Computing Project
UFCE3B-40-3
Project Draft

Neil Donnelly
10032122
neil.m.donnelly@gmail.com

February 15, 2013

Exploiting Multi-Core Processors through Functional Programming

λ

Acknowledgements

To Ron Burgundy. . . You stay classy

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Contents

1 Introduction

1.1 Context

1.1.1 Moore's Law

In 1965, George Moore observes that the number of components which can be deployed onto a semiconductor integrated circuit, or 'chip' appears to be exponentially doubling every year (Mack, 2011). Moore conjectures that this trend will continue for the proceeding ten years; with increasing chip area, decreasing feature size and improved circuit designs (Mack, 2003).

Moore does not however, observe that such rapid growth is infinitely sustainable, and predicts a declination in the rate of exponential growth to a doubling of components over a more sobering two year interval (Mack, 2011).

This observation is known as Moore's law, and drives semiconductor development with the inevitability of a self-fulfilling prophecy. Indeed, development out paces the law, resulting in a doubling of components in cycles of circa 18 months (Mack, 2011).

1.1.2 The Benefits of Moore's Prophecy

The dramatic increases in semiconductor development result in chips which are cheaper, lighter, faster, less power hungry and more reliable than their predecessors (Mack, 2011). This almost mystical effect of Moore's law on the semiconductor industry allows for (Mack, 2011) *'a life without tradeoffs'*, in which the cost of producing chip components continues to reduce along with their size. The net result being an almost constant production cost for manufacturers and an environ-

ment in which further chip development appears to be protected from traditional economic factors (Mack, 2011).

The cumulative effect of exponential gains gives present day chips incredible power and complexity, which both underpin the continued growth of the software industry and facilitate an inevitability of ubiquitous computing. Indeed, many processing intensive applications which are taken for granted, such as high definition video would be inconceivable without the exponential performance gains of which Moore's Law professes (Agarwal & Levy, 2007).

1.1.3 Moore's Gap

Unfortunately, whilst 'The Law' continues unabated, the level of performance which can be wrung from any given chip is beginning to decrease (Agarwal & Levy, 2007). This gap between the law's continued proliferation of micro architecture and the real world performance gains follows Pollock's Law (Borkar, 2007), which states that doubling the logic of a processor core results in performance gains of a mere 40%.

Pollocks Law is further compounded by what is proving to be an end to Chris Mack's *'no tradeoff'* era (Mack, 2011). The faster, smaller, less power hungry transistors rely upon voltages scaling down along with their size (Kadin & Reda, 2008). Unfortunately, with the laws of physics being less than accommodating in this regard; this is not currently possible (Kadin & Reda, 2008). The net result is that the designers are unable to furnish the world with the ever increasing clock speeds to which it is accustomed. This *'Moore's Gap'* (Agarwal & Levy, 2007) requires that chip designers implement new chip architectures in order to circumvent the immense power demands and thermal output which are

the consequence of complex, high frequency cores (Kadin & Reda, 2008).

1.1.4 A New Breed of Processor

In order to bridge the expanding gap between Moore's Law and real world performance; manufacturers turn their focus from producing chips which sport massively powerful single cores to the development of 'multi-core' processors (Borkar, 2007; Kadin & Reda, 2008). This new breed of chip comprises several processing cores, which run at lower clock speeds; the net result being that the aforementioned power and thermal constraints are conveniently sidestepped when implemented with an adequate frequency and power management scheme (Kadin & Reda, 2008).

Unfortunately however, it is hypothesised (Borkar, 2007) that continuation of multi-core development again results in unfeasible power demands, which are again attributed to the issue of voltage scaling. It is estimated that a single chip built upon a 300mm^2 processor die will require around 1000 watts of power in order to operate (Borkar, 2007).

It is therefore proposed that development of new 'many-core' processing architectures which are able to provide a panacea to the issues of voltage scaling (Borkar, 2007). The many-core architecture is similar to that of multi-core, with a significant reduction in core complexity and proliferation in the number of cores on any given chip. The result is a processor comprised of a plethora of much simpler cores, which being mindful of Pollock's Law allows for processors with the highest possible performance to complexity ratio (Borkar, 2007).

1.1.5 A New Breed of Problem

In 1967, Gene Amdahl addresses the AFIPS Spring Joint Computer Conference (Amdahl, 1967). He argues in favour of more powerful single cores; stating that for the vast majority of cases only a fraction of a program's code could be effectively parallelized (Amdahl, 1967). Amdahl's observations, simply put, state that the performance gains from a multi-processor or multi-core architecture are limited by the volume of serial code which is present in any given system. (Borkar, 2007).

A re-evaluation of Amdahl's Law (Gustafson, 1988) suggests that the initial observations regarding the volume of system code which can be parallelized may be misleading. Gustafson (Gustafson, 1988) conjectures that Amdahl's Law does not account for the problem size scaling with the available processing power, and that it is natural for software engineers to take advantage of new hardware in previously unforeseen ways. This hypothesis is further supported by Hill & Marty (Hill & Marty, 2008); who postulate that a reconciliation between Amdahl's observation and Gustafson's re-evaluation is required in order for the continued exploitation of Moore's Law. Unfortunately, there is no perfect solution to draw upon. The current consensus however, is that it is imperative for software engineers to begin to find ways in which to parallelize increasingly greater percentages of any given system in order to harness the maximum amount of available processing power (Gustafson, 1988; Borkar, 2007; Hill & Marty, 2008).

Herb Sutter (Sutter, 2005) recalls a pattern of chip and software development, which through the 1990's is known as *"Andy giveth, and Bill taketh away"*. This refers to the performance gains created by Intel's chips and the subsequent, al-

most inevitable expansion of the latest Windows release to consume the benefits thereof. This phenomenon however, is not exclusive to software created by the Microsoft corporation; but has perpetuated itself over the lifespan of the software industry (Sutter, 2005). In what appears to be software development's interpretation of Parkinson's Law (Parkinson, 1957); increases in chip throughput are rarely treated with a high level of delicacy (Sutter, 2005). The result is software which appears to expand over the available resources in an almost effortless manner.

This laissez-faire attitude towards development stems from the knowledge that no matter how inefficient a software product may be; within 18 - 24 months such inefficiency will be counterposed by further increased core speeds (Sutter, 2005). However, in a similar manner to which Chris Mack's *'no tradeoff'* era no longer applies to chip development (Mack, 2011); this *'free lunch'* for software developers appears to have reached its terminus (Sutter, 2005). Indeed, with it being suggested that more cores and lower speeds are the way in which to bridge Moore's Gap; many vendors could begin to see the performance of their products diminish significantly on newer hardware (Sutter, 2005; Sutter & Larus, 2005).

In response, a different approach is required (Sutter & Larus, 2005). Creating safe, threaded software for anything other than a trivial application is demanding, difficult and expensive (Sutter & Larus, 2005). With the future being multi-core, it appears that the software industry should seek to adapt to this new parallel world if it is to continue to advance. Somewhat ironically, a paradigm which may facilitate this brave new world of parallel development is published in the 1930's and implemented in the 1950's. It is known as functional programming, and predates what is widely considered to be the foundation of computing; the universal Turing machine (McCarthy, 1981; Copeland, 2004; Hinsen, 2009).

1.1.6 Language Selection

In all endeavours it is important to use the correct tool for the job at hand. Selection of a programming language will influence factors such as the effort required to complete the code, its runtime efficiency, memory usage, reliability and portability (Prechelt, 2000). Conveniently, the ease at which robust concurrent systems may be produced is also influenced by the implementation language (Peyton Jones, 1989; Hinsen, 2009).

In 1989 Peyton Jones (Peyton Jones, 1989) postulates that functional programming is inherently suited towards the production of parallel systems. The conclusion reached is an expectation that functional programming and parallelism will begin to find more widespread use (Peyton Jones, 1989). More recently, the realisation that the industry requires tools more suited to developing concurrent code is gaining further acceptance (Hinsen, 2009). The purported benefits of these tools must therefore be investigated in order to determine if they are worthwhile ventures.

1.2 Research Question

Can functional programming exploit the potential performance gains of multi-core processors?

2 Background

2.1 Concurrency

2.1.1 State

Any discussion regarding concurrency should likely begin with the consideration of both state and mutability. This discussion is no different.

> An object is said to "have state" if its behaviour is influenced by its history. (Abelson & Sussman, 1996)

The seminal book; Structure and Interpretation of Computer Programs (Abelson & Sussman, 1996) is the companion text for class 6.001 at the Massachusetts Institute of Technology. Between the years of 1980 and 2008, 6.001 fulfils the role of introducing first year students to programming and software engineering at one of the leading institutions for technological research and computer science.

What's interesting about the text is that it patiently waits until page 217 before approaching the matters of state, modularity and objects (Abelson & Sussman, 1996). Prior to page 217, students are instead introduced to concepts such as expressions, linear recursion, tree recursion, iteration and data abstraction. Indeed, the first appearance of an assignment operator occurs on page 220, with almost 50% of the book having elapsed (Abelson & Sussman, 1996).

In contrast, if compared to a recommended text for programming 101 at another institution, for instance: Learning Java (Niemeyer & Knudsen, 2005). It is found that the first appearance of the assignment operator is on page 31; the related heading of Instance Variables being only 12 pages further ahead (Niemeyer & Knudsen, 2005).

Why then, given that assignment and state are fundamental to producing meaningful systems (Abelson & Sussman, 1996) is there such a delay regarding this particular discussion? Thankfully, Abelson and Sussman are not beyond detailing their motives.

> With objects, we must be concerned with how a computational object can change and yet maintain its identity. This will force us to abandon our old substitution model of computation in favour of a more mechanistic but less theoretically tractable *environmental model* of computation. (Abelson & Sussman, 1996)

It would appear that the addition of the assignment operator and the concept of mutable state is of concern to the authors. With the results of their functions becoming less mathematically defined and more unpredictable, they have reached the somewhat counter-intuitive conclusion that assignment is in fact a more complex subject to comprehend than recursion (Abelson & Sussman, 1996). A conclusion which the vast majority of first year computing students would, most likely, emphatically refute. They elaborate further, detailing the primary culprit of this increased complexity.

> The difficulties of dealing with objects, change, and identity are a fundamental consequence of the need to grapple with time in our computational models. Those difficulties become even greater when we allow the concurrent execution of programs. (Abelson & Sussman, 1996)

Naturally, with Structure and Interpretation of Computer Programs focusing on the functional language of SCHEME, there is less of a reliance upon assign-

ment when discussing the basics of the language. However, Abelson & Sussman clearly state

There is no escaping the issue that developing concurrent software is difficult (Sutter & Larus, 2005; Hwu, Keutzer & Mattson, 2008).

Ah (Hwu et al., 2008) (Sutter & Larus, 2005) (Hwu et al., 2008)

2.1.2 Mutability

Mutable objects bad

Threads - concurrency - simultaneous execution - multi cores

single processor - simulated concurrency - interleaved

avoid producing inconsistent data

- more threads the more difficult more steps - difficult

- on the programmer - clojure not a parallel programming language, concurrency comes from programmer deciding what to make concurrent.

– what we do now to manipulate collection multiple threads - lock/synchronised - one thread can have lock at time, else it blocks - requires co-ordination, you must set it, manual decide to do it, convention. - synchronised can only help you with single step job.

– can't enforce locks, people must choose to do this, no enforcement, difficult to keep track of what's been locked - devs may not lock. decide later to share something, must now figure how to retrofit the locks.

– bottleneck on multi cpu machine, as blocking – no longer a maybe

– single lock is too much, readers block other readers, locking out readers as well as writers. – reader lock solution = special kind of lock, make a multi-reader single writer block, one writer at a time. writers still wait for readers.

– copy on write collections – unusual, good multi-threaded propertied. when collection is changed, copy is made inside and then atomically switched, so no locks required. reads are not blocked.

problem with copy-on-write collections : very expensive to write, as full copy made. use for mostly read occasional write.

– multi-step interaction still requires a lock for c-o-w collections. that's all you get in java or c#, still only talking about collections.

– persistant data structure in functional programming - structure is immutable. so to change you need to produce new DS, only. trick with persistant taka structure is – maintains performance guattuntees of the data structure. new versions cannot be full copies, linear time to copy everything is bad, so need a trick to make the change. share structure between changes.

– magic of clojure persistant hashmap/vector -

– structural sharing - shares as much structure with the update as possible.

– thread safety, data structure can never change, thread safety – no exceptions thrown from clojure structures

– back to locking – in real life, multiple activities and multiple data structures. - very hard. need to control more than one data asset in one operation.

– two options - coarse granulaity locking = create lock, this lock you use to control this set of things – fine gran lock = acquisition of one lock – can be documented well; very safe, safest if you're gonna use locks – confusing if you're gonna use multiple sets of locks – need to know what locks to use for data structures, not good to be locking multiple data structures, queuing for set locks resut in much needless blocking; also questions about reading. fine gran – lock the data structures, but you will need multiple locks; very dangerous (where most pro-

grammes end up) when software changes to need threads where there was never a need. new requirements. remembering previous strategy for locking will make such things impossible. basic strategy is a lock order - A= a,b Y= b,a - recepie for deadlock - A gets a Y gets b - difficult to enforce locking orders, numbered order, alphabetical ordered. fine grain gives beter throughput than coarse grained locks. should i lock while reading?

— concurrency methods — conventional way – direct references to things that can change - soon as you do it you're stuck with it. lock n' hope for best - manual - all decided by convention - not in programming language, convention is outside of the program - bad.

— clojure — must feel that things are changing sometimes - keep the reference, but the reference can mutate, indirect. indirect references to things which never change. can change the reference atomically refFred - at one point in program it refers to one thing, later refers to something else. don't really have a pointer to the structure, have a referemce. references are mutable, only thing which can change - ref types - they have concurrency semantics. must use functions to make them point to diff things via functions, enforced and automatic.

– from convention – to enforcement –

– no locks! – – three refs in clojure – – var= global values, can also be bound inside other threads - for isolating changes in threads. changes in threads will not cut across to other threads - like special variables in common lisp with thread semantics - stack discipline, will unwind to their initial state

– refs= transactional structure, co-ordinated changes, can be seen from multiple threads. refs make changes which can be seen from multiple threads vis transactions atomic – software transactional memory system – application level –

ref can only be changed in transaction, change is atomic and isolated. all changes make to set of refs in transaction - or none - isolated, you don't see the effects of other transactions and they don't see yours till commit – speculative and automatically retried. transactions will rety so you can be second when attempting transactions. therefore transactions cannot have side effects. refs only manipulated in transactions. All changes appear to happen at the same point in time. there is no staggering of changes. – consistent view of the entire world - readers never block writers, writers never block readers.

–agents= make requests for something to change, will do so in it's own time eventually; other threads can see the changes. agents are like workers, refs are like – manager independant state - agent is responsible for state - you send the agent an action, which says apply this function to whatever state you have inside of yourself. - you send actions to agents, the request gets queued up in the agent, and eventually this will eventually happen. high degree of independant. you can request, immediately continue and the request will change eventually. one action per agent serialised by the system. none-blocking. things happen in order – state is always available, can access it. may not reflect all actions which have been sent to it, but not having to wait is awesome. can make sure the work is done with a wait - so will block. - agent can send actions to other agents, will happen in order - agents can co-operate with transactions, can dispatch action within transaction, will only be sent when it commits, so can tie a side effect.

– lock the world and give me a valid report – consistency valuable – enforced consistency

2.2 Programming Languages

2.2.1 Imperative Programming

The overwhelming majority of programs in execution today are written in imperative programming languages, which in turn are derived from Alan Turing's Universal Turing Machine (Turing, 1936; Barendregt, Manzonetto & Plasmeijr, 2013). The nuances of imperative programming require little explanation and are well understood by the computing community in general.

2.2.2 Functional Programming

Derived from Alonso Church's untyped lambda (λ) calculus (Church, 1936). Functional programming leans towards using immutable data structures where possible and uses a mathematically tractable method of substitution to resolve the values of functions, and subsequently a particular computer program (Abelson & Sussman, 1996; Barendregt et al., 2013). In order to understand functional programming however, it is imperative that the λ calculus is first discussed.

2.2.3 λ

In 1932 Alonso Church publishes a paper detailing a formal model of computational expression (Church, 1932). The model is an extension of studies conducted into mathematical functions by Frege in 1893 and intends to allow the proof of computations by methods of substitution and variable binding (Rosser, 1984). Unfortunately, the paper and the method within are proved inconsistent when subjected to further investigation (Rosser, 1984).

Unperturbed, Church returns in 1936 with a paper proving the impossibility of

the Entscheidungsproblem (Church, 1936; Copeland, 2004). He does this using a refined and much stronger model of computation; the untyped λ calculus (Church, 1936). Alan Turing independently addresses the same problem, reaching the same conclusion through the computational method of his Universal Turing Machine (Turing, 1936; Copeland, 2004).

It is further observed that the computational processes of untyped λ calculus and Alan Turing's machine are equivalent in the types of functions which they are able to express. This parity between the two calculation methods is formalised as the Church-Turing Thesis (Copeland, 2004; Barendregt et al., 2013).

2.2.4 λ Notation & Evaluation

It is not the intention to give a complete definition of the calculus, however it is useful to understand some of the core concepts before discussing the features of functional programming languages.

As stated, λ calculus works by resolving the values of a function or program by a method of substitution. It consists of what would appear to be limited syntax and expressive potential; however is powerful enough to represent all computable functions (Church, 1936). The syntax is defined below in Backus-Naur form (Backus, 1959).

$$< expression > \ := \ < name > | < function > | < application >$$
$$< function > \ := \lambda < name > . < expression >$$
$$< application > \ := \ < expression >< expression >$$

The language allows the definition of three different λ expressions: names of

variables, function definitions and the application of function definitions to either variables, other function definitions or other applications.

A λ function (also known as an abstraction) is a representation of an anonymous function, which defines the *name* to the left of the '.' operator as an argument, with the *expression* on the right being the return value (Church, 1936). Hence the function $\lambda x.x$ would be a function which returns the supplied argument. As functions and names are both expressions, this means that λ calculus allows functions to be both supplied as arguments and returned (Church, 1936).

Expressions in λ calculus are evaluated by applying the function on the left to the expression on the right. Parenthesis are used in order to make explicit the order of application. For example, this series of expressions:

$$\lambda x.x\lambda y.y\lambda z.z$$

Could parenthesise to:

$$(\lambda x.x)((\lambda y.y)(\lambda z.z))$$

Or alternatively:

$$((\lambda x.x)(\lambda y.y))(\lambda z.z)$$

At this point, it is imperative to understand the difference between variables which have been bound and variables which are free to assume any value. If a function $\lambda x.y$ is declared, the *name* given to it's argument, in this case x, is considered bound within the function's *expression*. The result of this variable

scoping is that should we supply an argument to x, say t:

$$(\lambda x.y)t$$

Any occurrence of x within the function body y will have the same value as the parameter bound to the λ symbol. In this case, the value of λx is t. In contrast, y is a variable free to take on any value, as it does not exist within an expression in which y has been bound to a λ.

This distinction between free and bound variables allows for the same name to be used several times but exist with different values depending on its scope. It also introduces the concept of a closure, in which any functions nested within the expression of another function use that function's bound variable. For example:

$$\lambda$$

Somewhat interestingly, λ calculus only allows the definition of functions which accept a single argument. In order to evaluate multiple arguments, a technique called currying is employed. This consists of creating a function which accepts the first argument. The function then returns another function which will accept the second argument. This continues until such a point that a function is produced which returns an expression. If we consider a function definition which accepts three variables x, y, z and returns another variable r; its curried notation would be:

$$\lambda x.\lambda y.\lambda z.r$$

In order to invoke the function it must be supplied with three expressions (argu-

ments):

$$(((\lambda x.\lambda y.\lambda z.r)t)u)v$$

In this, t is supplied to λx, u to λy and v to λz. Due to the scoping and binding of variables, it is then possible to create an addition function in the following manner:

$$((\lambda x.\lambda y.(x + y))5)6$$

The bound variable x

Applications in λ are resolved via a method of reduction, which attempts to produce the simplest possible answer by recursively replacing function definitions with the values of their computations. Take the λ application:

$$\lambda x.$$

2.2.5 LISP

List Processing, or LISP (McCarthy, 1981)

2.2.6 Clojure

2.3 How Functional Programming May Help Concurrency

(Hinsen, 2009) (Sutter, 2005) (Sutter & Larus, 2005) dsdds

3 Hypothesis

It is hypothesised that clojure's immutable data structures and transactional assignment allows developers to produce systems which remain consistent over multiple threads of execution. Furthermore, parallel updates to unordered lists offer an opportunity to both break down a software problem into much smaller component parts than would otherwise be possible. If future chip architectures are to sacrifice raw serial processing power for greater cumulative throughput across multi, or indeed many cores; then allowing software developers to maintain a consistent view of the system throughout its runtime would appear to be the key to exploiting the potential performance gains of this emergent architecture.

It is therefore postulated that an implementation of a parallel problem in clojure will yield greater performance over the more common imperative languages as the number of system cores increase.

4 Objectives and Test Specification

4.1 Objectives

The overarching objective of this project is to determine if Clojure is able to outperform traditional imperative languages when faced with a suitably concurrent problem. To this end, the same program will be implemented in a number of different languages in order to compare the relative performance of the languages in question. The following metrics will be measured.

- Execution time
- Consistency
- Memory usage
- Production time

It is reasonable to assume that imperative languages easily outperform Clojure in terms of execution time on hardware which has fewer cores. Therefore it would also be useful to determine how many cores are required before Clojure:

1. Matches the imperative languages for speed.
2. Begins to assert execution time dominance.

Furthermore, it is often the case that languages perform optimally on a specific platform. Hence, ensuring that the code is potable to to different operating systems allows for further analysis of which conditions best facilitate Clojure.

4.2 Specification

4.2.1 Feature set

- **CPU Intensive Algorithm:** The application attempts to solve a problem which will push the CPU of the host machine to its limit for an extended period of time.
- **Multi-Threaded:** The application uses threads in order to take advantage of multiple CPUS.
- **Shared Data Reads:** The application utilises shared data structures which are read from by multiple threads.
- **Shared Data Writes:** The application utilises shared data structures which are written to by multiple threads.
- **Consistency Checking of Shared Data Access:** The application can check its shared data structure for instances of inconsistent reads/writes.
- **Execution Time Logging:** The application can log it's overall execution time and the execution time of individual threads.
- **Memory Usage Logging:** The application can log it's minimum, maximum and mean memory usage over the course of its execution.
- **Cross-Platform Implementation:** The application is executable on different operating systems and architectures in order to determine the most suitable environment for execution.
- **Test Suite:** The application is validated by a suite of unit tests in order to confirm that it operates correctly.

- **Reasonable Execution Time:** A run of the application lasts no more than five minutes, enabling a greater number of runs to be executed in a given period of time.

4.2.2 MoSCoW

In order to correctly prioritise program features and ensure that the most important requirements are satisfied immediately, the MoSCoW method of feature prioritisation is applied to the feature set (Clegg, 1994).

Must: Execution Time Logging, Multi-Threaded, Shared Data Reads, Shared Data Writes

Should: Test Suite, Consistency Checking of Shared Data Access

Could: CPU Intensive Algorithm, Reasonable Execution Time

Wont: Memory Usage Logging, Cross-Platform Implementation

4.3 Test Specification

4.3.1 Control Group

In order to measure how effective Clojure is, its performance is to be benchmarked in four metrics:

- **Execution Time :** How quickly an application completes..
- **Consistency :** The ability of the system to not produce errors when multiple threads are accessing shared data.
- **Memory Usage :** The amount of memory consumed during execution.
- **Production Time :** the length of time taken to implement the application.

Clojure will be pitted against a control group, consisting of three different imperative languages, namely:

- **C++ :** In theory the fastest executing language of the group, however also the most difficult to program with, and the most likely to produce inconsistent results.
- **Java :** The language that Clojure was originally written in and arguably the closest in terms of performance.
- **Python :** Another popular development language, especially for web based applications and scripting on Linux based systems.

4.3.2 Methodology

The Application A concurrent application will be developed in both Clojure and each of the control group languages. The application will perform an identical

task in each language, and will involve shared memory, which will be subjected to read/write access from multiple threads.

Production Time The time spent in minutes will be recorded in order to satisfy the metric of which language is most suited to intuitive concurrent programming. Unfortunately, as there is only one developer producing all four applications, this metric is somewhat subjective and should be taken lightly.

Execution of Applications Each of the control group and Clojure applications will be subjected to a series of 100 runs upon several machines of varying architectures and processing capabilities. This is to determine how well Clojure performs in relation to the control group in several different environments.

Execution Time The time taken in milliseconds to complete a run of the application. Will be logged at the end of each run and subjected to statistical analysis.

Consistency The application will log the consistency of the shared memory, checking it's final state against a set of pre-determined values. Any deviation from the predetermined values will confirm inconsistency. Deviations will be logged and sbjected to statistical analysis.

Memory Usage The minimum, maximum and mean memory usage will be logged upon each execution run and subjected to statistical analysis.

Statistical Analysis Each set of 100 runs will be analysed for it's mean value, standard deviation and P-Value. Likewise, the cumulative results of all runs for

each of the control group and Clojure will be subjected to an overall statistical analysis, in order to determine the best overall concurrent language.

References

Abelson, H. & Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*, 2nd edn, The MIT Press.

Agarwal, A. & Levy, M. (2007). The KILL Rule for Multicore, *DAC '07 Proceedings of the 44th annual Design Automation Conference*, ACM New York, NY, USA, pp. 750–753.

Amdahl, G. M. (1967). Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities, *Proc. Am. Federation of Information Processing Societies Conf.*, AFIPS Press, pp. 483–485.

Backus, J. W. (1959). The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference, *Proceedings of the International Conference on Information Processing*, UNESCO, pp. 125–132.

Barendregt, H., Manzonetto, G. & Plasmeijr, R. (2013). The Imperative and Functional Programming Paradigm, *Alan Turing - His Work and Impact*, Elsevier.

Borkar, S. (2007). Thousand Core Chips-A Technology Perspective, *DAC '07 Proceedings of the 44th annual Design Automation Conference*, ACM New York, NY, USA, pp. 746–749.

Church, A. (1932). A set of Postulates for the Foundation of Logic, *Annals of Mathematics, second series* pp. 346–366.

Church, A. (1936). An Unsolvable Problem of Elementary Number Theory, *American Journal of Mathematics* pp. 346–366.

Clegg, D. (1994). *CASE Method Fast-Track: A RAD Approach*, 1st edn, Addison Wesley.

Copeland, J. (2004). The Church-Turing Thesis, *NeuroQuantology* **2**(2): 101–115.

Gustafson, J. L. (1988). Reevaluating Amdahl's Law, *Communications of the ACM* **31**(5): 532–533.

Hill, M. D. & Marty, M. R. (2008). Amdahl's Law in the Multicore Era, *Computer* **41**(7): 33–38.

Hinsen, K. (2009). The Promises of Functional Programming, *Computing in Science & Engineering* **11**(4): 86–90.

Hwu, W., Keutzer, K. & Mattson, T. G. (2008). The Concurrency Challenge, *IEEE Design & Test of Computers* **25**(4): 312–320.

Kadin, M. & Reda, S. (2008). Frequency and Voltage Planning for Multi-Core Processors Under Thermal Constraints, *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, IEEE, pp. 463–470.

Mack, C. A. (2003). The end of the Semiconductor Industry As We Know It, *Proceedings of SPIE: Optical Microlithography XVI*, Vol. 5040, SPIE, pp. 21–31.

Mack, C. A. (2011). Fifty Years of Moore's Law, *Semiconductor Manufacturing, IEEE Transactions on* **24**(2): 202–207.

McCarthy, J. (1981). History of LISP, *History of Programming Languages I*, ACM New York, NY, USA, pp. 173–185.

Niemeyer, P. & Knudsen, J. (2005). *Learning Java*, 3rd edn, O'Reilly Media, Inc.

Parkinson, C. N. (1957). *Parkinson's Law and other studies in administration*, Houghton Mifflin Company, Boston.

Peyton Jones, S. L. (1989). Parallel Implementations of Functional Programming Languages, *The Computer Journal* **32**(2): 175–186.

Prechelt, L. (2000). An Empirical Comparison of Seven Programming Languages, *Computer* **33**(10): 23–29.

Rosser, J. B. (1984). Highlights of the History of the Lambda-Calculus, *Annals of the History of Computing* **6**(4): 337–349.

Sutter, H. (2005). The free lunch is over: a fundamental turn toward concurrency in software, *Dr Dobb's Journal* **30**(3). `http://www.gotw.ca/publications/concurrency-ddj.htm`: [Online; accessed 20-October-2012].

Sutter, H. & Larus, J. (2005). Software and the Concurrency Revolution, *Queue* **3**(7): 54–62.

Turing, A. (1936). On Computable Numbers, With an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, number 42 in *2*, pp. 230–265.