|epcc|

Project Preparation Report

Jim Walker

March 26, 2019

Contents

Chapter 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 work plan 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.

Chapter 2

Dissertation Review

The two dissertations we have chosen to review are *Assessing the Performance of Optimised Primality Tests* [4] by Cameron Curry, and *Extending ePython to support ePython interpreting across multiple interconnected Epiphany processors* [10] by Dongyu Liang. Liang's *Extending ePython* is focused on a programming language that is not typical in the High performance Computing (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' [10]. An Epiphany processor has 16 cores, or e-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 useful contribution to the territory. Their technical achievement in extending ePython's parallelism to cover many nodes is notable, especially as Liang claims to make this change without compromising or altering the ePython programmer's interface. However, we can unfortunately only take his word for this, as his appendices only provide his modified ePython. Liang does not include a listing of the form the original ePython would have taken.

The introduction of *Extending ePython* contains the assertion by Liang that the Epiphany processor 'is notoriously difficult to program' [10]. Liang does not cite any sources for this claim, which is made somewhat dubious by the existence of a Python implementation precisely for the Epiphany processor.

In the General Methodologies section (3.1.2) of the Construction (3.1), Liang discusses 'The principle of modifying ePython in the whole project is to make as few unnecessary adaptations as possible' [10]. Liang goes on to describe the methodologies used to develop their extensions, and clearly prioritises and justifies their design decisions. Unfortunately, in this section Liang goes on to discuss their implementation of host and device side activities. Whilst these details are useful to the reader in understanding how the e-cores communicate with each other, their inclusion in this section detracts from its clarity. This section could potentially be improved by moving the discussion of host and device side activities into their own part of the dissertation.

Sections 3.2 and 3.3 of *Extending ePython* contain a methodical presentation of Liang's extensions. Liang include all the detail necessary to understand what these extensions consist of, and provides useful

figures to help the reader visualise some concepts. As these are very technical sections, it could perhaps be improved by the additions of some listings of pseudo-code to clarify the author's occasionally verbose description of a complex technology.

We also question Liang's claims on the stability of their extension to ePython. Firstly, they do not precisely define what they mean by stability, if they means their system is numerically stable or if it simply does not crash. Secondly, Liang presents their code running on 32 cores across two nodes, presumably 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' [10], and saved it from its early end. The Supercomputer.io project was an attempt to collect multiple volunteer Epiphany 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.

Liang closes with some benchmarks to display the performance of their ePython extensions. Liang's range of tests is comprehensive, and figure 4.5, showing the good parallel efficiency of the implementation is particularly interesting. We feel that figure 4.4, which shows the results of the strong scaling test on the cluster, could be improved by using more, larger problem sizes, and plotting them against speedup. This change would help make the test more indicative of future use cases, and make change in performance size and its affect on performance starker.

2.2 Optimised Primality Tests

In *Optimised Primality Tests* Curry seeks to compare three different implementations of the Fermat Test, to assess which one PrimeGrid, a large distributed HPC project, should use. He argues that this is an important problem due to the extensive use of primes in computing, particularly cryptography. Curry also hopes to modify the Genefer implementation of the Fermat Test so that its residue calculation is consistent with other Fermat Test implementations.

Curry goes into great detail on the theoretical and practical background to his work. His frequent use of equations in section 2.1.4, 'GFN Primes & The Discrete Weighted Transform', show a firm grasp of the mathematical nature of his work, and how it is effected in hardware. Unfortunately Curry fails to emphasise how this background relates to his performance tests. Whilst he does account for the 'practicalities of digit representation in hardware', he does not make clear what he's doing effect this will have on his work, or how it is relevant to the specific performances of the implementations of the Fermat tests.

Curry goes on to describe his performance testing methodology, introducing the well accepted Roofline Model [20, 6, 1], which he uses to find the peak perofrmance of the processors he will run his test on. This valuable section is well documented and supported by useful figures and brief command extracts. Curry both justifies his choices and details them with clarity, making his testing reproducible and understandable.

Curry finds Genefer to be the faster than both the LLR and OpenPFGW's Fermat Test implementations and thoroughly investigates why that is. He uses tools such as CrayPAT and the afore-mentioned RoofLine model, to identify potential hardware bottlenecks or sources of software overhead. Curry even goes so far as to create a micro benchmark to test an assumption shows a rigorous approach to his experimentation.

Curry is motivated to modify Genefer's residue calculation due to it's lack of consistency with OpenPFGW and LLR. This difference 'is not an indication of incorrect results, it is simply a different representation of the least significant 64 bits'. He goes onto document his modification of Genefer's residue calculation, first implementing it with the same library used by OpenPFGW and LLR and then developing his own implementation of it to maintain the application's portability. Whilst Curry's implementation of a

consistent residue for Genefer is shown to be correct, Curry is correct to call this feature only 'potentially production ready for use', as he only shows us a few comparisons between calculations. To be sure that this implementation was ready would require much more testing of it than has been carried out here.

Chapter 3

Background and Literature Review

Programming languages were once very small and not abstracted from the machine code. The first release of the FORTRAN compiler in 1957 only supported 32 statements [2] . Although it provided the programmer with few abstractions to the machine code which their programs would be translated into, it was quickly picked up by most of the programming community, because programs could be written much faster, much more easily, and were just as good as their machine code equivalent [14]. However, despite these features, some programmers rejected FORTRAN, prefering to write in hexadecimal, either stuck in their ways or believing it to be faster.

Programming languages have improved, and new languages are still released, generally with new features. The minority which was left writing hexadecimal has been replaced with a small group of people writing in C or C++ and FORTRAN. This is often down to the high level of performance which programmers can find in C or C++ or Fortran, evidenced by Turner's 2015 Archer white paper [19], which found that FORTRAN continued to dominate in terms of per cent of total CPU cycles, as seen in table 3.

	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%

Table 3.1: Breakdown of usage by programming language [19]

Whilst Fortran (the capitalisation was dropped after FORTRAN 77) and C or C++ are still being worked on and improved, they are unable to stray too far from their foundations, due to a mixture of cultural and technical reasons. This means these languages are unable to adopt all the new features of more recent programming languages. One of those features is memory safety. In figure 3.1I reproduce from the paper *What can the Programming Language Rust do for Astrophysics* [3]by Blanci-Cuaresma and Bolmont show the kind of errors which can happen in a programming language in which memory safety is not guaranteed. These issues can be harder to debug in massively parallel applications, as tracing which process is writing to which memory location is harder.

D Java Julia when it came out - did anyone ever use it

HPC language needs to operate on large amounts (minumum gigabytes) of numerical data efficiently

No guarantee of memory safety has, in the past, been perceived by some programmers as the price to pay for granting the programmer fine grained memory control. Fine grained memory control is part of what allows programmers to finely tune their codes to achieve the maximum performance from them. However, this means that the programmer must spend a significant portion of their time engaging in defensive programming, to ensure that memory safety bugs do not occur in their code.

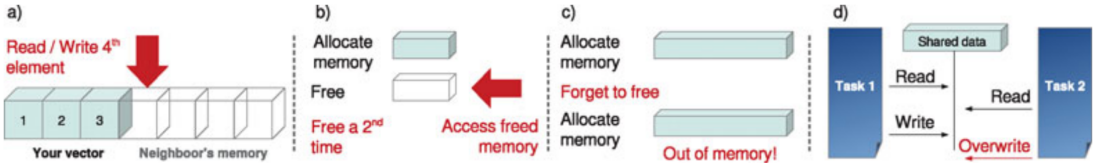

Figure 3.1: 'Potential Memory Safety Related Bugs' [3]. Reproduced with permission

The programming language Rust, first released in 2014, attempts to solve the problem of memory safety, without compromising the programmer's ability to control memory usage. '[P]ure Rust programs are guaranteed to be free of memory errors (dangling pointers, doublefrees) as well as data races' [13]. Rust is not the first programming language to offer some of these features [12, 5, 8]. However, it is one of the first languages to contain all of these memory features at the same time, whilst simultaneously delivering performance comparable to C and Fortran.

Some work has been done to investigate the applicability of Rust in scientific programming for bio-informatics [9] and astro-physics [3]. 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.

Rust's memory safety model does not allow the programmer to write unsafe code.

Rust's memory safety model also ensures that programmers do not need to use their time writing routines which are explicitly memory safe. Consider the listing below where we add four to a value after checking exists. In C we have to explicitly check if a value is null or not before proceeding with the operation. The programmer may forget to include this check, and potentially corrupt memory used by another part of the program.

```
// add_four.c

void add_four(int *foo){
  if(!foo){
    return;
  }
  *foo = *foo + 4;
}

void main{
  int *bar = 3;
  add_four(bar)
}
```

In comparison, the Rust version of this function needs no particular decoration to ensure that it is not carried out on an unexpected memory region. This is partly because Rust does not include null in the language. The closest equivalent that Rust has is the Option<T> type, like C++ std::optional<T>, which can either be `None` or `Some(value)`. This is very similar to Haskell's `Maybe` type. Rust forces the programmer to be mindful of when they might use a null type, and provides a simple, zero cost abstraction for dealing with it.

```
// add_four.rs
// MAKE FOO AN OPTION
fn add_four(foo: i32) -> i32{
  foo = foo + 4
}

fn main(){
```

```
    let bar = Some(3);
    bar.map(add_four);
}
```

Rust has many other features which are useful not just for general purpose programming, but HPC in particular. Rust's ownership memory model restricts values so that they can only ever by accessed through one owner, and must be explicitly borrowed to allow access. This leads to greater certainty about which process's functions are accessing which variables. It is this memory model which allows parallel libraries like Rayon to exist. Rayon implements OpenMP style parallel loops, and because the rustc compiler is built on LLVM, it is able to efficiently vectorise code. Rust is certainly a strong contender for a HPC language, but it is as yet unproven. This dissertation project will aim to investigate how applicable Rust is to HPC. To acheive this aim, we will make use of mini-apps.

In high performance computing, 'small self contained proxies for real applications' referred to as mini-apps [7], are often used to test some particular configuration of a complex HPC stack [11, 17]. As HPC systems and the applications which run on them are so complex, it is often difficult to accurately predict the performance of a HPC stack. This is a fundamental difficulty in comparing HPC configuration performances, and stastics comparing HPC clusters will often refer to a theoretical and a benchmarked performance [18].

Chapter 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.

draft mini app criteria, max around 1000 lines of C, not including driver code. what is rust like to program, i will reflect and inspect on writing in ownership model, compiler's feedback.

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

4.1 Project Goals

- Find a software artifact that is representative of HPC use
- Port that software artefact to Rust
- Assess how easily C or C++ developers can understand Rust
- Find theoretical hardware capabilities
- Single Node Performance Tests
- Strech goal - Multi node performance test
- strech goal - 3 benchmarks

Chapter 5

Work plan

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

Introduction :

- Current state of Programming Languages in HPC
- How are benchmarks/miniapps used in HPC

Background :

- What do programming languages need to be used in HPC?
- Why did D fail to replace C or C++ in HPC? What does this mean for Rust?
- What is Rust? What are its unique features of interest?
- How will my dissertation provide an indication of Rust's ability to succeed in HPC?

Methods

Implementation :

- Overview of Software which I will port
- Technical description of my implementation
- Discussion of what implementation and learning Rust was like
- Correctness testing

Performance Analysis :

- How does performance compare to original app?
- How close do we get to hardware limit?
- Why does performance differ? (potentially go onto machine code analysis)
- How are both implementations affected by scaling?

Programmable :

- Did I like programming in Rust?
- Can other people understand it?

Conclusion :

- Summary of findings
- Further work

Gantt Chart to go here

Chapter 6

Risk Analysis

In planning our project, we have endevoured to be aware of all risks to the project which are potential and plausible. We collected these risks through consultation with our dissertation supervisor and fellow students. We present these risks in our risk register (table 6.1).

The most likely risk in the risk register is risk 3. It is normal for dissertation students to be given technical support in their dissertation projects by their dissertation supervisors and EPCC with relavent domain knowledge. However, in our case the available technical support is likely to be more limited, as Rust is not a well known programming language amongst EPCC staff. Our mitigation for this risk is to rely upon the Rust community for issues which are particular to the Rust language. We do not expect the Rust community have a great deal of knowledge in the HPC space, but our expectation is that EPCC staff will be able to provide the necessary HPC expertise for us, whilst the Rust community provides guidance on the Rust language. It will fall to us to synthesise this knowledge where necessary, which is well within the objectives of this dissrtation.

6.1 Risk Register

We intend to update the risk register at weekly intervals throughout the project lifetime, in an effort to remain cogniscant of risks and make necessary adjustments to our work plan.

ID	Risk	Probability	Severity	Mitigation
1	Poor port selection	Medium	Medium	Confirm selection with advisers
2	No knowledge of Rust amongst EPCC staff	High	Low	Rely on Rust community for language issues
3	Rust Community has poor knowledge of HPC considerations	Medium	Low	No mitigation. We will have to take this risk and work around it.
4	Run out of time for multiple node performance tests	Medium	Low	Jettison this goal.
5	Run out of time to implement all benchmarks	Medium	Low	Drop benchmark.

Table 6.1: Risk Register

Chapter 7

Preliminary Findings

Thus far the work done in this project has been exploratory. We have implemented a simple program in C and in Rust, and compared performance. The program was a simple SAXPY (Single precision A-X plus Y) performed on a large array of random digits. We were not able to use this program to accurately compare the two programs, as we found Rust's aggressive `-release` optimisation (similar in intent to `-O3` found in C compilers) removed much of the processing done by the loop. This exercise illustrated to us some of the difficulties inherit in comparing two programming languages. Not only did we find proof that compiled programs are not wholly representative of their source code, we also found that 'translating' a program from one language to another had its own pitfalls and considerations. We had to consider whether to use follow Rust idioms, like using iterators, or instead remaining more faithful to the C implementation and using a loop to increment a variable to access an array value. Ultimately, we have decided to write idiomatic, or Rustic Rust, unless there is more performant way to express the code.

We have made this decision because, as both C and Rust are turing complete, Rust can express the same algorithms as C, especially through the use of unsafe Rust. Unsafe Rust is a subset of the Rust programming language which allows users to avoid Rust's strict memory model. If we write Rust to be like C, and especially if we write it in a way to avoid or minimise the features of Rust, then the comparison between implementations becomes dishonest. However, we feel it is fair to deviate from idiomatic Rust where there are noticable performance benefits and that deviation is minimal, due to the great emphasis HPC programmers place on performance.

We have installed Rust on Cirrus and compiled and run the simple SAXPY program mentioned above. We have also found the Rust package management system, cargo, to work with no required set up. This package system is similar to python's pip or Node JS's npm and does not require administrator privaleges. We believe this will be useful to HPC developers, as many Rust follows a 'batteries not included' philosophy, requiring users to download and install packages for common utilities like random number generation. Alongside our simple programs, we have compiled some complex Rust programs with higher number of dependencies [16, 15].

- formal criteria for what makes a HPC language
- research has found rust to fit some criteria for becoming a HPC language
- Rust has been installed on Cirrus
- Rust ownership and borrowing has been explored. (should I include saxpy fragment)
- Rust is capable of very aggressive optimisation
- List some Interesting potential HPC features in rust

Chapter 8

Conclusion

Should i refer to the whole document, or ignore the diss review?

In conclusion, I think this diss will be fun. Reviewing the dissertations has made me consider my performance process, and how I will structure my own dissertation.

Chapter 9

Bibliography

[1] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[2] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN Automatic Coding System. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, IRE-AIEE-ACM '57 (Western), pages 188–198, New York, NY, USA, 1957. ACM.

[3] 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.

[4] Cameron Curry. Assessing the Performance of Optimised Primality Tests. Master's thesis, EPCC, 2016.

[5] Python Software Foundation. Python 3 documentation: gc âĂŤ garbage collector interface. `https://docs.python.org/3/library/gc.html`.

[6] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[7] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.

[8] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL*, volume 96, pages 295–308, 1996.

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

[10] Dongyu Liang. Extending epython to support parallel Python interpreting acrosss multiple interconnected Epiphany processors. Master's thesis, EPCC, 2017.

[11] 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.

[12] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. *SIGPLAN Not.*, 40(1):378–391, January 2005.

[13] 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.

[14] Michael Metcalf. The seven ages of Fortran. *Journal of Computer Science & Technology*, 11, 2011.

[15] "ogham". "exa". `https://github.com/ogham/exa`.

[16] "sharkdp". "exa". `https://github.com/sharkdp/bat`.

[17] 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.

[18] "TOP500". "Lists - November 2018 ". `https://www.top500.org/lists/2018/11/`.

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

[20] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.