small program which would not be used. As with Babel Stream, I found converting from C's data types into Rust to be a stumbling point due to Rust's safety constraints. For example, in the C implementation, the vector holding the column index of the matrix was composed of values of type `s64Int`, which a signed 64 bit int. This datatype is directly analogous to Rust's `i64` data type, except in C you may use numbers of type `s64Int` to index into arrays, where as in Rust you must only use numbers of type `usize`. Errors of this type are easily dealt with however, as they are explicitly pointed out to the programmer at compile time, and can be remedied with casts in the simple format `as usize`. I found sparse matrix vector multiplication easier to port to serial Rust than Babel Stream, but this could have been that by this point I was already more familiar with Rust's way of doing things.

Given the difficulty I had trying to install the dependencies for the reference implementation of the K-means clustering Kernel, it was surprisingly easy to get NetCDF working with Rust. I simply found a NetCDF rust library [13], which I added to my implementation's `Cargo.toml` file. I was then able to easily compile and use this library within my K-means implementation.

An interesting factor in writing the K-Means cluster in Rust was porting the original helper functions, which were used to make 2d integer and 2d float arrays. In the original C implementation, these 2d arrays were `float**` and `int**`. When I was porting these data structures to Rust, it was important to consider if data locality impacted their use. The original implementation used the data a column wise operation, so that the next datum to be used was likely to have already been loaded in the same cache line as the previous one. This allowed me to write my implementation as a vector of `f32` or `i32` vectors.

The Rust vector of vectors was generated from a single one dimensional vector using the same algorithm as the reference implementation, where sections of the original vector are read into the new vectors within the vector of vectors. Although the original is well suited to C's memory management idioms, it was easy to write the same method in safe rust. The ease with which I was able to re-implement this routine is another suggestion of Rust's ability to replace C's use in HPC.

3.2.2 Serial Optimisation

Next, I eliminated any bugs found in my serial implementation of the code by comparing outputs between my implementation and the reference implementation. During this process I would also move the code away from its C style towards more idiomatic Rust. To achieve more idiomatic Rust, I used the linting tool Clippy [37], which was developed by the Rust team. Clippy includes a category of lints which highlight 'code that should be written in a more idiomatic way' [37]. I implemented all of Clippy's recommended rewrites, which would often include replacing the use of for loops over integer indexes to access vector variables with calls to the vector's `iter()` method. This particular replacement requires code to be rewritten in a much more functional

style.

For example, all of the array operations in Babel Stream where originally written in a C style, and then transformed to use iterators. Listing 3.1, shows the original, more succinct for loop form of Babel Stream's add operation. This style is rejected by Clippy, which prefers the style presented by listing 3.2.

```
for i in 0..self.c.len() as usize {
    self.c[i] = self.b[i] + self.a[i]
}
```

Listing 3.1: Babel Stream Add, before applying idiomatic Rust style

```
for ((c, b), a) in self.c.iter_mut()
                   .zip(self.b.iter())
                   .zip(self.a.iter()){
    *c = *b + *a;
}
```

Listing 3.2: Babel Stream Add, after applying idiomatic Rust style

Whilst the more idiomatic rust style in listing 3.2 is less succinct than 3.1, it does have some benefits which the C style for loop does not possess. For example, if the stream object's c array had been of greater length than its a or b arrays, the more C-like implementation would fail at run time with an index out of bounds error, whereas the more idiomatic code only write to as many elements of c as the least elements there are of any of the arrays it is zipped with.

Also note in listing 3.2 the distinction between the methods `iter()` and `iter_mut()`, the first of which creates an iterator, and the second of which creates an iterator which may change its elements. Although an in-depth investigation was not carried out to see if the compiler made use of any optimisations here from the greater amount of information available to it, the time to run this fragment did decrease when converted to idiomatic Rust, from 0.09501 seconds to 0.09079 seconds.

A bug in my SpMV implementation was found at this stage. When launched with certain parameters, the C version ran without error, whilst the Rust version would panic and fail every time, with the error message:

```
thread `main' panicked at `attempt to shift left with overflow', main.rs:8:13
```

It became apparent that this was occurring because although I had mirrored the types used by the reference implementation, the behaviour of those types differed. In the reference implementation, radius was of type `int`, which is a 32-bit integer. I therefore translated this into a `i32` type in Rust. These values are used as upper limits in an initialisation loop, where intermediate values of the same type are bit-shifted before being stored in the `col_index` array. In C, the operation shown in the listing 3.3 sets foo to 2, when all numbers are 32 bit integers.

```
int foo = 1 << 33;
```

Listing 3.3: Bit shift overflow in C

This occurs because the value 1 overflows and rolls over. In Rust however, this code

causes the program to panic and quit [1]. The Rust language does not consider this behaviour to be unsafe, but finds that that the programmer 'should' find it 'undesirable, unexpected or unsafe' [38]. However, Rust does recognise that some programs do rely upon overflow arithmetic, and provides mechanisms to enable this feature in the language. Fortunately, I was not required to use this feature after changing radius from the `i32` type to `usize` type, which is 64 bits. This choice was made because the radius values were being cast to `usize` more often than they were being used as `i32`. This had the consequence of making the program impossible to bit shift overflow, as a radius of 64 requires a stencil diameter greater than $2^{32} - 1$, which would in turn require a `col_index` array terabytes in size, which the Cirrus hardware does not support.

When this optimisation pass was applied to K-means, it showed the limits of Clippy's linter. Clippy flagged concise for loops with warnings, and suggested overly verbose rewrites of them. For example, on line 110 of the kernel, just before the second part of the maximisation is about to begin, Clippy complains that listing 3.4 has a 'needless range loop'.

```
for k in 1..clusters_d.len as usize {
```

Listing 3.4: Needless range loop

Clippy argues this pattern should be avoided, because 'iterating the collection itself makes the intent more clear and is probably faster' [7]. However, its suggested replacement is much longer, and the deeply chained methods take longer to comprehend.

```
for (k, <item>) in old_cluster_centres.iter()
                                      .enumerate()
                                      .take(clusters_d.len as usize)
                                      .skip(1) {
```

Listing 3.5: Clippy's suggested iterator

It would be difficult to argue that the code suggested by Clippy is idiomatic, as idiomatic code is generally agreed to be code which uses features of the language to achieve conciseness. This code fragment is clearly not concise, and I therefore did not make Clippy's suggested correction.

3.2.3 Parallelisation

I then parallelised the kernel with Rayon [31], at the same loops where the reference implementation uses OpenMP. Sometimes this would be a simple matter of replacing the `iter()` method with `par_iter()`, but parallelising more complex operations like reductions and initialisation was slightly more difficult.

Parallelising Babel Stream was simple. As listing **??** shows, Babel Stream's add operation remains largely the same, only that the `iter()` method has been replaced by the `par_iter()` method, and that the method for each has to be called. As the serial

[1] The compiler will catch this error before run time if it can calculate the value 1 will be shifted by

version of this loop had no inter loop dependencies, it coud easily be transformed from a for loop to a parallel for each loop.

```
self.c.par_iter_mut()
            .zip(self.b.par_iter())
            .zip(self.a.par_iter())
            .for_each(|((c, b), a)| *c = *a + *b);
```

Listing 3.6: Babel Stream Add, parallelised

This pattern was applicable to the copy, multiply, add, and triad methods. The dot method needed more alteration than these methods to be parallelised, as the original, Clippy compliant code was very different to the final code used. The original code in Listing 3.7 updates the sum value from within a for loop before returning it.

```
let mut sum1: T = T::from(0).unwrap();
for (a, b) in self.a.iter()
                    .zip(self.b.iter()){
                      sum1 += a * b;
                    }
sum1
```

Listing 3.7: Serial Dot Product

This update pattern does not work with a Rayon parallel for each loop, as threads are not able to write to a shared variable. The Rust compiler gives the error that the closure does not implement `FnMut`, which is 'The version of the call operator that takes a mutable receiver' [42]. A mutable receiver in this case refers to a mutable variable which is created, and lives on, outside of the iterator's scope. This error demonstrates the utility of Rust's mutable and immutable variables in parallel operations.

To solve this error, the expression is rewritten using the fold method. It was quite difficult to find how to exactly write this, as the serial fold method has a different call signature to the Rayon parallel fold. The final implementation of Babel Stream's fold is shown in Listing 3.8.

```
let sum1: T = T::from(0).unwrap();
self.a.par_iter()
    .zip(self.b.par_iter())
    .fold(|| sum1, |acc, it| acc + *it.0 * *it.1).sum()
```

Listing 3.8: Parallel dot product

In this listing, a zero of type T is generated, and the vectors a and b are zipped together, as before. The fold method then takes two arguments, both of which are closures, or anonymous functions. The first closure is used to create the identity value, which is the value which can be used as the initial accumulator value when the zipped vector of `a` and `b` is divided between threads. The zipped vector of `a` and `b` takes the form

$$[(a_1, b_1), (a_2, b_2), \ldots, (a_{n-1}, b_{n-1})]$$

The fold is applied, resulting in the form:

$$[a_1 b_1, a_2 b_2, \ldots, a_{n-1} b_{n-1}]$$

Which is reduced to the a single number, through calling `sum()`

$$a_1b_1 + a_2b_2 + \cdots + a_{n-1}b_{n-1}$$

Although the initial change of perspective required to use Rayon's fold was confusing, once the cognitive leap had been made the simplicity was clear. Although it was hard to use Rayon's fold method, I did not find it to be prohibitively difficult.

Most of the methods of Babel Stream were easy to parallelise, but this does not necessarily show us the expressiveness of the Rayon library. The parallelised methods were so simple that they were extremely unrepresentative of production HPC code. The sparse matrix vector multiplication parallelisation was more representative of the type of parallelism which is done in HPC, and was therefore more complex than babel stream. Even so, parallelising the central processing loop of SpMV was trivial.

```
for row in 0..size2 as usize {
  let first = stencil_size * row;
  ...
  result[row] += temp;
}
```

Listing 3.9: Serial SpMV

```
result.par_iter_mut()
      .enumerate()
      .for_each( |(row, item)| {
  let first = stencil_size * row;
  ...
  *item += temp;
}
```

Listing 3.10: Parallel SpMV

In the parallel version of the spare matrix vector multiplication I created a parallel mutable iterator over the result vector, and enumerated it. This allowed me to access the items and the indexes of the vector, which I used without changing the internal logic of the for loop at all. The applicability of this common HPC pattern from C into Rust indicates that Rust is an expressive language for HPC.

The K-means kernel's Expectation stage, or E-step, was harder to parallelise. This difficulty arose from trying to do two, seemingly mutually exclusive things, within the same loop. The code required me to update the values of the array and perform a reduction on another variable external to the loop. I had encountered the difficulty of reducing to a shared variable from multiple threads before, with Babel Stream's dot product (see Listing 3.8), but was unaware of how to perform this reduction with side effects.

After some experimentation, I found the solution, a simple `map()` and then `sum()`.

```
for (idx, item) in x.iter()
                    .enumerate(){
    ...
    labels[idx] = k_best;
    dist_sum_new += dist_min;
}
```

Listing 3.11: K-means Rust serial E-step

```
dist_sum_new = labels.par_iter_mut()
                .enumerate()
                .map(|(idx, item)| {
    item = k_best;
    dist_min
    }).sum();
```

Listing 3.12: K-means Rust parallel E-step

In Listing 3.11 I am creating an iterator over `x`, which is a vector of the length as the vector `labels`. I had originally chosen this vector to be the one which created the iterator merely for convenience. I had to change this vector to `labels` in the parallel implementation however, as it is the items of `labels` that need to be updated. I then

used the index value to retrieve the necessary values from `x` to calculate the value of `k_best`. The value of `dist_min` is also calculated for that particular index value. These `dist_min` values are left in a map structure, which is reduced by the sum and written to `dist_sum_new`, yielding the same result as the serial implementation.

reflection on this use pattern

3.2.4 Parallel Optimisation

Once I had parallelised the Rust implementations, I carried out another optimisation pass. This optimisation pass allowed me to find issues caused by parallelism, and make improvements only possible through parallelism. One such improvement was parallel initialisation.

Parallel initialisation is an important feature of programs which run on cache coherent non uniform memory address (CC-NUMA) systems. CC-NUMA systems often use a first touch allocation policy, which means that the each memory address that is written to, or page, is located near to the processor which first touched it. The 18 core Intel Xeon processors on Cirrus use this particular memory allocation policy, which therefore means that 'poorly written applications (e.g., initializations performed at a single processor before main parallel computation begins) will locate pages incorrectly based on the first access and cause several remote memory accesses later' [1]. Without parallel initialisation, the Rust implementation of Babel Stream falls under this definition of a poorly written application. Preliminary testing had also found that the Rust implementation's performance had failed to scale past 8 threads, whilst the C++ implementation's performance continued to increase up to 24 threads.

I had not yet prioritised written parallel initialisation in Rust as there was no clear way to do it. This was largely because in the C++ form of parallel initialisation, allocating the memory to be used and then initialising that memory are two distinct steps, where as in Rust they are the same step.

```
#pragma omp parallel for
for (int i = 0; i < array_size; i++)
{
    a[i] = initA;
    b[i] = initB;
    c[i] = initC;
}
```

Listing 3.13: Babel Stream C parallel initialisation

```
vec![0.0; arr_size].par_iter()
        .map(|_| T::from(0.2).unwrap())
        .collect_into_vec(&mut self.b);
```

Listing 3.14: Babel Stream Rust parallel initialisation

Listing **??** shows how the C++ version of Babel Stream carries out its first touch in parallel, by adding a simple `#pragma` statement to the code. This pattern is not reproducible with Rayon, as it doesn't use parallel for loops, but instead uses parallel iterators. The solution was found to be using the map function to collect values into a vector, as shown in listing **??**.

This routine works by using the `vec!` macro to create a vector of length `arr_size`, where every value in the vector is 0.0. This vector is then used to generate a parallel iterator. The parallel iterator performs a map, taking all values from the vector, and generating a corresponding 0.2, of type T. The `|_|` notation here means that although the closure signature requires a value, that value will not be used in the closure's method. These values of 0.2 are then collected into the vector held in `self.b`.

The use of map here to generate values for parallel initialisation seems like it is an unlikely use case scenario, but I discovered how to use it from the rayon documentation on the map method [33]. Whilst clear documentation always helps a language to become more accepted, this use case was shown to not be as flexible as was needed by all kernels which were ported to Rust, as was found later when attempting to implement parallel initialisation for SpMV.

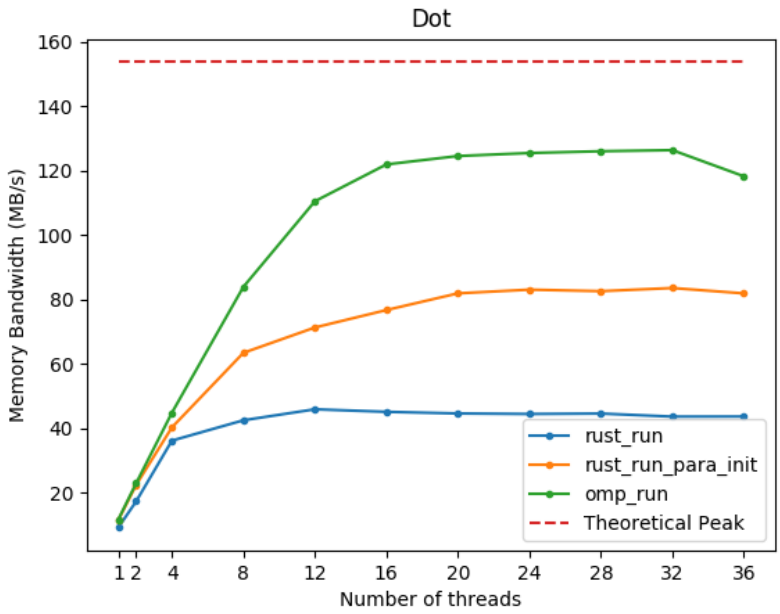

Figure 3.2: Babel Stream Memory Bandwidth initialisation comparison

Figure 3.2 shows, the use of parallel initialisation greatly improved the memory bandwidth of the Babel Stream. However, the improvement in performance still did not bring it to a parity with the C++ and OMP implementation, for reasons discussed further in section 4.1.

The initialisation routine for the Sparse Matrix Vector Multiplication kernel was not as easy to implement. The difficulty was that the parallel loop used to write to elements of the vector `col_index`, wrote to the vector in chunks of five. This made it very hard to translate into Rust, as Rayon only uses parallel iterators, and has no exact equivalent of parallel for loops.

Within an iterator, the programmer may only access the current element of the array, and the next element of the array. This restricted functionality is not expressive enough

for the initialisation routine shown in listing 3.15, as we are unable to step in fours, and we are unable to access the next element of the vector without starting the iterator routine from it's first instruction again.

```
#pragma omp for private (i,j,r)
for (row=0; row<size2; row++) {
j = row/size; i=row%size;
elm = row*stencil_size;
colIndex[elm] = REVERSE(LIN(i,j),lsize2);
    for (r=1; r<=radius; r++, elm+=4) {
      colIndex[elm+1] = REVERSE(LIN((i+r)%size,j),lsize2);
      colIndex[elm+2] = REVERSE(LIN((i-r+size)%size,j),lsize2);
      colIndex[elm+3] = REVERSE(LIN(i,(j+r)%size),lsize2);
      colIndex[elm+4] = REVERSE(LIN(i,(j-r+size)%size),lsize2);
    }
...
}
```

Listing 3.15: SpMV C Parallel Initilisation

Several methods were used in an attempt to solve this problem, including creating an object which would have permanence between iterations. However, this method was unsuccessful as the rayon iterator's did not implement `FnMut`. This problem was ultimately solved using Rust's parallel primitives:

- `Mutex` - Protects shared data through mutual exclusion of locks.
- `channel` - Used to send data between threads.
- `Arc` - An atomic reference counter, which provides shared ownership of a value between threads.
- `thread` - The most basic threading model available in Rust. Platform agnostic.

These primitives were then used to initialise the `col_index` array thusly.

1. The main thread creates a vector, and wraps it in a `Mutex` which is wrapped in an `Arc`, which is labelled `col_index`
2. The main thread uses the `channel` to create a sender and a receiver nodes.
3. The main thread enters a loop, where it creates clones of the n threads worth of `col_index` constructs and sender nodes, which it then moves into spawned threads. Each thread is given a consecutive thread ID starting from 0.
4. Each thread calculates the section of `col_index` it will write to from its thread ID and the size of the overall `col_index`. This section is called `my_col_index`, and is created as a vector of zeros of the correct length for that thread and filled with zeros, which are then overwritten according to the original algorithm.
5. Each thread then attempts to aquire the lock for the shared `col_index`, and checks the length of it.
   (a) If the length of `col_index` is the same as the lower bound of that thread's section, then the thread appends its `my_col_index` to `col_index`, which it then releases the lock for.

(b) Otherwise, the releases the lock and periodically re-acquires it until `col_index` is the right length.

6. Once the last thread has appended their `my_col_index` to `col_index`, it sends an empty message to the master thread.

7. The master thread, which has been blocking, receives this message, and joins all the child threads. It then acquires the lock for `col_index` and unwraps it, so that it can now be used as a normal vector.

This whole routine is 62 lines of code, which is more than four times the original 16 lines of code. It was hard to find this solution, as requiring threads to operate in a specific order is not a typical use case scenario. The solution is complex, and brittle. Its verbosity makes it harder to read than the original code, and the need for careful array calculations feels unfaithful to the Rust philosophy of safety.

When implementing this solution, I kept running into array index out of bounds errors, implying that I was trying to write outside the array boundaries. These errors would crash the program. The cause of this issue was traced to the original implementation of the program, in C. I found this error by checking the final index written to by the threads, which was two more than the length of the array, if the program was run with certain input parameters. This error goes unnoticed in the C version of SpMV, because whilst a write overrun of 16 bits can cause a program to crash, on a modern system like Cirrus it is unlikely to. I corrected my program's threads to write only within the boundaries of their vectors, and filed an issue for this bug on the ParRes Kernels' GitHub repository [46].

Despite the negatives of the Rust version of the parallel initialisation, it did not have any bugs. Figure 3.3 shows the benefit of implementing parallel initialisation for SpMV, which gives the Rust version better scaling than the C version, although its final speed is still slower than the C version's final speed. This difference will be discussed in more detail in section 4.2

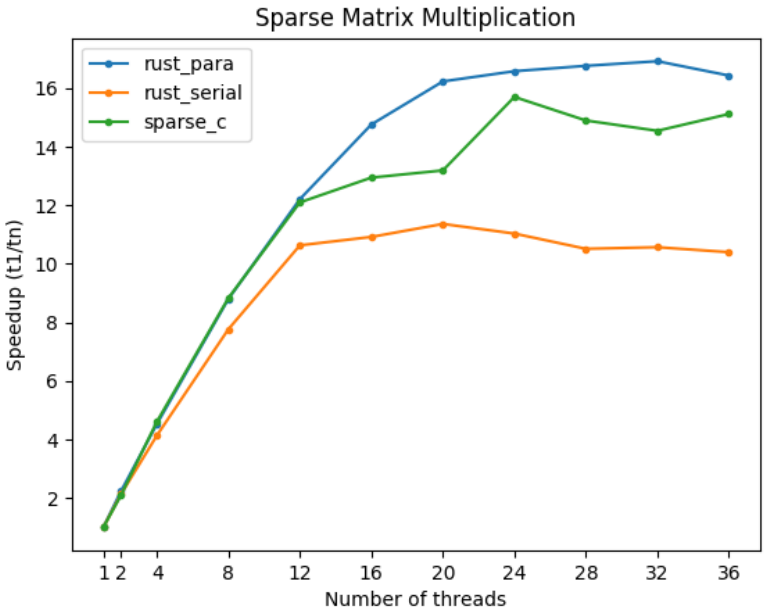

Figure 3.3: SpMV speed up comparison

The K-means kernel did not use any parallel initialisation, and therefore did not undergo parallel optimisation.

3.3 Experimentation

All experiments were run on a single node of Cirrus. No other users had access to that node whilst experiments were being run. Each node on Cirrus has 36 cores, spread across two 18 core Intel Xeon processors. Each processor shares a 45MiB L3 cache between its 18 cores, with smaller L1 and L2 caches for each individual core.Each processor belongs to a separate NUMA region, leading to increased latency when retrieving data from the other NUMA region [6].

To reduce the impact of anomalous runs, both versions of each Kernel were run for 100 iterations, and the average speed was taken. Experiments were submitted to cirrus through the use of Portable Batch Script (PBS) files, which returned program output to timestamped files. A sample PBS submission file can be found in Appendix B.

Babel Stream's experiments were run on vectors of size of 1GB, leading to a total data size of 3GB. Each vector was filled with 1.25×10^8 64-bit floating point numbers. Experiments were run multiple times with varying chunk sizes. Chunk sizes here refers to the amount of data a thread works on, in a given iteration. Varying chunk size gave greater insight into the inner workings of the Rayon library, and see the cost of context switching for threads.

For the sparse matrix vector multiplication kernel, the vectors `col_index` and `matrix` were given a size of 8.2GB, whilst the `result` and `vector` vectors had size 0.134 GB. The generated matrix had sparsity 3.63×10^{-5}. Such large vector sizes were used to ensure that a large amount of data passed through the system's L3 caches. Chunk size was not varied in this instance to allow for a clearer comparison between Rayon and OpenMP in the Roofline model.

K-means used a population of size 73MB, and 16 clusters. This meant that the parallelised E-step of the calculation operated on total of 147MB of data, most of which were 32-bit floats. The rest of the data was 64 bit integers of type `usize` which were used to refer to array indexes. This size of data set exceeded the L3 cache capacity, but was unable to be raised due to instability in the generating python script.

I used version 6.3.0 of GNU project C and C++ compiler, `gcc g++` to compile and run the reference implementations of the kernels. Version 1.34.2 of the Rust compiler, `rustc` was used.

do I need to justify this choice

3.4 Questionnaire

To further assess the suitability of Rust for HPC, I presented staff and students at Edinburgh's Parallel Computing Centre (EPCC) with a questionnaire. The aim of this questionnaire was to examine how easily people with little to no experience of Rust could understand it. The more understandable a language is, the easier it is to learn, the more likely it is to be adopted. This questionnaire would provide valuable data on the usability of Rust as a language.

The Questionnaire was formed of seven multiple choice questions, designed to test the participant's knowledge of Rust. Each question first presented the participant with a fragment of Rust code, and was then asked what that fragment of code did. On some questions, context specific information was given to the participant, such as on question four, which told the participant that 'A vector's pop method return an optional value, or none'. The decision to give the participant this extra information was made so that they could deal with certain functionality which was not unique to Rust, but had a particular name which might be different to something they had already encountered in another language. In this case, the idea of the optional value is seen in other languages, like Haskell's maybe type [8].

An eighth question was also given, which asked the participant how skilled they were at various programming languages. To minimise the factors of impostor syndrome [20] and the Dunning-Kruger effect [18] on this question, (which were hopefully minimal due to the anonymous nature of the questionnaire), each skill level was given concrete examples of what they corresponded to. For example, basic knowledge was the ability to 'write loops, [and] conditionals', whilst advanced knowledge was the ability to 'effectively use the more esoteric features of this language'. Unfortunately, self assessment will never be as good as independent assessment, but without the ability to ask for

a large amount of time from my participants, it had to suffice.

The questionnaire was left out in the lunch area at 12am. Staff and academics were notified of its presence via email at 11:30am and questionnaires were collected at 5pm. I watched the first few questionnaires be completed, but did not intervene in the process. I then returned to my desk to make sure I did not effect the data collection by being there. As all the questionnaires were anonymous, and the people answering them all had a limited amount of time and low investment in the results, people cheating on the questionnaire was not considered a risk.

A full copy of the questionnaire can be found in Appendix A.

Chapter 4

Results

4.1 Babel Stream

Babel Stream's results show that Rust and Rayon are unable to scale as well as C++ and OpenMP. Figures 4.1, 4.2 and 4.3 all show that Rust and C++ have similar performance but that at higher thread counts, there is a great deal of difference between the threading performance of both implementations. In each figure, 1gb refers to the size of the single array in that execution run, and chunk_xxMB refers to how large a subsection of that array the threads are assigned to.

For example, both Rust and C++ having very similar memory bandwidths in the Dot product's serial execution, with Rust at 11.5 GB/s and C++ at 11.6 GB/s, giving a difference in bandwidth of just 1%. However, this difference later widens at 32 threads to 35% with Rust at 87.3 GB/s and C++ at 135.1 GB/s.

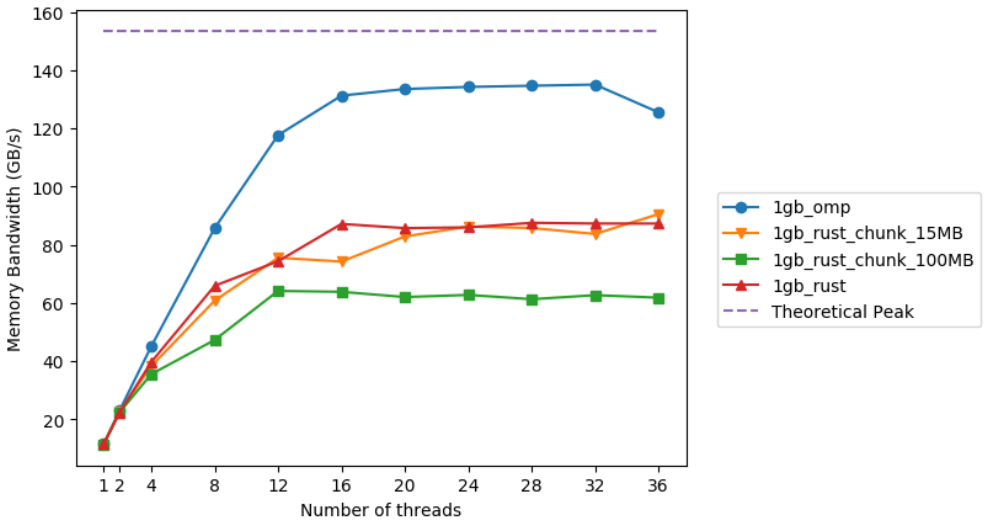

Figure 4.1: Babel Stream — Dot product bandwidth

Whilst the performance difference is not so pronounced for both the add and triad benchmarks, shown in Figure 4.2 and 4.3 it is still quite prominent, with a performance difference of % for add and % for triad. It is interesting that the dot product is able to attain such a higher level of memory bandwidth. Although a deep investigation into why the dot product attains a higher bandwidth than the add or triad benchmarks, I believe likely due to the hardware's implementation of combined operational units, like fused multiply adds, as a cursory inspection of the assembly code here did not reveal any sufficient optimisations at that level.

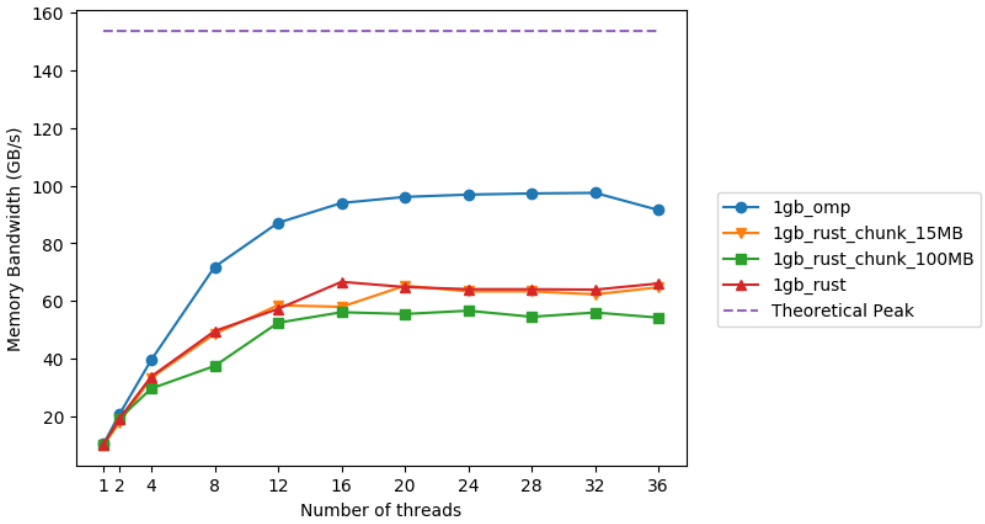

Figure 4.2: Babel Stream — Add bandwidth

This increasing difference lead me to believe that an examination of assembly code would not be beneficial in this circumstance, as it seemed like the low level, assembly implementation of both of the dot product was not that different. Instead, it seemed like the threading implementation was so different that it was what was causing the problem, which is much easier to understand in its high level expression. I decided to investigate the thread scheduling implementation.

The decision to investigate thread scheduling was made because it was easy to investigate. There was also the possibility that Rayon's context switching was more costly to performance than OpenMP's.

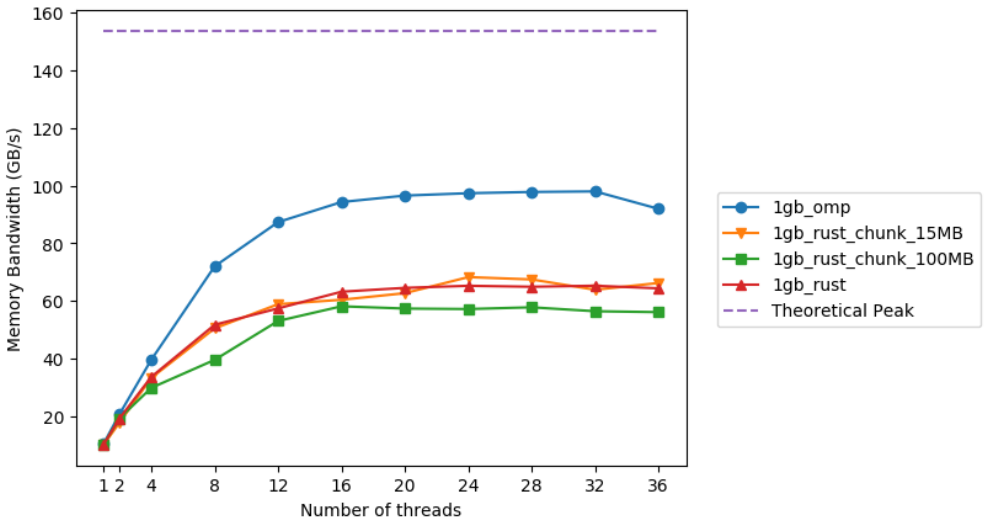

Figure 4.3: Babel Stream — Triad bandwidth

4.2 Sparse Matrix

4.3 K-means

4.4 Questionnaire

The results show zero correlation between competency and score. This suggests that it is difficult to predict how easy a HPC programmer will understand Rust. Figure 4.4

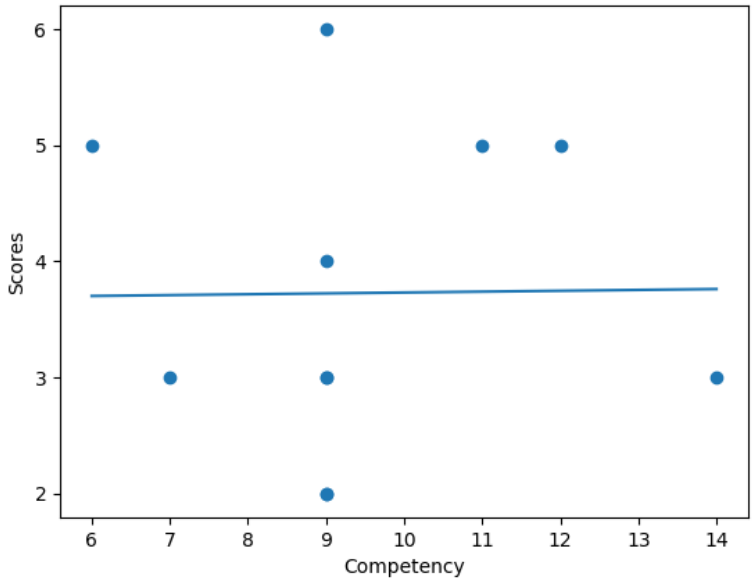

Figure 4.4: Questionnaire — Score against Competency

Chapter 5

Conclusions

This is the place to put your conclusions about your work. You can split it into different sections if appropriate. You may want to include a section of future work which could be carried out to continue your research.

Appendix A

Questionnaire

On the next page, I present a replica of the questionnaire used to collect data.

High Performance Rust MSc Questionnaire

Jim Walker
s1893750@ed.ac.uk

Questionnaire Information

About this project:
This project aims to evaluate the usability of Rust from the perspective of HPC programmers.
Who is responsible for data collected?
Jim Walker
What is involved in this study?
A multiple choice paper questionnaires which asks participants what particular fragments of Rust code do. Participants are also requested to self identify how proficient they are at the following programming languages: Fortran, C, C++, Python, Ruby, Java, JavaScript, Haskell and Rust. I will collect no other data from the participants.
"Responses will be digitised and used to create figures in my MSc dissertation. The data will be retained securely until the dissertation is marked, after which the data will be deleted. A secure back up will also be created and destroyed.
What are the risks involved in this study?
I do not anticipate any risks to participants. Exceptionally, people could try to ascertain which participants got higher marks on the questionnaire from the skill levels the participant applied to the various languages, but the risk of this affecting a participants future career progress would be negligible.
What are the benefits of taking part in this study?
People can test their knowledge on Rust. Once all data has been collected, correct answers will be circulated through the EPCC mailing list.
What are your rights as a participant?
Taking part in the study is voluntary. You may choose not to take part or subsequently cease participation at any time.
Will I receive any payment or monetary benefits?
No.
For more information:
You can contact Jim Walker directly, or his supervisor Magnus Morton, m.morton@epcc.ed.ac.uk

Question 1

What does the function `foo` do?

```
fn foo(m: i32, n: i32) -> i32 {
    if m == 0 {
        n.abs()
    } else {
        foo(n % m, m)
    }
}
```

☐ It finds the greatest common divisor of m and n

☐ It doesn't compile.

☐ It finds the closest prime number to n

☐ It calls itself infinitely.

Question 2

In Rust, `vec!` is used to create a vector. All variables in Rust are immutable by default. What happens when we try to run this program?

```
let v = vec![2,3];
v.push(3);
print!("{:?}", v);
```

☐ [2,3,2] is printed.

☐ [2,2,2,3] is printed.

☐ The program does not compile.

☐ The program compiles, but crashes when it tries to push 3 to v.

Question 3

Idomatic Rust code oten uses patterns associated with functional languages. Given an immutable vector, v, please select what the line of code below does.

```
let a = v.iter().fold(1, |acc, x| acc * x);
```

☐ Every element of v is set to 1, and then copied to a.

☐ Every element of v is multiplied together and the result is stored in a.

☐ Every element of v is multiplied by 1 and the result is stored in a.

☐ The program does not compile.

Question 4

A vector's pop method return an optional value, or none. What does this fragment of code print?

```
let mut stack = Vec::new();

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop() {
  print!("{} ", top);
}
```

☐ Some(3) Some(2) Some(1)

☐ 3 2 1 None None None...

☐ 3 2 1

☐ Some(3) Some(2) Some(1) None None None...

Question 5

What does this fragment of code do?

```
let a: Vec<i32> = (1..).step_by(3)
                       .take(3)
                       .map(|x| x * 2)
                       .collect();
```

☐ Sets a to [2, 4, 6]

☐ The program doesn't compile.

☐ [4, 10, 16]

☐ [2. 8, 14]

Question 6

In this question, a and b are both vectors of the same length. The method `par_chunks` returns a parallel iterator over at most `chunk_size` elements at a time. What does this fragment of code do?

```
a.par_chunks(chunk_size)
    .zip(b.par_chunks(chunk_size))
    .map(|(x,y)| x.iter()
                  .zip(y.iter())
                  .fold(0, | acc, ele| acc + *ele.0 * *ele.1)
    ).sum();
```


☐ Sum reduction

☐ Dot Product

☐ Element wise sum

Question 7

The Rust compiler's borrow checker makes sure that values are mutably borrowed if they are altered from a different function than the one they were created in. What does this program do?

```
fn plus_one(x: &mut i32){
    *x += 1;
}
fn main(){
    let x = 64;
    plus_one(&mut x);
    println!("{}", x+1);
}
```

☐ Print 65.

☐ Prints an undefined value.

☐ It doesn't compile.

☐ Print 66.

Question 8

Please tick the boxes below to show your level of skill in the varying programming languages.

- Basic knowledge: I am able to write loops, conditionals, and can name at least three data types in this language.
- Comprehensive: I can write large programs in this language. I am aware of the most common unique featues of the language, and understand some of them well enough for it to inform my programming in this language.
- Advanced: I have a deep understanding of the inner workings of this language. I can confidently and effectively use the more esoteric features of this language in my programs.

	None	Basic	Comprehensive	Advanced
Fortran				
C				
C++				
Python				
Ruby				
Java				
JavaScript				
Haskell				
Rust				

Appendix B

Launch Scripts

Some people include in their thesis a lot of detail, particularly computer code, which no-one will ever read. You should be careful that anything like this you include should contain some element of uniqueness which justifies its inclusion.

Bibliography

[1] L. N. Bhuyan, H. Wang, and R. Iyer. Impact of CC-NUMA Memory Management Policies on the Application Performance of Multistage Switching Networks. *IEEE Trans. Parallel Distrib. Syst.*, 11(3):230–246, March 2000.

[2] OpenMP Architecture Review Board. Openmp architecture review board members. https://www.openmp.org/about/members/.

[3] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface. https://www.openmp.org/wp-content/uploads/cspec10.pdf, October 1998. Version 1.0.

[4] Sanjay Chakraborty, N. K. Nagwani, and Lopamudra Dey. "weather forecasting using incremental k-means clustering". *CoRR*, abs/1406.4756, 2014.

[5] Shizhao Chen, Jianbin Fang, Donglin Chen, Chuanfu Xu, and Zheng Wang. Optimizing Sparse Matrix-Vector Multiplication on Emerging Many-Core Architectures. *CoRR*, abs/1805.11938, 2018.

[6] Cirrus. Cirrus hardware. http://www.cirrus.ac.uk/about/hardware.html.

[7] Clippy. needless_range_loop. https://rust-lang.github.io/rust-clippy/master/index.html#needless_range_loop.

[8] Haskell Community. Maybe - haskell wiki. https://wiki.haskell.org/Maybe.

[9] The Rust Community. Rust Cookbook - Data Parallelism. https://rust-lang-nursery.github.io/rust-cookbook/concurrency/parallel.html.

[10] Brendan Eich. Future tense. https://www.slideshare.net/BrendanEich/future-tense-7782010, April 2011. Lecture given at Mozilla all-hands.

[11] EPCCed. ffs. https://github.com/EPCCed/ffs.

[12] FFTW. FFTW3. http://fftw.org/.

[13] Michael Hiley. rust-netcdf. https://github.com/mhiley/rust-netcdf.

[14] Ron Hipschman. How SETI@home works. http://seticlassic.ssl.berkeley.edu/about_seti/about_seti_at_home_1.html.

[15] Graydon Hoare. Project servo. http://venge.net/graydon/talks/intro-talk-2.pdf, July 2010. Lecture given at Mozilla Annual Summit.

[16] K. Kevin and Fey Katherina. clap. https://github.com/clap-rs/clap.

[17] Steve Klabnik and Carol Nichols. The rust programming language. https://doc.rust-lang.org/book/.

[18] Justin Kruger and David Dunning. Unskilled and unaware of it: how difficulties in recognizing one's own incompetence lead to inflated self-assessments. *Journal of personality and social psychology*, 77(6):1121, 1999.

[19] Yang L, Chiu SC, Thomas MA, and Liao WK. High Performance Data Clustering: A Comparative Analysis of Performance for GPU, RASC, MPI, and OpenMP Implementations. *The Journal of supercomputing*, 70(1):284–300, 2014.

[20] Joe Langford and Pauline Rose Clance. The imposter phenomenon: recent research findings regarding dynamics, personality and family patterns and their implications for treatment. *Psychotherapy: Theory, Research, Practice, Training*, 30(3):495, 1993.

[21] Barbara Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN Not.*, 23(5):17–34, January 1987.

[22] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press.

[23] A. C. Mallinson, S. A. Jarvis, W. P. Gaudin, and J. A. Herdman. Experiences at Scale with PGAS versions of a Hydrodynamics Application. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 9:1–9:11, New York, NY, USA, 2014. ACM.

[24] Matthew Martineau and Simon McIntosh-Smith. The arch project: physics miniapps for algorithmic exploration and evaluating programming environments on hpc architectures. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 850–857. IEEE, 2017.

[25] Matt Miller. Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape. https://www.youtube.com/watch?v=PjbGojjnBZQ, Feburary 2019. Lecture given at Bluehat Conference 2019.

[26] OpenFOAM. OpenFOAM-plus. https://develop.openfoam.com/Development/OpenFOAM-plus.

[27] I Ordovás-Pascual and J Sánchez Almeida. A fast version of the k-means classification algorithm for astronomical applications. *Astronomy & Astrophysics*, 565:A53, 2014.

[28] Dennis M. Ritchie. The development of the C language. *SIGPLAN Not.*, 28(3):201–208, March 1993.

[29] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 99–108, New York, NY, USA, 2015. ACM.

[30] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: A High-productivity Programming language for HPC with Logical Regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 81:1–81:12, New York, NY, USA, 2015. ACM.

[31] Josh Stone and Niko Matsakis. rayon. https://github.com/rayon-rs/rayon.

[32] Josh Stone and Niko Matsakis. "trait rayon::iter::paralleliterator - fold". https://docs.rs/rayon/1.0.3/rayon/iter/trait.ParallelIterator.html#method.fold.

[33] Josh Stone and Niko Matsakis. Trait rayon::iter::ParallelIterator - Map. https://docs.rs/rayon/1.0.3/rayon/iter/trait.ParallelIterator.html#method.map.

[34] Bajrne Stroustrup. Faqs: When was c++ invented? http://www.stroustrup.com/bs_faq.html#invention.

[35] Bjarne Stroustrup. The essence of C++. https://www.youtube.com/watch?v=86xWVb4XIyE, May 2014. Lecture given at the University of Edinburgh.

[36] Deakin T T, Price J, Martineau M, and McIntosh-Smith S. GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In *Paper presented at P3MA Workshop at ISC High Performance, Frankfurt, Germany.*, 2016.

[37] The Rust Language Team. rust-clippy. https://github.com/rust-lang/rust-clippy.

[38] The Rust Language Team. The rust reference: Behavior not considered unsafe. https://doc.rust-lang.org/reference/behavior-not-considered-unsafe.html.

[39] The Rust Language Team. The Rustonomicon - Data Races and Race Conditions. https://doc.rust-lang.org/nomicon/races.html.

[40] The Rust Language Team. The Rustonomicon - Meet Safe and Unsafe. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html.

[41] The Rust Language Team. The Rustonomicon - Ownership. https://doc.rust-lang.org/nomicon/ownership.html.

[42] The Rust Language Team. Trait std::ops::fnmut. https://doc.rust-lang.org/std/ops/trait.FnMut.html.

[43] Parallel Research Tools. Kernels/openmp/sparse. https://github.com/ParRes/Kernels/tree/master/OPENMP/Sparse.

[44] Linus Torvalds. linux. https://github.com/torvalds/linux.

[45] Andy Turner. Parallel Software usage on UK National HPC Facilities 2009-2015: How well have applications kept up with increasingly parallel hardware? *EPCC*, 2015. Archer White Paper.

[46] Jim Walker. OpenMP Sparse access outside array boundaries. https://github.com/ParRes/Kernels/issues/405.

[47] Xintian Yang, Srinivasan Parthasarathy, and Ponnuswamy Sadayappan. Fast sparse matrix-vector multiplication on gpus: Implications for graph mining. *Proceedings of The Vldb Endowment - PVLDB*, 4, 03 2011.

[48] Jiawei Zhuang. CS205_final_project. https://github.com/JiaweiZhuang/CS205_final_project.