|epcc|

High Performance Rust

Jim Walker

July 24, 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 Methodology **2**
2.1 Mini-Apps 2
2.1.1 Selection 2
2.1.2 Implementation 4
2.1.3 Experimentation 6
2.2 Questionnaire 6

3 Babel Stream **7**
3.1 Development 7
3.2 Comparison 7

4 Sparse Matrix Multiplication **8**
4.1 Development 8
4.2 Comparison 8

5 K-means **9**
5.1 Development 9
5.2 Comparison 9

6 Rust's usability **10**

7 Conclusions **11**

A Stuff which is too detailed **12**

B Stuff which no-one will read **13**

List of Tables

List of Figures

2.1 Flow Diagram for Implementation Process 4

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' [1].

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

Chapter 2

Methodology

2.1 Mini-Apps

Mini-apps are a well established method of assessing new programming languages or techniques within HPC [2, 4, 3]. 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.

2.1.1 Selection

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

- **The program's kernel (i.e. the part of the program responsible for more than two thirds processing time) should not be more than 1500 lines.** To ensure that I fully implemented three ports of existing mini-apps, it was necessary to limit the size of the mini-apps that could be considered. This was an unfortunately necessary decision to make. Whilst it reduced the field of possible mini-apps, an analysis of the rejected mini-apps found that many of them devoted lots of code to subtle computational variations, which were of more importance to a particular rarefied domain, rather than presenting a novel approach to parallelism. (i could cite some benchmarks here like bookleaf or something)
- **The program must use shared memory parallelism and target the CPU.** Rust's (supposed) zero cost memory safety features are its unique feature. 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 mini-app considered is capable of scaling to the high core counts normally seen in HPC.I will be running the mini-apps 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 mini-app 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 mini-apps, 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 mini-app's performance factors.

Should I quickly describe the mini-apps I selected and explain my choices here or later?

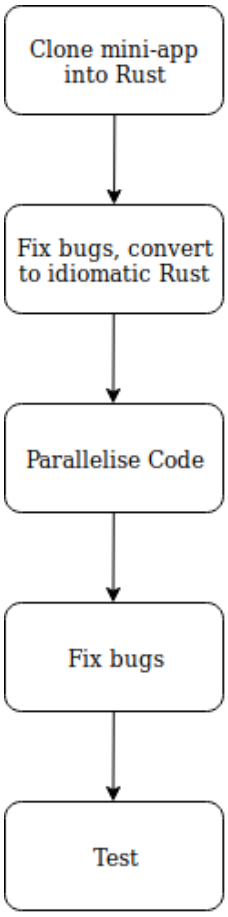

Figure 2.1: Flow Diagram for Implementation Process

2.1.2 Implementation

Implementation of all three programs follows the same process, as outlined in Figure 2.1. Once a candidate mini-app 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 mini-apps being called with slightly different syntax. This difference was deemed to be so superficial as to be allowable. Mini-app output was ensured to be as similar as possible to aid data-collection from both implementations.

Parallising code was done using the Rayon library etc etc,

Bugs at this stage could be hard to fix as they could come from original implementations

testing is discussed in full detail next section, box could instead say ‘Ready for testing’

Babel Stream

- Type problems due to generics leading to verbose code and obfusticating 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
                                             });
}
```

Sparse Matrix

- Bit shift overflow causes Rust to crash not just run on, have to be more careful about mini-app 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

2.1.3 Experimentation

2.2 Questionnaire

Chapter 3

Babel Stream

3.1 Development

3.2 Comparison

Chapter 4

Sparse Matrix Multiplication

4.1 Development

4.2 Comparison

Chapter 5

K-means

5.1 Development

5.2 Comparison

Chapter 6

Rust's usability

Here are some questionnaire results.

Chapter 7

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] Steve Klabnik and Carol Nichols. The rust programming language.

[2] 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.

[3] 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.

[4] 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.