High Performance Rust

Jim Walker

July 28, 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

1 Introduction **1**

2 Background **2**

2.1 Kernels 2

2.2 C/C++ 3

2.2.1 OpenMP 3

2.3 Rust 3

2.3.1 Rayon 3

3 Methodology **4**

3.1 Kernel Selection 4

3.1.1 Babel Stream 5

3.1.2 Sparse 6

3.1.3 K Means 6

3.2 Implementation 7

3.2.1 Porting to Serial Rust 7

3.2.2 Bug fixing, conversion to idiomatic Rust 8

3.2.3 Parallelisation 8

3.2.4 Debug final implementation 9

3.2.5 Babel Stream 9

3.2.6 Sparse Matrix 10

3.2.7 Experimentation 10

3.3 Questionnaire 10

4 Babel Stream **11**

4.1 Development 11

4.2 Comparison 11

5 Sparse Matrix Multiplication **12**

5.1 Development 12

5.2 Comparison 12

6 K-means **13**

6.1 Development 13

6.2 Comparison 13

7 Rust’s usability **14**

8 Conclusions **15**

A Stuff which is too detailed **16**

B Stuff which no-one will read **17**

List of Tables

List of Figures

3.1 Flow Diagram for Implementation Process 7

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' [3].

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

Chapter 2

Background

2.1 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 [4, 7, 5]. 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.

This dissertation will follow a similar approach of evaluating a program through the performance of a mini-app, 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 mini-app 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

2.2 C/C++

When was it developed, who by etc etc, how is it used in HPC today?

2.2.1 OpenMP

2.3 Rust

2.3.1 Rayon

Talk about the underlying nature of Rayon and its random scheduling. Not official library for easy parallelism but it's used a lot in the book.

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 longlist, I selected the Babel Stream, sparse matrix vector multiplication and K-means clustering.

3.1.1 Babel Stream

Babel Stream is a memory benchmarking 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 [9]. 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. Tests found the kernel to scale well, and altough 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.

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 arguement. 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 seperatly caluclated reference values, and examined to see if their average error is greater than that number types epsilon value.

Babel Stream's simple mathematical operations provide an insight into the memory bandwidth of a programming language, and give an indication of how the design choices of a language can influence their performance.

3.1.2 Sparse

The Sparse Kernel [10] forms part of the Parallel Research Kernels suite, developed by the Parallel Research Tools group. Sparse matrix vector multiplication is a common HPC operation, used to sovle a broad range of scientific problems [6, 11, 1].

The kernel mostly one file, sparse.c, which in total is 353 lines of code. The implementation is in C and OpenMP, and 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.

The program reperesents 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 24th elemet of the row start vector, which would give us the y index of the elemnt. 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 24th 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 sparse matrix vector mulitplication 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 initilisation 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

K means is common etc, this implementation also uses netcdf which is also v common so interesting to see how hard it is to get Rust to work with it.

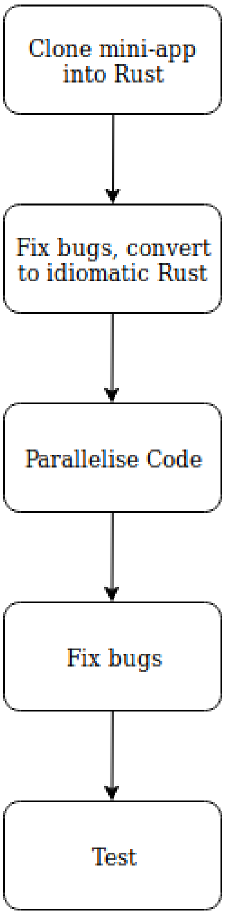

Figure 3.1: Flow Diagram for Implementation Process

3.2 Implementation

Implementation of all three programs follows 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 mulitplication 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 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 ensured to be 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 Rusts type system and the use of generics. The original C++ implmentation of the program uses templates to allow the user to choose to use 32 or 64 bit floating numbers when running the program. To achive the same thing in Rust, generics types have to be used, which are defined through traits. This made reading error messages slightly difficult, but easier to parse once the offending code was removed into a smaller example, and stripped of its generic type. Generics in Rust also necissitate the slightly cumbersome sytax `T::from(0).unwrap()` to generate a zero of type T. This expression generates an option type, which in this case is `Some(0.0)`, and is then unwrapped into simpy `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, 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 succsefully casted 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.

3.2.2 Bug fixing, conversion to idiomatic Rust

Next, I would eliminate 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 conventions towards more idiomatic Rust. To achieve more idiomatic Rust, I used the linting tool Clippy [2], which was developed by the Rust team. Clippy includes a category of lints under which highlight 'code that should be written in a more idiomatic way' [2]. I implemented all of Clippy's recommended rewrites, which would often include replacing the use of for loops to access vector variables with calls to the vectors `iter()` method. This particular replacement could require code to be rewritten in a much more functional style. (should I give an example?)

3.2.3 Parallelisation

I would then parallelise the kernel using Rayon [8] at the same loops where the reference implementation used OpenMP to parallelise its loops. Sometimes this would be a simple matter of replacing the `iter()` method with `par_iter()`, but more parallising more complex operations like reductions and initilisations was slightly more difficult.

3.2.4 Debug final implementation

Once I had parallised the Rust implementation, I would again debug the program process at this stage could be hard to fix as they could come from original implementations. An intersting example of such a bug is discussed further in this section, from the sparse matrix vector multiplication kernel.

Once the implementation process had been finished, testing could begin.

3.2.5 Babel Stream

- Type problems due to generics leading to verbose code and obfuscating debugging
- The compiler did help with type debugging a little, but had limitations - give example
- Idiomatic serial Rust was faster than C like rust, potentially due to iter_mut allowing optimisations? Evidence from triad and add.
- Once I figured out the for_each pattern is was easy to apply it to other operations
- Realised that Rust's serial init was a bottleneck
- difficulty in writing para init as not a common use case scenario, and obfusticated by type

Initialisation is the very verbose - Explain why it's so verbose, process for finding this to be worth doing etc.

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

Explain what's going on in this code fragment, compare it to the C original. Whilst this is a lot, Rust does reduce need for defensive coding. Could do a sloc comparison between original and new, if it was felt to be worth doing.

```
pub fn triad(&mut self){
  let scalar_imut = self.scalar;
  self.a.par_chunks_mut(self.chunk_size)
        .zip(self.c.par_chunks(self.chunk_size))
        .zip(self.b.par_chunks(self.chunk_size))
        .for_each(|((a, c), b)|
             for ((a_i, c_i,), b_i) in a.iter_mut()
                                       .zip(c.iter())
                                       .zip(b.iter()){
                                         *a_i = *b_i + scalar_imut * *c_i
                                        });
}
```

3.2.6 Sparse Matrix

- Bit shift overflow causes Rust to crash not just run on, have to be more careful about kernel input parameters. Initially thought this might be a bug. Give example.
- Found init bug, was very difficult to implement para init. Filed bug report with original project
- remember that class you tried to build to increment stuff? m8

3.2.7 Experimentation

3.3 Questionnaire

Chapter 4

Babel Stream

4.1 Development

4.2 Comparison

Chapter 5

Sparse Matrix Multiplication

5.1 Development

5.2 Comparison

Chapter 6

K-means

6.1 Development

6.2 Comparison

Chapter 7

Rust's usability

Here are some questionnaire results.

Chapter 8

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

Stuff which is too detailed

Appendices should contain all the material which is considered too detailed to be included in the main bod but which is, nevertheless, important enough to be included in the thesis.

Appendix B

Stuff which no-one will read

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] 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.

[2] The Rust Project Developers. rust-clippy. https://github.com/rust-lang/rust-clippy.

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

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.