Project Preparation Report

Jim Walker

March 15, 2019

Contents

1 Introduction

This Project Preparation report first provides a review of previous dissertations. It then presents the necessary background and literature for the project, and details a project proposal. A workplan for the project is sketched out, wherein we present chapter summaries and headings for the final dissertation alongside a Gantt chart. We identify risks and provide mitigations for them, before explaining our preliminary findings.

2 Dissertation Review

The two dissertations I have chosen to review are *Assessing the Performance of Optimised Primality Tests* [3] by Cameron Curry, and *Extending ePython to support ePython interpreting across multiple interconnected Epiphany processors* [5] by Dongyu Liang. I have chosen these dissertations because they are relevant to my own project. Liang's *Extending ePython* is focused on a programming language that is not typical in the HPC space, just my project will. Curry's *Optimised Primality Tests* compared implementations of a core HPC function. My project will compare my Rust implementation of some HPC code, with its original C implementation. In this review, I will summarise both dissertations, and then discuss what features of the dissertation I should emulate or avoid.

2.1 Extending ePython

Liang's *Extending ePython* chiefly aims to extend ePython, a python interpreter for the Epiphany processor, to 'support parallel programming on multiple interconnected Epiphanies' [5]. An Epiphany processor has 16 cores, which makes it a useful platform for highly parallel codes. Liang first presents the technologies and paradigms which will they will use in their work. They go on to briefly describe the construction of their Epiphany cluster, before discussing at length their non trivial extension of ePython. Lastly, Liang closes with a results section which proves his achievements.

Extending ePython shows that Liang has made a novel, and valid contribution to the territory. Their technical achievement in extending ePython's parallelism to cover many nodes is notable, especially as Liang claims he to make this change without compromising or altering the ePython programmer's view. However, we can unfortunately only take his word for this, as his appendices only provide his modified ePython, and do not include a listing of the form the original ePython would have taken.

We also question Liang's claims on the stability of their extension to ePython. Firstly, does not precisly define what he means by stability, if he means his system is numerically stable or if it simply does not crash. Secondly, Liang presents their code running on 32 cores across two nodes, presumbly because they only had access to two nodes. Liang goes on to make the claim that their extension 'could have become a cornerstone of the Supercomputer.io project' [5], and saved it from its early end. The Supercomputer.io project was an attempt to collect multiple volunter ephiphany based nodes across the internet to build an extremely large, extremely distributed cluster. Whilst Liang's extension to ePython would certainly have been a valuable contribution to the problems faced by Supercomputer.io, it is somewhat spurious for them to claim that their work could have been a 'cornerstone' for this project. The high latency and large scale of the Supercomputer.io project is not really comparable to running a small cluster of two nodes, connected through a LAN.

2.2 Optimised Primality Tests

In *Optimised Primality Tests* Curry seeks to compare three different implementations of the Fermat Tests, to assess which one PrimeGrid, a large distributed HPC project, should use. He argues that this is an important problem due to the use of extensive use of primes in computing. Curry also hopes to modify

Table 1: Programming Language Usage on some HPC Machines [7]

	HECToR Phase 2a	**HECToR Phase 2b**	**HECToR Phase 3**	**Archer**
Fortran	66.3%	65.2%	66.8%	69.3%
C++	8.9%	2.7%	4.4%	7.4%
C	0.4%	3.6%	5.4%	6.3%
Unidentified	29.1%	30.0%	24.2%	19.4%

the Genefer implementation of the Fermat Test so that its residue calculation is consistent with other Fermat Test implementations.

After detailing the necessary theoretical and practical background to his work, Curry presents his performance analysis. where he does some computer science. Curry finds Genefer to be the fastest implementation, THAN WHAT, grounding his tests in the context of the hardware's limitations WELL ACCECPTED (lots of citations here) hardware . He goes onto document his modification of Genefer's residue calculation, and validates its correctness against other implementations of Fermat tests.

Curry's work is well structured and presented. His inclusion of many listings, equations, tables and graphs provide the reader with a strong grounding for the dissertation. The role of the extra maths is not made clear, is it just filler?

I need to write some more stuff here but I don't know what

3 Background and Literature Review

Programming was once incredibly hard. The very first electronic computers were programmed in hexadecimal by people intimately familiar with their machines. These programs were brittle and often illegible to anyone but the author. However, when compilers were introduced it was argued by some that compilers produced machine code that was slow and not as precise as continuing to write in hexadecimal [1].

Technology has vastly improved since the first compilers were written, and so too has the ease and accessibility of programming. These days programming in Python, one of the most accessible languages, is taught to secondary school children, who require little to no knowledge of the machine they are programming on.

High Performance Computing (HPC) is often seen to be lagging behind these language advances. Programming large, extremely fast programs to run on many cores on many nodes is still hard. Whilst these programs are now less brittle and are easier to read, they are still difficult to program to a high standard. This difficulty arises in part because the two most common languages in HPC, C/++ and FORTRAN,[1] are not memory safe. Consider the program in the listing below, which shows how a programmer might lose track of which thread is writing to which value at which time.

Listing to go here

Until now, languages which can offer memory safety, or other useful types of safety such as type safety, have been thought to be unsuitable for HPC due to the speed cost of using them. The Rust programming language has began to challange this paradigm.

Rust is a programming language, released in 2014 that uses an ownership model of memory to offer strong safety guarantees. '[P]ure Rust programs are guaranteed to be free of memory errors (dangling pointers, doublefrees) as well as data races' [6]. Alongside this, Rust's lack of a garbage collector and fine grained memory control allow the language to achieve speeds comparable to C and FORTRAN. These features of Rust make it a plausible contender to become an accepted HPC language.

[1]Archer has Stats on this, can def find something on google scholar

Some work has been done to investigate the applicability of Rust in scientific programming for bio-informatics [4] and astro-physics [2]. Scientific kernels and apps such as the ones featured in these papers are common in HPC. The results of these investigations have been promising, and show that Rust is as fast, if not faster than implementations written in C or FORTRAN.

4 Project Proposal

Port a HPC mini app or benchmark into Rust. Document process and learning curve. Compare performance. Go into technical detail about how it works like it does.

Mini app criteria, longlist, how will i shortlist or choose from this

5 Workplan

Work item - thing that you get Stretch goal - Rust MPI

Introduction Map the territory, justify choice of benchmark/miniapp

Background Rust, HPC, Other languages

Implementation What was programming like

Performance Analysis How does performance compare to original app? How close do we get to hardware limit?

Conclusion Is Rust any good? efficiency/performance how could this affect message passing

Gantt Chart to go here

6 Risk Analysis

We present our risk register.

Risk	Probability	Severity	Mitigation
Poor port selection	Medium	Medium	Confirm selection with advisers
No knowledge of Rust amongst EPCC staff	High	Low	Rely on Rust community for language issues

Table 2: Risk Register

7 Preliminary Findings

Rust is capable of very aggressive optimisation. It is being used more and more frequently. Parallel iterator from the rayon crate is comparable to OpenMP.

8 Conclusion

9 References

[1] The jargon file: The story of mel. `http://catb.org/jargon/html/story-of-mel.html`. Posted: 1983-05-21 Accessed: 2019-02-20.

[2] Sergi Blanco-Cuaresma and Emeline Bolmont. What can the programming language rust do for astrophysics? *Proceedings of the International Astronomical Union*, 12(S325):341âĂŞ344, 2016.

[3] Cameron Curry. Assessing the performance of optimised primality tests. Master's thesis, EPCC, 2016.

[4] Johannes KÃűster. Rust-Bio: a fast and safe bioinformatics library. *Bioinformatics*, 32(3):444–446, 10 2015.

[5] Dongyu Liang. Extending epython to support parallel python interpreting acrosss multiple interconnected epiphany processors. Master's thesis, EPCC, 2017.

[6] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM.

[7] Andy Turner. Parallel software usage on uk national hpc facilities 2009-2015: How well have applications kept up with increasingly parallel hardware? *EPCC*, 2015.