|epcc|

Project Preparation Report

Jim Walker

March 19, 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 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.

Chapter 2

Dissertation Review

The two dissertations I have chosen to review are *Assessing the Performance of Optimised Primality Tests* [5] by Cameron Curry, and *Extending ePython to support ePython interpreting across multiple interconnected Epiphany processors* [10] 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 High Perf 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'. Liang does not cite any sources for this claim, which is made somewhat dubious by the existance of a Python implementation precisly for the Ephipany 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'. 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 divice 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 techical sections, it could perhaps be improved by the additions of some listings of psuedo-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 precisly 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, 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' [10], 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.

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 intersting. 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 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, 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 harderware. 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 relavent 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 [15][7][2], which he uses to find the peak perofrmance ofthe processors he will run his test on. This valuable section is well documented and supported by useful figures and brief commands. Curry simultaneously justifies his choices whilst detailing them with the necessary clarity to make his testing reproducible.

Curry finds Genefer to be the faster than both the LLR and OpenPFGW's Fermat Test implementations and thoroughly investigates why that is, using tools such as CrayPAT and the afore-mentioned RoofLine model to identify potential hardware bottlenecks or sources of softare overhead. That Curry goes so far as to create a micro benchmark to test an assumption shows a rigourus 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 minimalistic. The first release of the FORTRAN compiler in 1957 only supported 32 statments [3] . 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 [13]. However, despite these features, some programmers rejected FORTRAN, prefering to write in hexadecimal, either stuck in their ways or believing it to be faster [1].

Programming languages have improved, and new languages are still released, gnerally with new features. The minority which was left writing hexadecimal has been replaced with a small group of people writing in C/++ and FORTRAN. This is often down to the high level of performance which programmers can find in C/++ or Fortran, evidenced by Turner's 2015 Archer white paper, which found that FORTRAN continued to dominate in terms of % 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 [14]

Whilst Fortran (the capitalisation was dropped after FORTRAN 77) and 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 inflexibilty means these languages are unable to adopt all the new features of more recent programming languages. One of those features is memory safety. Figure 3.1 from the paper *What can the Programming Language Rust do for Astrophysics* 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.

No gaurantee of memory safety has, in the past, been percieved 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 acheive 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.

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' [12]. Rust

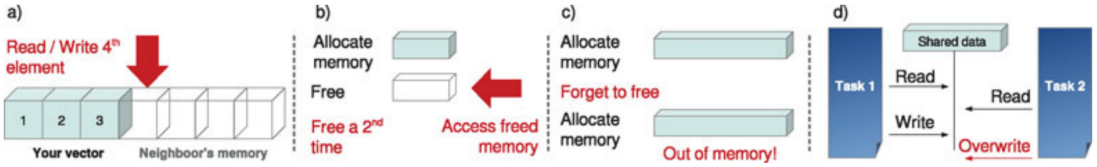

Figure 3.1: Potential Memory Safety Related Bugs [4]

is not the first programming language to offer some of these features [11][6][8]. However, it is one of the first languages to contain all of these memory features at the same time, whilst simultaneously delivering performance comparble 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 [4]. 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 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 carried out on an unexpected memory region. This is partly because Rust does not have include null in the language. The closest equivalent that Rust has is the Option<T> type, which can either be `None` or `Some(value)`. This is very simillar 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

fn add_four(foo: i32) -> Option<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. Values are restricted by the compiler so that they can only ever by accessed through one owner, and must be explicitly borrowed to allow access, leading to greater certainty in which process's functions are accessing which variables. Rust libraries like rayon allow for OpenMP style parallel loops, and whilst still experimental, SIMD support exists in the language through explicit statements. 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.

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.

Mini app criteria, longlist, 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 artifact to Rust
- Assess how easily C/++ developers can understand Rust
- Find theoretical hardware capabilities
- Single Node Performance Tests
- Strech goal - Multi node performance test

Chapter 5

Workplan

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/++ 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?

Implementation :

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

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?

Conclusion :

- Summary of findings
- Further work

Gantt Chart to go here

Chapter 6

Risk Analysis

We present our risk register. We intend to update it at weekly intervals throughout the project lifetime.

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
Run out of time for multiple node performance tests	Medium	Low	Jettison this goal

Table 6.1: Risk Register

Chapter 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.

- formal criteria for what makes a HPC language
- research has found rust to fit some criteria for becomming 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 strucutre my own dissertation.

Chapter 9

Bibliography

[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] 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.

[3] 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.

[4] 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.

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

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

[7] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[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] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. *SIGPLAN Not.*, 40(1):378–391, January 2005.

[12] 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.

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

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

[15] 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.